iPhone Application Programming Guide

General

2010-03-24

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Chapter 1 The Core Application 17

Chapter 2 Window and Views 47

Chapter 3 Event Handling 77

Chapter 4 Graphics and Drawing 107

Chapter 5 Text and Web 119

Introduction

Note: This document was previously titled *iPhone OS Programming Guide*.

The iPhone SDK provides the tools and resources needed to create native iPhone applications that appear as icons on the user's Home screen. Unlike a web application, which runs in Safari, a native application runs directly as a standalone executable on an iPhone OS–based device. Native applications have access to all the features that make the iPhone and iPod touch interesting, such as the accelerometers, location service, and Multi-Touch interface. They can also save data to the local file system and even communicate with other installed applications through custom URL schemes.

In iPhone OS, you develop native applications using the UIKit framework. This framework provides fundamental infrastructure and default behavior that makes it possible to create a functional application in a matter of minutes. Even though the UIKit framework (and other frameworks on the system) provide a significant amount of default behavior, they also provide hooks that you can use to customize and extend that behavior.

Who Should Read This Document?

This document is intended for both new and experienced iPhone OS developers who are creating native iPhone applications. Its purpose is to orient you to the architecture of an iPhone application and to show you the key customization points in the UIKit and other key system frameworks. Along the way, this document also provides guidance to help you make appropriate design choices. It also points out additional documents that may offer advice or further discussion of a given subject.

Although many of the frameworks described in this document are also present in Mac OS X, this document does not assume any familiarity with Mac OS X or its technologies.

Prerequisites

Before you start reading this document, you should have at least a fundamental understanding of the following Cocoa concepts:

- Basic information about Xcode and Interface Builder and their role in developing applications
- How to define new Objective-C classes
- How to manage memory, including how to create and release objects in Objective-C
- The role of delegate objects in managing application behaviors
- The role of the target-action paradigm in managing your user interface

Developers who are new to Cocoa and Objective-C can get information about all of these topics in *Cocoa Fundamentals Guide*.

Development of iPhone applications requires an Intel-based Macintosh computer running Mac OS X v10.5 or later. You must also download and install the iPhone SDK. For information about how to get the iPhone SDK, go to http://developer.apple.com/iphone/.

Organization of This Document

This document has the following chapters:

- "The Core Application" (page 17) contains key information about the basic structure of every iPhone application, including some of the critical tasks every application should be prepared to handle.
- "Window and Views" (page 47) describes the iPhone windowing model and shows you how you use views to organize your user interface.
- "Event Handling" (page 77) describes the iPhone event model and shows you how to handle Multi-Touch and motion-based events. It also shows you how to incorporate copy and paste operations into your applications.
- "Graphics and Drawing" (page 107) describes the graphics architecture of iPhone OS and shows you how to draw shapes and images and incorporate animations into your content.

- "Text and Web" (page 119) describes the text support in iPhone OS, including examples of how you manage the system keyboard.
- "Files and Networking" (page 133) provides guidelines for working with files and network connections.
- "Multimedia Support" (page 145) shows you how to use the audio and video technologies available in iPhone OS.
- "Device Support" (page 167) shows you how to take advantage of external accessories, location tracking services, the accelerometers, and the built-in camera.
- "Application Preferences" (page 195) shows you how to configure your application preferences and display them in the Settings application.

Providing Feedback

If you have feedback about the documentation, you can provide it using the built-in feedback form at the bottom of every page.

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. You can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bugs or enhancement requests, go to the Bug Reporting page of the ADC website, which is at the following URL:

http://developer.apple.com/bugreporter/

You must have a valid ADC login name and password to file bugs. You can obtain a login name for free by following the instructions found on the Bug Reporting page.

See Also

The following documents provide important information that all developers should read prior to developing applications for iPhone OS:

- *iPhone Development Guide* provides important information about the iPhone development process from the tools perspective. This document covers the configuration of devices and the use of Xcode (and other tools) for building, running, and testing your software.
- *Cocoa Fundamentals Guide* provides fundamental information about the design patterns and practices used to develop iPhone applications.
- *iPhone Human Interface Guidelines* provides guidance and important information about how to design your iPhone application's user interface.

The following reference and conceptual documents provide additional information about key iPhone topics:

- *UIKit Framework Reference* and *Foundation Framework Reference* provide reference information for the classes discussed in this document.
- *View Controller Programming Guide for iPhone OS* provides information on the use of view controllers in creating interfaces for iPhone applications.

- *Table View Programming Guide for iPhone OS* provides information about working with table views, which are used frequently in iPhone applications.
- *The Objective-C Programming Language* introduces Objective-C and the Objective-C runtime system, which is the basis of much of the dynamic behavior and extensibility of iPhone OS.

The Core Application

Every iPhone application is built using the UIKit framework and therefore has essentially the same core architecture. UIKit provides the key objects needed to run the application and to coordinate the handling of user input and the display of content on the screen. Where applications deviate from one another is in how they configure these default objects and also where they incorporate custom objects to augment their application's user interface and behavior.

Although customizations to your application's user interface and basic behavior occur down within your application's custom code, there are many customizations that you must make at the highest levels of the application. Because these application-level customizations affect the way your application interacts with the system and other applications installed on a device, it is important to understand when you need to act and when the default behavior is sufficient. This chapter provides an overview of the core application architecture and the high-level customization points to help you make determinations about when to customize and when to use the default behavior.

Important: The UIKit classes are generally not thread safe. All work related to your application's user interface should be performed on your application's main thread.

Core Application Architecture

From the time your application is launched by the user, to the time it exits, the UIKit framework manages the majority of the application's key infrastructure. An iPhone application receives events continuously from the system and must respond to those events. Receiving the events is the job of the `UIApplication` object but responding to the events is the responsibility of your custom code. In order to understand where you need to respond to events, though, it helps to understand a little about the overall life cycle and event cycles of an iPhone application. The following sections describe these cycles and also provide a summary of some of the key design patterns used throughout the development of iPhone applications.

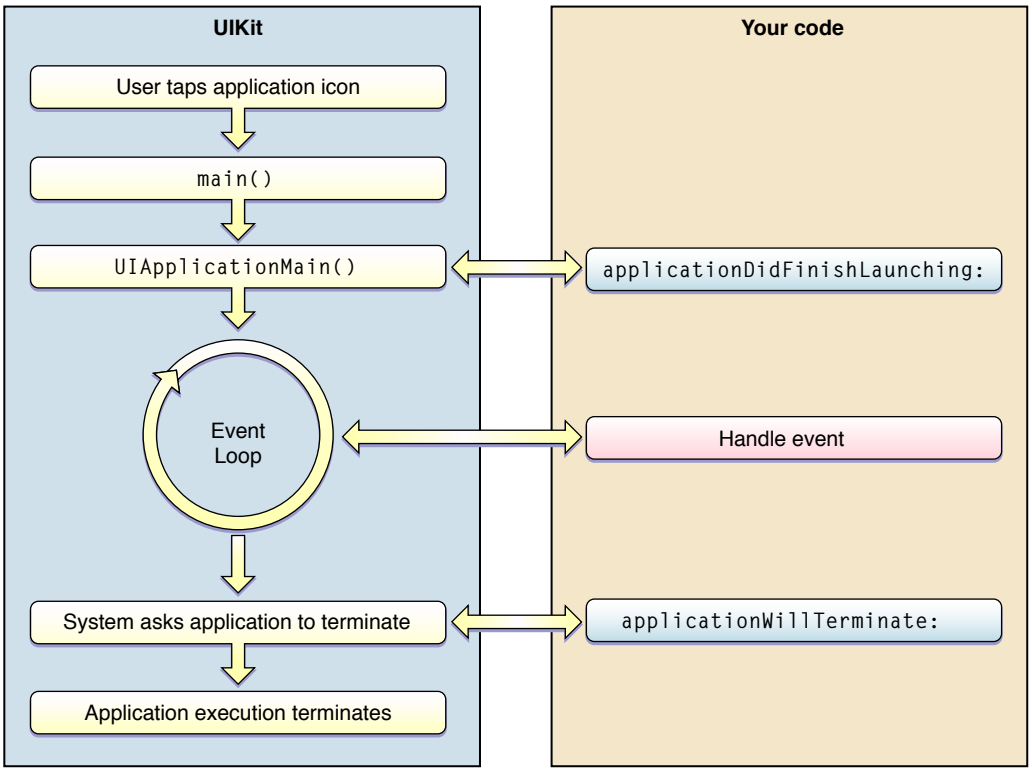
The Application Life Cycle

The application life cycle constitutes the sequence of events that occurs between the launch and termination of your application. In iPhone OS, the user launches your application by tapping its icon on the Home screen. Shortly after the tap occurs, the system displays some transitional graphics and proceeds to launch your application by calling its `main` function. From this point on, the bulk of the initialization work is handed over to UIKit, which loads the application's user interface and readies its event loop. During the event loop, UIKit coordinates the delivery of events to your custom objects and responds to commands issued by your application. When the user performs an action that would cause your application to quit, UIKit notifies your application and begins the termination process.

Figure 1-1 depicts the simplified life cycle of an iPhone application. This diagram shows the sequence of events that occur from the time the application starts up to the time it quits. At initialization and termination, UIKit sends specific messages to the application delegate object to let it know what is happening. During

the event loop, UIKit dispatches events to your application's custom event handlers. Handling initialization and termination events is discussed later in "Initialization and Termination" (page 30), and the event handling process is introduced in"The Event-Handling Cycle" (page 20) and covered in more detail in later chapters.

Figure 1-1 Application life cycle



The Main Function

In an iPhone application, the `main` function is used only minimally. Most of the actual work needed to run the application is handled by the `UIApplicationMain` function instead. As a result, when you start a new application project in Xcode, every project template provides an implementation of the standard `main` function like the one in "Handling Critical Application Tasks." The `main` routine does only three things: it creates an autorelease pool, it calls `UIApplicationMain`, and it releases the autorelease pool. With few exceptions, you should never change the implementation of this function.

Listing 1-1 The `main` function of an iPhone application

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Note: An autorelease pool is used in memory management. It is a Cocoa mechanism used to defer the release of objects created during a functional block of code. For more information about autorelease pools, see *Memory Management Programming Guide for Cocoa*. For specific memory-management guidelines related to autorelease pools in iPhone applications, see "Allocating Memory Wisely" (page 42).

The `UIApplicationMain` function at the heart of the preceding listing takes four parameters and uses them to initialize the application. Although you should never have to change the default values passed into this function, it is worth explaining their purpose in terms of starting the application. In addition to the `argc` and `argv` parameters passed into `main`, this function takes two string parameters that identify the principal class (that is, the class of the application object) and the class of the application delegate. If the value of the principal class string is `nil`, UIKit uses the `UIApplication` class by default. If the value of the application delegate's class is `nil`, UIKit assumes that the application delegate is one of the objects loaded from your application's main nib file (which is the case for applications built using the Xcode templates). Setting either of these parameters to a non-`nil` value causes the `UIApplicationMain` function to create an instance of the corresponding class during application startup and use it for the indicated purpose. Thus, if your application uses a custom subclass of `UIApplication` (which is not recommended, but certainly possible), you would specify your custom class name in the third parameter.

The Application Delegate

Monitoring the high-level behavior of your application is the responsibility of the application delegate object, which is a custom object that you provide. Delegation is a mechanism used to avoid subclassing complex UIKit objects, such as the default `UIApplication` object. Instead of subclassing and overriding methods, you use the complex object unmodified and put your custom code inside the delegate object. As interesting events occur, the complex object sends messages to your delegate object. You can use these "hooks" to execute your custom code and implement the behavior you need.

Important: The delegate design pattern is intended to save you time and effort when creating applications and is therefore a very important pattern to understand. For an overview of the key design patterns used by iPhone applications, see "Fundamental Design Patterns" (page 22). For a more detailed description of delegation and other UIKit design patterns, see *Cocoa Fundamentals Guide*.

The application delegate object is responsible for handling several critical system messages and *must* be present in every iPhone application. The object can be an instance of any class you like, as long as it adopts the `UIApplicationDelegate` protocol. The methods of this protocol define the hooks into the application life cycle and are your way of implementing custom behavior. Although you are not required to implement all of the methods, every application delegate should implement the methods described in "Handling Critical Application Tasks" (page 30).

For additional information about the methods of the `UIApplicationDelegate` protocol, see *UIApplicationDelegate Protocol Reference*.

The Main Nib File

Another task that occurs at initialization time is the loading of the application's main nib file. If the application's information property list (`Info.plist`) file contains the `NSMainNibFile` key, the `UIApplication` object loads the nib file specified by that key as part of its initialization process. The main nib file is the only nib file that is loaded for you automatically; however, you can load additional nib files later as needed.

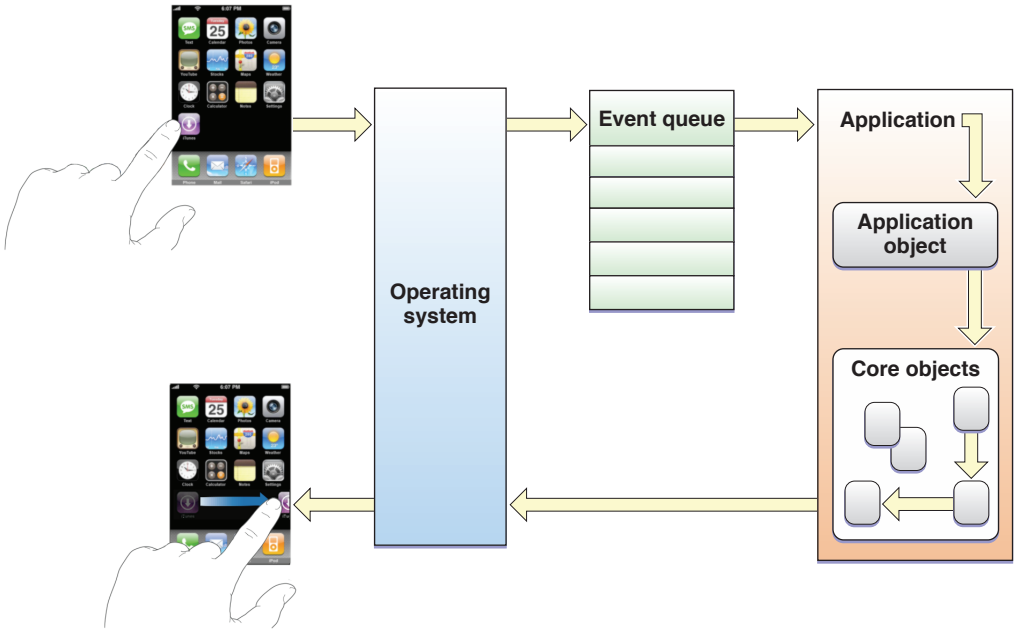
Nib files are disk-based resource files that store a snapshot of one or more objects. The main nib file of an iPhone application typically contains a window object, the application delegate object, and perhaps one or more other key objects for managing the window. Loading a nib file reconstitutes the objects in the nib file, converting each object from its on-disk representation to an actual in-memory version that can be manipulated by your application. Objects loaded from nib files are no different than the objects you create programmatically. For user interfaces, however, it is often more convenient to create the objects associated with your user interface graphically (using the Interface Builder application) and store them in nib files rather than create them programmatically.

For more information about nib files and their use in iPhone applications, see "Nib Files" (page 29). For additional information about how to specify your application's main nib file, see "The Information Property List" (page 26).

The Event-Handling Cycle

After the `UIApplicationMain` function initializes the application, it starts the infrastructure needed to manage the application's event and drawing cycle, which is depicted in Figure 1-2. As the user interacts with a device, iPhone OS detects touch events and places them in the application's event queue. The event-handling infrastructure of the `UIApplication` object takes each event off the top of this queue and delivers it to the object that best suited to handle it. For example, a touch event occurring in a button would be delivered to the corresponding button object. Events can also be delivered to controller objects and other objects indirectly responsible for handling touch events in the application.

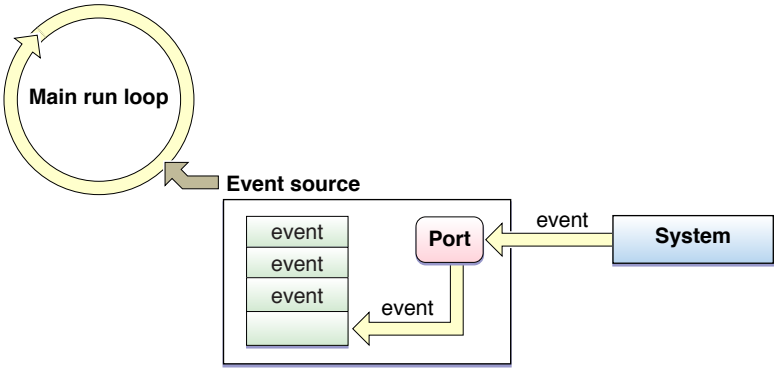
Figure 1-2 The event and drawing cycle



In the iPhone OS Multi-Touch event model, touch data is encapsulated in a single event object (`UIEvent`). To track individual touches, the event object contains touch objects (`UITouch`), one for each finger that is touching the screen. As the user places fingers on the screen, moves them around, and finally removes them from the screen, the system reports the changes for each finger in the corresponding touch object.

When it launches an application, the system creates both a process and a single thread for that application. This initial thread becomes the application's main thread and is where the `UIApplication` object sets up the **main run loop** and configures the application's event-handling code. Figure 1-3 shows the relationship of the event-handling code to the main run loop. Touch events sent by the system are queued until they can be processed by the application's main run loop.

Figure 1-3 Processing events in the main run loop



Note: A run loop monitors sources of input for a given thread of execution. When an input source has data to process, the run loop wakes up the thread and dispatches control to the handler for that input source. When the handler finishes, control passes back to the run loop, which processes the next event or puts the thread to sleep if there is nothing more to do. You can install your own input sources, including ports and timers, on a run loop using the `NSRunLoop` class of the Foundation framework. For more on `NSRunLoop` and run loops in general, see *Threading Programming Guide*.

The `UIApplication` object configures the main run loop with an input source that processes touch events by dispatching them to the appropriate responder objects. A responder object is an object that inherits from the `UIResponder` class and that implements one or more methods for processing the different phases of a touch event. Responder objects in an application include instances of `UIApplication`, `UIWindow`, `UIView`, and all `UIView` subclasses. The application typically dispatches events to the `UIWindow` object representing the application's main window. The window object, in turn, forwards the event to its **first responder**, which is typically the view object (`UIView`) on which the touch took place.

In addition to defining the methods you use to handle events, the `UIResponder` class also defines the programmatic structure of the **responder chain**, which is the Cocoa mechanism for cooperative event handling. The responder chain is a linked series of responder objects in an application, which usually starts at the first responder. If the first responder object cannot handle the event, it passes it to the next responder in the chain. The message continues traveling up the chain—toward higher-level responder objects such as the window, the application, and the application's delegate—until the event is handled. If the event isn't handled, it is discarded.

The responder object that handles the event tends to set in motion a series of programmatic actions that result in the application redrawing all or a portion of its user interface (as well as other possible outcomes, such as the playing of a sound). For example, a control object (that is, a subclass of `UIControl`) handles an event by sending an action message to another object, typically the controller that manages the current set of active views. While processing the action message, the controller might change the user interface or adjust the position of views in ways that require some of those views to redraw themselves. When this happens, the view and graphics infrastructure takes over and processes the required redraw events in the most efficient manner possible.

For more information about events, responders, and how you handle events in your own custom objects, see "Event Handling" (page 77). For information about how windows and views fit into the event-handling scheme, see "The View Interaction Model" (page 52). For additional information about the graphics infrastructure and how views are updated, see "The View Drawing Cycle" (page 107).

Fundamental Design Patterns

The design of the UIKit framework incorporates many of the design patterns found in Cocoa applications in Mac OS X. Understanding these design patterns is crucial to creating iPhone applications, so it is worth taking a few moments to learn about them. The following sections provide a brief overview of these design patterns.

Table 1-1 Design patterns used by iPhone applications

Design pattern	Description
Model-View-Controller	The **Model-View-Controller (MVC)** design pattern is a way of dividing your code into independent functional areas. The **model** portion defines your application's underlying data engine and is responsible for maintaining the integrity of that data. The **view** portion defines the user interface for your application and has no explicit knowledge of the origin of data displayed in that interface. The **controller** portion acts as a bridge between the model and view and facilitates updates between them.
Delegation	The **delegation** design pattern is a way of modifying complex objects without subclassing them. Instead of subclassing, you use the complex object as is and put any custom code for modifying the behavior of that object inside a separate object, which is referred to as the delegate object. At pre-defined times, the complex object then calls the methods of the delegate object to give it a chance to run its custom code.
Target-action	Controls use the **target-action** design pattern to notify your application of user interactions. When the user interacts with a control in a predefined way (such as by tapping a button), the control sends a message (the action) to an object you specify (the target). Upon receiving the action message, the target object can then respond in an appropriate manner (such as by updating application state in response to the button push).
Managed memory model	The Objective-C language uses a reference-counted scheme for determining when to release objects from memory. When an object is first created, it is given a reference count of 1. Other objects can then use the `retain`, `release`, or `autorelease` methods of the object to increase and decrease that reference count appropriately. When an object's reference count reaches 0, the Objective-C runtime calls the object's cleanup routines and then deallocates it.

For a more thorough discussion of these design patterns, see *Cocoa Fundamentals Guide*.

The Application Runtime Environment

The runtime environment of iPhone OS is designed for the fast and secure execution of programs. The following sections describe the key aspects of this runtime environment and provide guidance on how best to operate within it.

Fast Launch, Short Use

The strength of iPhone OS–based devices is their immediacy. A typical user pulls a device out of a pocket or bag and uses it for a few seconds, or maybe a few minutes, before putting it away again. The user might be taking a phone call, looking up a contact, changing the current song, or getting some piece of information during that time.

In iPhone OS, only one foreground application runs at a time. This means that every time the user taps your application's icon on the Home screen, your application must launch and initialize itself quickly to minimize the delay. If your application takes a long time to launch, the user may be less inclined to use it.

In addition to launching quickly, your application must be prepared to exit quickly too. Whenever the user leaves the context of your application, whether by pressing the Home button or by using a feature that opens content in another application, iPhone OS tells your application to quit. At that time, you need to save any unsaved changes to disk and exit as quickly as possible. If your application takes more than 5 seconds to quit, the system may terminate it outright.

Even though your application does not run in the background when the user switches to another application, you are encouraged to make it appear as if that is the case. When your application quits, you should save out information about your application's current state in addition to any unsaved data. At launch time, you should look for this state information and use it to restore your application to the state it was in when it was last used. Doing so provides a more consistent user experience by putting the user right back where they were when they last used your application. Saving the user's place in this way also saves time by potentially eliminating the need to navigate back through multiple screens' worth of information each time an application is launched.

The Application Sandbox

For security reasons, iPhone OS restricts an application (including its preferences and data) to a unique location in the file system. This restriction is part of the security feature known as the application's "sandbox." The **sandbox** is a set of fine-grained controls limiting an application's access to files, preferences, network resources, hardware, and so on. In iPhone OS, an application and its data reside in a secure location that no other application can access. When an application is installed, the system computes a unique opaque identifier for the application. Using a root application directory and this identifier, the system constructs a path to the application's home directory. Thus an application's home directory could be depicted as having the following structure:

/*ApplicationRoot*/*ApplicationID*/

During the installation process, the system creates the application's home directory and several key subdirectories, configures the application sandbox, and copies the application bundle to the home directory. The use of a unique location for each application and its data simplifies backup-and-restore operations,

application updates, and uninstallation. For more information about the application-specific directories created for each application and about application updates and backup-and-restore operations, see "File and Data Management" (page 133).

Important: The sandbox limits the damage an attacker can cause to other applications and to the system, but it cannot prevent attacks from happening. In other words, the sandbox does not protect your application from direct attacks by malicious entities. For example, if there is an exploitable buffer overflow in your input-handling code and you fail to validate user input, an attacker might still be able to crash your program or use it to execute the attacker's code.

The Virtual Memory System

To manage program memory, iPhone OS uses essentially the same virtual memory system found in Mac OS X. In iPhone OS, each program still has its own virtual address space, but (unlike Mac OS X) its usable virtual memory is constrained by the amount of physical memory available. This is because iPhone OS does not write volatile pages to disk when memory gets full. Instead, the virtual memory system frees up volatile memory, as needed, to make sure the running application has the space it needs. It does this by removing memory pages that are not being used and that contain read-only contents, such as code pages. Such pages can always be loaded back into memory later if they are needed again.

If memory continues to be constrained, the system may also send notifications to the running applications, asking them to free up additional memory. All applications should respond to this notification and do their part to help relieve the memory pressure. For information on how to handle such notifications in your application, see "Observing Low-Memory Warnings" (page 32).

The Automatic Sleep Timer

One way iPhone OS attempts to save power is through the automatic sleep timer. If the system does not detect touch events for an extended period of time, it dims the screen initially and eventually turns it off altogether. Although most developers should leave this timer on, game developers and developers whose applications do not use touch inputs can disable this timer to prevent the screen from dimming while their application is running. To disable the timer, set the `idleTimerDisabled` property of the shared `UIApplication` object to `YES`.

Because it results in greater power consumption, disabling the sleep timer should be avoided at all costs. The only applications that should consider using it are mapping applications, games, or applications that do not rely on touch inputs but do need to display visual content on the device's screen. Audio applications do not need to disable the timer because audio content continues to play even after the screen dims. If you do disable the timer, be sure to reenable it as soon as possible to give the system the option to conserve more power. For additional tips on how to save power in your application, see "Reducing Power Consumption" (page 43).

The Application Bundle

When you build your iPhone application, Xcode packages it as a bundle. A **bundle** is a directory in the file system that groups related resources together in one place. An iPhone application bundle contains the application executable and any resources used by the application (for instance, the application icon, other

images, and localized content). Table 1-2 lists the contents of a typical iPhone application bundle, which for demonstration purposes here is called `MyApp`). This example is for illustrative purposes only. Some of the files listed in this table may not appear in your own application bundles.

Table 1-2 A typical application bundle

File	Description
`MyApp`	The executable file containing your application's code. The name of this file is the same as your application name minus the `.app` extension. This file is required.
`Settings.bundle`	The settings bundle is a file package that you use to add application preferences to the Settings application. This bundle contains property lists and other resource files to configure and display your preferences. See "Displaying Application Preferences" (page 38) for more information.
`Icon.png`	The 57 x 57 pixel icon used to represent your application on the device home screen. This icon should not contain any glossy effects. The system adds those effects for you automatically. This file is required. For information about this image file, see "Application Icon and Launch Images" (page 28).
`Icon-Settings.png`	The 29 x 29 pixel icon used to represent your application in the Settings application. If your application includes a settings bundle, this icon is displayed next to your application name in the Settings application. If you do not specify this icon file, the `Icon.png` file is scaled and used instead. For information about this image file, see "Displaying Application Preferences" (page 38).
`MainWindow.nib`	The application's main nib file contains the default interface objects to load at application launch time. Typically, this nib file contains the application's main window object and an instance of the application delegate object. Other interface objects are then either loaded from additional nib files or created programmatically by the application. (The name of the main nib file can be changed by assigning a different value to the `NSMainNibFile` key in the `Info.plist` file. See "The Information Property List" (page 26) for further information.)
`Default.png`	The 480 x 320 pixel image to display when your application is launched. The system uses this file as a temporary background until your application loads its window and user interface. For information about this image file, see "Application Icon and Launch Images" (page 28).
`iTunesArtwork`	The 512 x 512 icon for an application that is distributed using ad-hoc distribution. This icon would normally be provided by the App Store; because applications distributed in an ad-hoc manner do not go through the App Store, however, it must be present in the application bundle instead. iTunes uses this icon to represent your application. (The file you specify for this property should be the same one you would have submitted to the App Store (typically a JPEG or PNG file), were you to distribute your application that way. The filename must be the one shown at left and must not include a filename extension.)
`Info.plist`	Also known as the information property list, this file is a property list defining key values for the application, such as bundle ID, version number, and display name. See "The Information Property List" (page 26) for further information. This file is required.

File	Description
`sun.png` (or other resource files)	Nonlocalized resources are placed at the top level of the bundle directory (`sun.png` represents a nonlocalized image file in the example). The application uses nonlocalized resources regardless of the language setting chosen by the user.

`en.lproj`
`fr.lproj`
`es.lproj`
other language-specific project directories
	Localized resources are placed in subdirectories with an ISO 639-1 language abbreviation for a name plus an `.lproj` suffix. (For example, the `en.lproj`, `fr.lproj`, and `es.lproj` directories contain resources localized for English, French, and Spanish.) For more information, see "Internationalizing Your Application" (page 39).

An iPhone application should be internationalized and have a *language*`.lproj` directory for each language it supports. In addition to providing localized versions of your application's custom resources, you can also localize your application icon (`Icon.png`), default image (`Default.png`), and Settings icon (`Icon-Settings.png`) by placing files with the same name in your language-specific project directories. Even if you provide localized versions, however, you should always include a default version of these files at the top-level of your application bundle. The default version is used in situations where a specific localization is not available.

You use the methods of the `NSBundle` class or the functions associated with the `CFBundleRef` opaque type to obtain paths to localized and nonlocalized image and sound resources stored in the application bundle. For example, to get a path to the image file `sun.png` (shown in "Responding to Interruptions" (page 30)) and create an image file from it would require two lines of Objective-C code:

```
NSString* imagePath = [[NSBundle mainBundle] pathForResource:@"sun"
ofType:@"png"];
UIImage* sunImage = [[UIImage alloc] initWithContentsOfFile:imagePath];
```

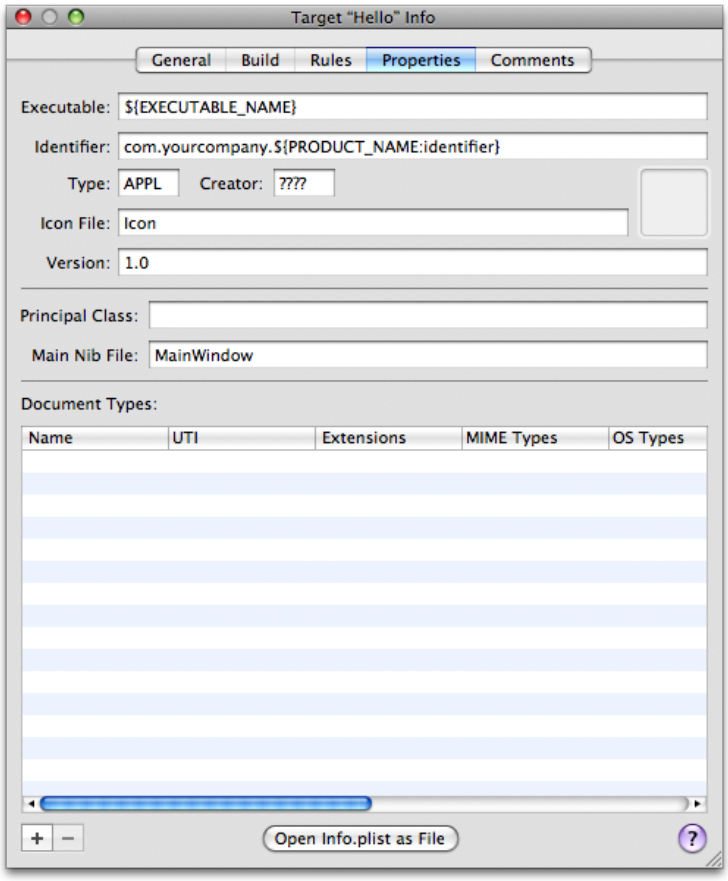
Calling the `mainBundle` class method returns an object representing the application bundle. For information on loading resources, see *Resource Programming Guide*.

The Information Property List

The information property list is a file named `Info.plist` that is included with every iPhone application project created by Xcode. It is a property list whose key-value pairs specify essential runtime-configuration information for the application. The elements of the information property list are organized in a hierarchy in which each node is an entity such as an array, dictionary, string, or other scalar type.

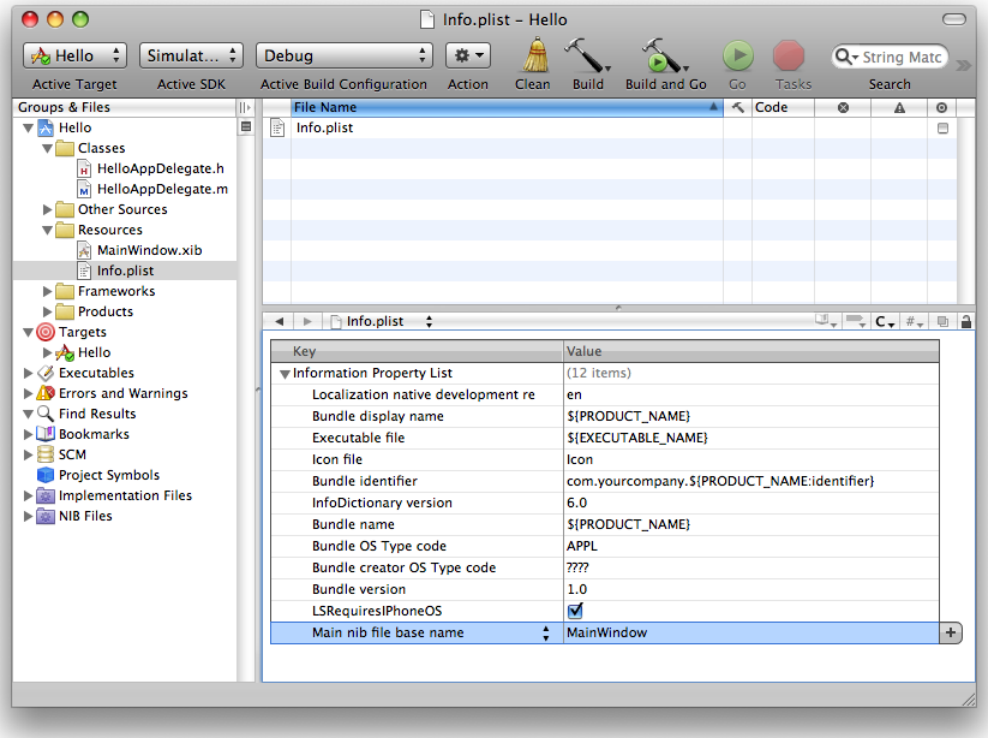
In Xcode, you can access the information property list by choosing Edit Active Target *TargetName* from the Project menu. Then in the target's Info window, click the Properties control. Xcode displays a pane of information similar to the example in Figure 1-4.

Figure 1-4 The Properties pane of a target's Info window



The Properties pane shows you some, but not all, of the properties of the application bundle. When you click the "Open Info.plist as File" button, or when you select the `Info.plist` file in your Xcode project, Xcode displays a property list editor window similar to the one in Figure 1-5. You can use this window to edit the property values and add new key-value pairs. To see the actual key names added to the `Info.plist` file, Control-click the Information Property List item in the editor and select Show Raw Keys/Values from the contextual menu that appears.

Figure 1-5 The information property list editor



Xcode automatically adds some important keys to the Info.plist file of all new projects and sets the initial value. However, there are several keys that iPhone applications use commonly to configure their launch environment and runtime behavior. The following list identifies some of the important keys that you might want to add to your application's `Info.plist` file specifically for iPhone OS.

- `UIStatusBarStyle`
- `UIInterfaceOrientation`
- `UIRequiredDeviceCapabilities`
- `UIRequiresPersistentWiFi`

For detailed information about these and other properties you can include in your application's `Info.plist` file, see *Information Property List Key Reference*.

Application Icon and Launch Images

The file for the icon displayed in the user's Home screen has the default name of `Icon.png` (although the `CFBundleIconFile` property in the `Info.plist` file lets you rename it). It should be a PNG image file located in the top level of the application bundle. The application icon should be a 57 x 57 pixel image without any shine or round beveling effects. Typically, the system applies these effects to the icon before

displaying it. You can override the application of the shine effect, however, by including the `UIPrerenderedIcon` key in your application's `Info.plist` file. (For more information about the `UIPrerenderedIcon` key, see *Information Property List Key Reference*.)

Note: If you are distributing your application to local users using ad-hoc distribution (instead of going through the App Store), your bundle should also include a 512 x 512 pixel version of your application icon in a file called `iTunesArtwork`. This file provides the icon that iTunes displays when distributing your application.

The file for the application's launch image is named `Default.png`. This image should closely resemble the application's initial user interface; the system displays the launch image before an application is ready to display its user interface, giving users the impression of a quick launch. The launch image should also be a PNG image file, located in the top level of the application bundle. If the application is launched through a URL, the system looks for a launch image named `Default-`*scheme*`.png`, where *scheme* is the scheme of the URL. If that file is not present, it chooses `Default.png` instead.

To add an image file to a project in Xcode, choose Add to Project from the Project menu, locate the file in the browser, and click Add.

Note: In addition to the icons and launch image at the top level of your bundle, you can also include localized versions of those images in your application's language-specific project subdirectories. For more information on localizing resources in your application, see "Internationalizing Your Application" (page 39).

Nib Files

A nib file is a data file that stores a set of "freeze-dried" objects that the application plans to use later. Applications use nib files most often to store the windows and views that make up their user interface. When you load a nib file into your application, the nib-loading code turns the contents into real objects that your application can manipulate. In this way, nib files eliminate the need to create these same objects programmatically from your code.

Interface Builder is the visual design environment that you use to create nib files. You assemble nib files using a combination of standard objects (such as the windows and views provided by the UIKit framework) and custom objects from your Xcode projects. Creating view hierarchies within Interface Builder is a simple matter of dragging and dropping views in place. You can also configure the attributes of each object using the inspector window and create connections between objects to define their runtime relationships. All of the changes you make are subsequently saved to disk as part of your nib file.

At runtime, you load nib files into your application when you need the objects they contain. Typically, you load a nib file when your user interface changes and you need to display some new views on the screen. If your application uses view controllers, the view controller handles the nib loading process for you automatically but you can also load nib files yourself using the methods of the `NSBundle` class.

For information on how to design your application's user interface, see *iPhone Human Interface Guidelines*. For information on how to create nib files, see *Interface Builder User Guide*.

Handling Critical Application Tasks

This section describes the handful of tasks that every iPhone application should perform. These tasks are part of the overall life cycle of the application and are therefore part of the key ways your application integrates with iPhone OS. In the worst case, failure to handle some of these tasks could even lead to the termination of your application by the operating system.

Initialization and Termination

During initialization and termination, the `UIApplication` class sends appropriate messages to the application delegate to let it perform any necessary tasks. Although your application is not required to respond to these messages, nearly all iPhone applications should handle them. Initialization time is where you prepare your application's user interface and put the application into its initial running state. Similarly, termination is the time when you should be writing unsaved data and key application state to disk.

Because one iPhone application must quit before another can be launched, the time it takes to execute your initialization and termination code should be as small as possible. Initialization time is not the time to start loading large data structures that you do not intend to use right away. Your goal during startup should be to present your application's user interface as quickly as possible, preferably in the state it was in when your application last quit. If your application requires additional time at launch to load data from the network or do other tasks that might be slow, you should get your interface up and running first and then launch the slow task on a background thread. Doing so gives you the opportunity to display a progress indicator or other feedback to the user to indicate that your application is loading the necessary data or doing something important.

Table 1-3 lists the methods of the `UIApplicationDelegate` protocol that you implement in your application delegate to handle initialization and termination chores. This table also lists some of the key chores you should perform in each method.

Table 1-3 Responsibilities of the application delegate

Delegate method	Description
`applicationDidFinish-Launching:`	Use this method to restore the application to the state it was in during the previous session. You can also use this method to perform any custom initialization to your application data structures and user interface.
`applicationWillTerminate:`	Use this method to save any unsaved data or key application state to disk. You can also use this method to perform additional cleanup operations, such as deleting temporary files.

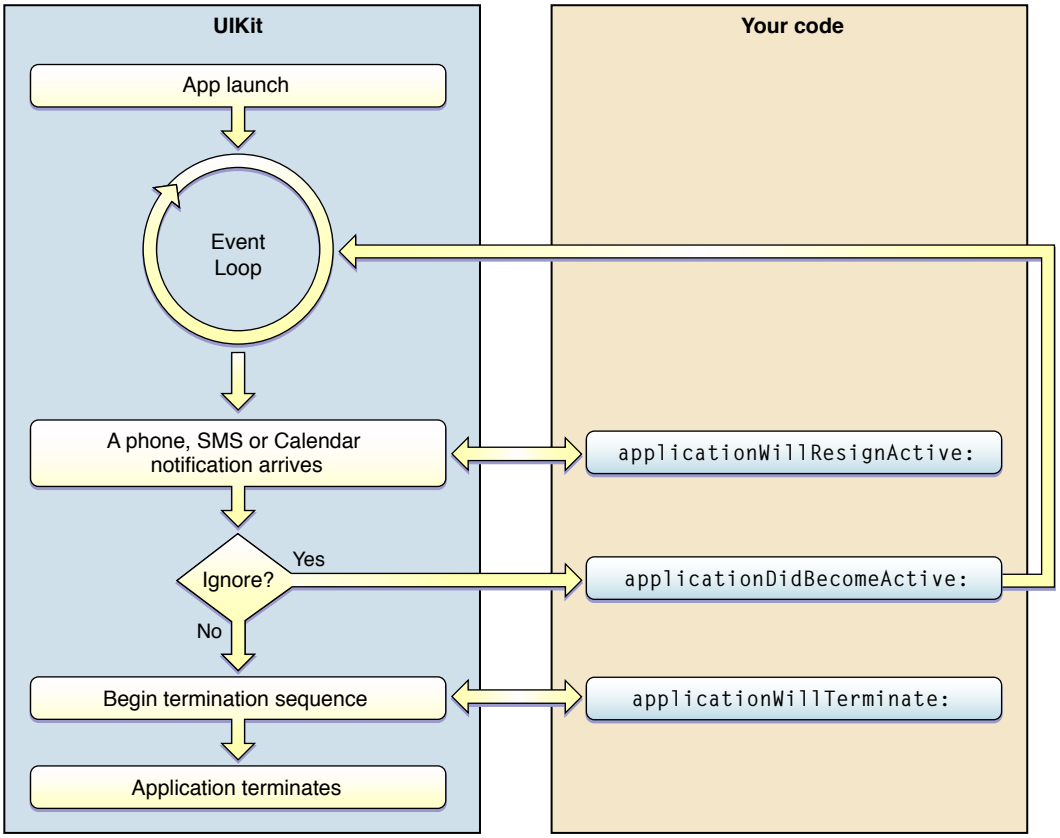
Responding to Interruptions

Besides the Home button, which terminates your application, the system can interrupt your application temporarily to let the user respond to important events. For example, an application can be interrupted by an incoming phone call, an SMS message, a calendar alert, or by the user pressing the Sleep button on a device. Whereas a press of the Home button terminates your application, these interruptions may be only

temporary. If the user ignores the interruption, your application continues running as before. If the user decides to take a call or reply to an SMS message, however, the system does proceed to terminate your application.

Figure 1-6 shows the sequence of events that occurs during the arrival of a phone call, SMS message, or calendar alert. The steps that immediately follow describe the key points in the sequence in more detail, including some of the things your application should do in response to each event. This sequence does not reflect what happens when the user pushes the Sleep/Wake button; that sequence is described after the steps below.

Figure 1-6 The flow of events during an interruption



1. The system detects an incoming phone call or SMS message, or a calendar event occurs.
2. The system calls your application delegate's `applicationWillResignActive:` method. The system also disables the delivery of touch events to your application.

   Interruptions amount to a temporary loss of control by your application. If such a loss of control might affect your application's behavior or cause a negative user experience, you should take appropriate steps in your delegate method to prevent that from happening. For example, if your application is a game, you should pause the game. You should also disable timers, throttle back your OpenGL frame rates (if using OpenGL), and generally put your application into a sleep state. While in the resigned state, your application continues to run but should not do any significant work.
3. The system displays an alert panel with information about the event. The user can choose to ignore the event or respond to it.

4. If the user ignores the event, the system calls your application delegate's `applicationDidBecomeActive:` method and resumes the delivery of touch events to your application.

   You can use this delegate method to reenable timers, throttle up your OpenGL frame rates, and generally wake up your application from its sleep state. For games that are in a paused state, you should consider leaving the game in that state until the user is ready to resume play. For example, you might display an alert panel with controls to resume play.

5. If the user responds to the event, instead of ignoring it, the system calls your application delegate's `applicationWillTerminate:` method. Your application should terminate as usual, saving any needed contextual information to return the user to the same place in your application upon your next launch.

   After terminating your application, the system proceeds to launch the application responsible for the interruption.

Depending on what the user does while responding to an interruption, the system may launch your application again when that interruption ends. For example, if the user takes a phone call and then hangs up, the system relaunches your application. If, while on the call, the user goes back to the Home screen or launches another application, the system does not relaunch your application.

Important: When the user takes a call and then relaunches your application while on the call, the height of the status bar grows to reflect the fact that the user is on a call. Similarly, when the user ends the call, the status bar height shrinks back to its regular size. Your application should be prepared for these changes in the status bar height and adjust its content area accordingly. View controllers handle this behavior for you automatically. If you lay out your user interface programmatically, however, you need to take the status bar height into account when laying out your views and implement the `layoutSubviews` method to handle dynamic layout changes.

If the user presses the Sleep/Wake button on a device while running your application, the system calls your application delegate's `applicationWillResignActive:` method, stops the delivery of touch events, and then puts the device to sleep. When the user wakes the device later, the system calls your application delegate's `applicationDidBecomeActive:` method and begins delivering events to the application again. As you do with other interruptions, you should use these methods to put your application into a sleep state (or pause the game) and wake it up again. While in the sleep state, your application should use as little power as possible.

Observing Low-Memory Warnings

When the system dispatches a low-memory warning to your application, heed it. iPhone OS notifies the frontmost application whenever the amount of free memory dips below a safe threshold. If your application receives this warning, it must free up as much memory as it can by releasing objects it does not need or clearing out memory caches that it can easily recreate later.

UIKit provides several ways to receive low-memory warnings, including the following:

- Implement the `applicationDidReceiveMemoryWarning:` method of your application delegate.
- Override the `didReceiveMemoryWarning` method in your custom `UIViewController` subclass.
- Register to receive the `UIApplicationDidReceiveMemoryWarningNotification` notification.

Upon receiving any of these warnings, your handler method should respond by immediately freeing up any unneeded memory. View controllers should purge any views that are currently offscreen, and your application delegate should release any data structures it can or notify other application objects to release memory they own.

If your custom objects have known purgeable resources, you can have those objects register for the `UIApplicationDidReceiveMemoryWarningNotification`notification and release their purgeable resources directly. Have these objects register if you have a few objects that manage most of your purgeable resources and it is appropriate to purge all of those resources. If you have many purgeable objects or want to coordinate the release of only a subset of those objects, however, you might want to use your application delegate to release the desired objects.

Important: Like the system applications, your applications should always handle low-memory warnings, even if they do not receive those warnings during your testing. System applications consume small amounts of memory while processing requests. When a low-memory condition is detected, the system delivers low-memory warnings to all running programs (including your application) and may terminate some background applications (if necessary) to ease memory pressure. If not enough memory is released—perhaps because your application is leaking or still consuming too much memory—the system may still terminate your application.

Customizing Your Application's Behavior

There are several ways to customize your basic application behavior to provide the user experience you want. The following sections describe some of the customizations that you must make at the application level.

Launching in Landscape Mode

Applications in iPhone OS normally launch in portrait mode to match the orientation of the Home screen. If you have an application that runs in both portrait and landscape modes, your application should always launch in portrait mode initially and then let its view controllers rotate the interface as needed based on the device's orientation. If your application runs in landscape mode only, however, you must perform the following steps to make it launch in a landscape orientation initially.

- In your application's `Info.plist` file, add the `UIInterfaceOrientation` key and set its value to the landscape mode. For landscape orientations, you can set the value of this key to `UIInterfaceOrientationLandscapeLeft` or `UIInterfaceOrientationLandscapeRight`.
- Lay out your views in landscape mode and make sure that their autoresizing options are set correctly.
- Override your view controller's `shouldAutorotateToInterfaceOrientation:` method and return `YES` only for the desired landscape orientation and `NO` for portrait orientations.

Important: The preceding steps assume your application uses view controllers to manage its view hierarchy. View controllers provide a significant amount of infrastructure for handling orientation changes as well as other complex view-related events. If your application is not using view controllers—as may be the case with games and other OpenGL ES–based applications—you are responsible for rotating the drawing surface (or adjusting your drawing commands) as needed to present your content in landscape mode.

The `UIInterfaceOrientation` property hints to iPhone OS that it should configure the orientation of the application status bar (if one is displayed) as well as the orientation of views managed by any view controllers at launch time. In iPhone OS 2.1 and later, view controllers respect this property and set their view's initial orientation to match. Using this property is also equivalent to calling the `setStatusBarOrientation:animated:` method of `UIApplication` early in the execution of your `applicationDidFinishLaunching:` method.

Note: To launch a view controller–based application in landscape mode in versions of iPhone OS prior to v2.1, you need to apply a 90 degree rotation to the transform of the application's root view in addition to all the preceding steps. Prior to iPhone OS 2.1, view controllers did not automatically rotate their views based on the value of the `UIInterfaceOrientation` key. This step is not necessary in iPhone OS 2.1 and later, however.

Communicating with Other Applications

If an application handles URLs of a known type, you can use that URL scheme to communicate with the application. In most cases, however, URLs are used simply to launch another application and have it display some information that is relevant to your own application. For example, if your application manages address information, you could send a URL containing a given address to the Maps application to show that location. This level of communication creates a much more integrated environment for the user and alleviates the need for your application to implement features that exist elsewhere on the device.

Apple provides built-in support for the `http`, `mailto`, `tel`, and `sms` URL schemes. It also supports `http`–based URLs targeted at the Maps, YouTube, and iPod applications. Applications can register their own custom URL schemes as well. To communicate with an application, create an `NSURL` object with some properly formatted content and pass it to the `openURL:` method of the shared `UIApplication` object. The `openURL:` method launches the application that has registered to receive URLs of that type and passes it the URL. When the user subsequently quits that application, the system often relaunches your application but may not always do so. The decision to relaunch an application is made based on the user's actions in the handler application and whether returning to your application would make sense from the user's perspective.

The following code fragment illustrates how one application can request the services of another application ("todolist" in this example is a hypothetical custom scheme registered by an application):

```
NSURL *myURL = [NSURL
URLWithString:@"todolist://www.acme.com?Quarterly%20Report#200806231300"];
[[UIApplication sharedApplication] openURL:myURL];
```

Important: If your URL type includes a scheme that is identical to one defined by Apple, the Apple-provided application is launched instead of your application. If multiple third-party applications register to handle the same URL scheme, it is undefined as to which of the applications is picked to handle URLs of that type.

If your application defines its own custom URL scheme, it should implement a handler for that scheme as described in "Implementing Custom URL Schemes" (page 35). For more information about the system-supported URL schemes, including information about how to format the URLs, see *Apple URL Scheme Reference*.

Implementing Custom URL Schemes

You can register URL types for your application that include custom URL schemes. A custom URL scheme is a mechanism through which third-party applications can interact with each other and with the system. Through a custom URL scheme, an application can make its services available to other applications.

Registering Custom URL Schemes

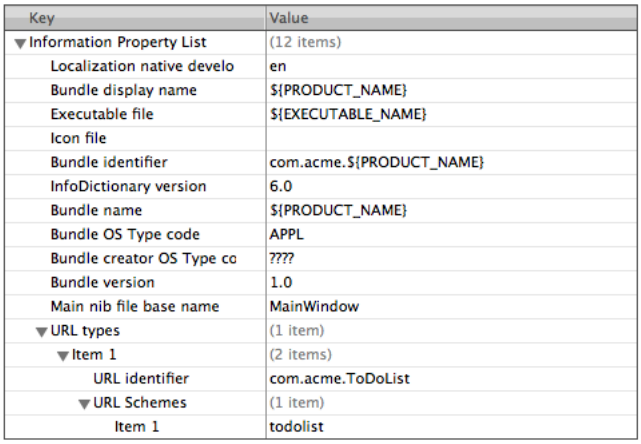
To register a URL type for your application, you must specify the subproperties of the `CFBundleURLTypes` property, which was introduced in "The Information Property List" (page 26). The `CFBundleURLTypes` property is an array of dictionaries in the application's `Info.plist` file, with each dictionary defining a URL type the application supports. Table 1-4 describes the keys and values of a `CFBundleURLTypes` dictionary.

Table 1-4 Keys and values of the `CFBundleURLTypes` property

Key	Value
`CFBundleURLName`	A string that is the abstract name for the URL type. To ensure uniqueness, it is recommended that you specify a reverse-DNS style of identifier, for example, `com.acme.myscheme`. The URL-type name provided here is used as a key to a localized string in the `InfoPlist.strings` file in a language-localized bundle subdirectory. The localized string is the human-readable name of the URL type in a given language.
`CFBundleURLSchemes`	An array of URL schemes for URLs belonging to this URL type. Each scheme is a string. URLs belonging to a given URL type are characterized by their scheme components.

Figure 1-7 shows the `Info.plist` file of an application being edited using the built-in Xcode editor. In this figure, the URL types entry in the left column is equivalent to the `CFBundleURLTypes` key you would add directly to an Info.plist file. Similarly, the "URL identifier" and "URL Schemes" entries are equivalent to the `CFBundleURLName` and `CFBundleURLSchemes` keys.

Figure 1-7 Defining a custom URL scheme in the `Info.plist` file



After you have registered a URL type with a custom scheme by defining the `CFBundleURLTypes` property, you can test the scheme in the following way:

1. Build, install, and run your application.
2. Go to the Home screen and launch Safari. (In the iPhone simulator you can go to the Home screen by selecting Hardware > Home from the menu.)
3. In the address bar of Safari, type a URL that uses your custom scheme.
4. Verify that your application launches and that the application delegate is sent a `application:handleOpenURL:` message.

Handling URL Requests

The delegate of an application handles URL requests routed to the application by implementing the `application:handleOpenURL:` method. You especially need the delegate to implement this method if you have registered custom URL schemes for your application.

A URL request based on a custom scheme assumes a kind of protocol understood by those applications requesting the services of your application. The URL contains information of some kind that the scheme-registering application is expected to process or respond to in some way. Objects of the `NSURL` class, which are passed into the `application:handleOpenURL:` method, represent URLs in the Cocoa Touch framework. `NSURL` conforms to the RFC 1808 specification; it includes methods that return the various parts of a URL as defined by RFC 1808, including user, password, query, fragment, and parameter string. The "protocol" for your custom scheme can use these URL parts for conveying various kinds of information.

In the implementation of `application:handleOpenURL:` shown in Listing 1-2, the passed-in URL object conveys application-specific information in its query and fragment parts. The delegate extracts this information—in this case, the name of a to-do task and the date the task is due—and with it creates a model object of the application.

Listing 1-2 Handling a URL request based on a custom scheme

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url {
```

```
    if ([[url scheme] isEqualToString:@"todolist"]) {
        ToDoItem *item = [[ToDoItem alloc] init];
        NSString *taskName = [url query];
        if (!taskName || ![self isValidTaskString:taskName]) { // must have a
task name
            [item release];
            return NO;
        }
        taskName = [taskName
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

        item.toDoTask = taskName;
        NSString *dateString = [url fragment];
        if (!dateString || [dateString isEqualToString:@"today"]) {
            item.dateDue = [NSDate date];
        } else {
            if (![self isValidDateString:dateString]) {
                [item release];
                return NO;
            }
            // format: yyyymmddhhmm (24-hour clock)
            NSString *curStr = [dateString substringWithRange:NSMakeRange(0,
4)];
            NSInteger yeardigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(4, 2)];
            NSInteger monthdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(6, 2)];
            NSInteger daydigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(8, 2)];
            NSInteger hourdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(10, 2)];
            NSInteger minutedigit = [curStr integerValue];

            NSDateComponents *dateComps = [[NSDateComponents alloc] init];
            [dateComps setYear:yeardigit];
            [dateComps setMonth:monthdigit];
            [dateComps setDay:daydigit];
            [dateComps setHour:hourdigit];
            [dateComps setMinute:minutedigit];
            NSCalendar *calendar = [NSCalendar currentCalendar];
            NSDate *itemDate = [calendar dateFromComponents:dateComps];
            if (!itemDate) {
                [dateComps release];
                [item release];
                return NO;
            }
            item.dateDue = itemDate;
            [dateComps release];
        }

        [(NSMutableArray *)self.list addObject:item];
        [item release];
        return YES;
    }
    return NO;
}
```

Be sure to validate the input you get from URLs passed to your application; see Validating Input in *Secure Coding Guide* to find out how to avoid problems related to URL handling. To learn about URL schemes defined by Apple, see *Apple URL Scheme Reference*.

Displaying Application Preferences

If your application uses preferences to control various aspects of its behavior, how you expose those preferences to the user depends on how integral they are to your program.

- Preferences that are integral to using the application (and simple enough to implement directly) should be presented directly by your application using a custom interface.
- Preferences that are not integral, or that require a relatively complex interface, should be presented using the system's Settings application.

When determining whether a set of preferences is integral, think about the intended usage pattern. If you expect the user to make changes to preferences somewhat frequently, or if those preferences play a relatively important role in how the application behaves, they are probably integral. For example, the settings in a game are usually integral to playing the game and something the user might want to change quickly. Because the Settings application is a separate application, however, you would use it only for preferences that you do not expect the user to access frequently.

If you choose to implement preferences inside your application, it is up to you to define the interface and write the code to manage those preferences. If you choose to use the Settings application, however, your application must provide a Settings bundle to manage them.

A **settings bundle** is a custom resource you include in the top level of your application's bundle directory. An opaque directory with the name `Settings.bundle`, the settings bundle contains specially formatted data files (and supporting resources) that tell the Settings application how to display your preferences. These files also tell the Settings application where to store the resulting values in the preferences database, which your application then accesses using the `NSUserDefaults` or `CFPreferences` APIs.

If you implement your preferences using a settings bundle, you should also provide a custom icon for your preferences. The Settings application looks for an image file with the name `Icon-Settings.png` at the top level of your application bundle and displays that image next to your application name. The image file should be a 29 x 29 pixel PNG image file. If you do not provide this file at the top level of your application bundle, the Settings application uses a scaled version of your application icon (`Icon.png`) by default.

For more information about creating a Settings bundle for your application, see "Application Preferences" (page 195).

Turning Off Screen Locking

If an iPhone OS–based device does not receive touch events for a specified period of time, it turns off the screen and disables the touch sensor. Locking the screen in this way is an important way to save power. As a result, you should leave this feature enabled except when absolutely necessary to prevent unintended behavior in your application. For example, you might disable screen locking if your application does not receive screen events regularly but instead uses other features (such as the accelerometers) for input.

To disable screen locking, set the `idleTimerDisabled` property of the shared `UIApplication` object to `YES`. Be sure to reset this property to `NO` at times when your application does not need to prevent screen locking. For example, you might disable screen locking while the user is playing a game but should reenable it when the user is in setup screens or is otherwise not actively playing the game.

Internationalizing Your Application

Ideally, the text, images, and other content that iPhone applications display to users should be localized for multiple languages. The text that an alert dialog displays, for example, should be in the preferred language of the user. The process of preparing a project for content localized for particular languages is called internationalization. Project components that are candidates for localization include:

- Code-generated text, including locale-specific aspects of date, time, and number formatting
- Static text—for example, an HTML file loaded into a web view for displaying application help
- Icons (including your application icon) and other images when those images either contain text or have some culture-specific meaning
- Sound files containing spoken language
- Nib files

Using the Settings application, users select the language they want to see in their applications' user interfaces from the Language preferences view (see Figure 1-8). They get to this view from the International group of settings, accessed from General settings.

Figure 1-8 The Language preference view

The chosen language is associated with a subdirectory of the bundle. The subdirectory name has two components: an ISO 639-1 language code and a `.lproj` suffix. For example, to designate resources localized to English, the bundle would be named `en.lproj`. By convention, these language-localized subdirectories are called `lproj` directories.

Note: You may use ISO 639-2 language codes instead of those defined by ISO 639-1. However, you should should not include region codes (as defined by the ISO 3166-1 conventions) when naming your `lproj` directories. Although region information is used for formatting dates, numbers, and other types of information, it is not taken into consideration when choosing which language directory to use. For more information about language and region codes, see "Language and Locale Designations" in *Internationalization Programming Topics*.

An `lproj` directory contains all the localized content for a given language. You use the facilities of the `NSBundle` class or the `CFBundleRef` opaque type to locate (in one of the application's `lproj` directories) resources localized for the currently selected language. Listing 1-3 gives an example of such a directory containing content localized for English (`en`).

Listing 1-3 The contents of a language-localized subdirectory

```
en.lproj/
    InfoPlist.strings
    Localizable.strings
    sign.png
```

This subdirectory example has the following items:

- The `InfoPlist.strings` file contains strings assigned as localized values of certain properties in the project's `Info.plist` file (such as `CFBundleDisplayName`). For example, the `CFBundleDisplayName` key for an application named Battleship in the English version would have this entry in `InfoPlist.strings` in the `fr.lproj` subdirectory:

  ```
  CFBundleDisplayName = "Cuirassé";
  ```

- The `Localizable.strings` file contains localized versions of strings generated by application code.
- The `sign.png` file in this example is a file containing a localized image.

To internationalize strings in your code for localization, use the `NSLocalizedString` macro in place of the string. This macro has the following declaration:

```
NSString *NSLocalizedString(NSString *key, NSString *comment);
```

The first parameter is a unique key to a localized string in a `Localizable.strings` file in a given `lproj` directory. The second parameter is a comment that indicates how the string is used and therefore provides additional context to the translator. For example, suppose you are setting the content of a label (`UILabel` object) in your user interface. The following code would internationalize the label's text:

```
label.text = NSLocalizedString(@"City", @"Label for City text field");
```

You can then create a `Localizable.strings` file for a given language and add it to the proper `lproj` directory. For the above key, this file would have an entry similar to the following:

```
"City" = "Ville";
```

Note: Alternatively, you can insert `NSLocalizedString` calls in your code where appropriate and then run the `genstrings` command-line tool. This tool generates a `Localizable.strings` template that includes the key and comment for each string requiring translation. For further information about `genstrings`, see the `genstrings(1)` man page.

For more information about internationalization and how you support it in your iPhone applications, see *Internationalization Programming Topics*.

Tuning for Performance and Responsiveness

At each step in the development of your application, you should consider the implications of your design choices on the overall performance of your application. The operating environment for iPhone applications is more constrained because of the mobile nature of iPhone and iPod touch devices. The following sections describe the factors you should consider throughout the development process.

Do Not Block the Main Thread

You should be careful what work you perform from the main thread of your application. The main thread is where your application handles touch events and other user input. To ensure your application is always responsive to the user, you should never use the main thread to perform long-running tasks or to perform tasks with a potentially unbounded end, such as tasks that access the network. Instead, you should always move those tasks onto background threads. The preferred way to do so is to wrap each task in an operation object and add it to an operation queue, but you can also create explicit threads yourself.

Moving tasks into the background leaves your main thread free to continue processing user input, which is especially important when your application is starting up or quitting. During these times, your application is expected to respond to events in a timely manner. If your application's main thread is blocked at launch time, the system could kill the application before it even finishes launching. If the main thread is blocked at quitting time, the system could kill the application before it has a chance to write out crucial user data.

For more information about using operation objects and threads, see *Concurrency Programming Guide*.

Using Memory Efficiently

Because the iPhone OS virtual memory model does not include disk swap space, applications are somewhat more limited in the amount of memory they have available for use. Using large amounts of memory can seriously degrade system performance and potentially cause the system to terminate your application. When you design, therefore, make it a high priority to reduce the amount of memory used by your application.

There is a direct correlation between the amount of free memory available and the relative performance of your application. Less free memory means that the system is more likely to have trouble fulfilling future memory requests. If that happens, the system can always remove code pages and other nonvolatile resources from memory. However, removing those resources may only be a temporary fix, especially when those resources are needed again a short time later. Instead, minimize your memory use in the first place, and clean up the memory you do use in a timely manner.

The following sections provide more guidance on how to use memory efficiently and how to respond when there is only a small amount of available memory.

Reducing Your Application's Memory Footprint

Table 1-5 lists some tips on how to reduce your application's overall memory footprint. Starting off with a low footprint gives you more room for the data you need to manipulate.

Table 1-5 Tips for reducing your application's memory footprint

Tip	Actions to take
Eliminate memory leaks.	Because memory is a critical resource in iPhone OS, your application should not have any memory leaks. Allowing leaks to exist means your application may not have the memory it needs later. You can use the Instruments application to track down leaks in your code, both in the simulator and on actual devices. For more information on using Instruments, see *Instruments User Guide*.
Make resource files as small as possible.	Files reside on the disk but must be loaded into memory before they can be used. Property list files and images are two resource types where you can save space with some very simple actions. To reduce the space used by property list files, write those files out in a binary format using the `NSPropertyList-Serialization` class. For images, compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iPhone applications—use the `pngcrush` tool.)
Use Core Data or SQLite for large data sets.	If your application manipulates large amounts of structured data, store it in a Core Data persistent store or in a SQLite database instead of in a flat file. Both Core Data and SQLite provides efficient ways to manage large data sets without requiring the entire set to be in memory all at once. The Core Data feature was introduced in iPhone OS 3.0.
Load resources lazily.	You should never load a resource file until it is actually needed. Prefetching resource files may seem like a way to save time, but this practice actually slows down your application right away. In addition, if you end up not using the resource, loading it simply wastes memory.
Build your program using Thumb.	Adding the `-mthumb` compiler flag can reduce the size of your code by up to 35%. Be sure to turn this option off for floating-point-intensive code modules, however, because the use of Thumb on these modules can cause performance to degrade.

Allocating Memory Wisely

iPhone applications use a managed memory model, whereby you must explicitly retain and release objects. Table 1-6 lists tips for allocating memory inside your program.

Table 1-6 Tips for allocating memory

Tip	Actions to take
Reduce your use of autoreleased objects.	Objects released using the `autorelease` method stay in memory until you explicitly drain the autorelease pool or until the next time around your event loop. Whenever possible, avoid using the `autorelease` method when you can instead use the `release` method to reclaim the memory occupied by the object immediately. If you must create moderate numbers of autoreleased objects, create a local autorelease pool and drain it periodically to reclaim the memory for those objects before the next event loop.
Impose size limits on resources.	Avoid loading large resource files when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iPhone OS–based devices. If you must use large resource files, find ways to load only the portion of the file that you need at any given time. For example, rather than load the entire file into memory, use the `mmap` and `munmap` functions to map portions of the file into and out of memory. For more information about mapping files into memory, see *File-System Performance Guidelines*.
Avoid unbounded problem sets.	Unbounded problem sets might require an arbitrarily large amount of data to compute. If the set requires more memory than is available, your application may be unable to complete the calculations. Your applications should avoid such sets whenever possible and work on problems with known memory limits.

For detailed information on how to allocate memory in iPhone applications, and for more information on autorelease pools, see Cocoa Objects in *Cocoa Fundamentals Guide*.

Floating-Point Math Considerations

The processors found in iPhone–OS based devices are capable of performing floating-point calculations in hardware. If you have an existing program that performs calculations using a software-based fixed-point math library, you should consider modifying your code to use floating-point math instead. Hardware-based floating-point computations are typically much faster than their software-based fixed-point equivalents.

Important: Of course, if your code does use floating-point math extensively, remember to compile that code without the `-mthumb` compiler option. The Thumb option can reduce the size of code modules but it can also degrade the performance of floating-point code.

Reducing Power Consumption

Power consumption on mobile devices is always an issue. The power management system in iPhone OS conserves power by shutting down any hardware features that are not currently being used. In addition to avoiding CPU-intensive operations or operations that involve high graphics frame rates, you can help improve battery life by optimizing your use of the following features:

- The CPU
- Wi-Fi and baseband (EDGE, 3G) radios

- The Core Location framework
- The accelerometers
- The disk

The goal of your optimizations should be to do the most work you can in the most efficient way possible. You should always optimize your application's algorithms using Instruments and Shark. But it is important to remember that even the most optimized algorithm can still have a negative impact on a device's battery life. You should therefore consider the following guidelines when writing your code:

- Avoid doing work that requires polling. Polling prevents the CPU from going to sleep. Instead of polling, use the `NSRunLoop` or `NSTimer` classes to schedule work as needed.
- Leave the `idleTimerDisabled` property of the shared `UIApplication` object set to `NO` whenever possible. The idle timer turns off the device's screen after a specified period of inactivity. If your application does not need the screen to stay on, let the system turn it off. If your application experiences side effects as a result of the screen turning off, you should modify your code to eliminate the side effects rather than disable the idle timer unnecessarily.
- Coalesce work whenever possible to maximize idle time. It takes more power to do work periodically over an extended period of time than it does to perform the same amount of work all at once. Performing work periodically prevents the system from powering down hardware over a longer period of time.
- Avoid over-accessing the disk. For example, if your application saves state information to the disk, do so only when that state information changes and coalesce changes whenever possible to avoid writing small changes at frequent intervals.
- Do not draw to the screen faster than needed. Drawing is an expensive operation when it comes to power. Do not rely on the hardware to throttle your frame rates. Draw only as many frames as your application actually needs.
- If you use the `UIAccelerometer` class to receive regular accelerometer events, disable the delivery of those events when you do not need them. Similarly, set the frequency of event delivery to the smallest value that is suitable for your needs. For more information, see "Accessing Accelerometer Events" (page 173).

The more data you transmit to the network, the more power must be used to run the radios. In fact, accessing the network is the most power-hungry operation you can perform and should be minimized by following these guidelines:

- Connect to external network servers only when needed, and do not poll those servers.
- When you must connect to the network, transmit the smallest amount of data needed to do the job. Use compact data formats and do not include excess content that will simply be ignored.
- Transmit data in bursts rather than spreading out transmission packets over time. The system turns off the Wi-Fi and cell radios when it detects a lack of activity. When it transmits data over a longer period of time, your application uses much more power than when it transmits the same amount of data in a shorter amount of time.
- Connect to the network using the Wi-Fi radios whenever possible. Wi-Fi uses less power and is preferred over the baseband radios.
- If you use the Core Location framework to gather location data, disable location updates as soon as you can and set the distance filter and accuracy levels to appropriate values. Core Location uses the available GPS, cell, and Wi-Fi networks to determine the user's location. Although Core Location works hard to

minimize the use of these radios, setting the accuracy and filter values gives Core Location the option to turn off hardware altogether in situations where it is not needed. For more information, see "Getting the User's Current Location" (page 177).

Tuning Your Code

iPhone OS comes with several applications for tuning the performance of your application. Most of these tools run on Mac OS X and are suitable for tuning some aspects of your code while it runs in the simulator. For example, you can use the simulator to eliminate memory leaks and make sure your overall memory usage is as low as possible. You can also remove any computational hotspots in your code that might be caused by an inefficient algorithm or a previously unknown bottleneck.

After you have tuned your code in the simulator, you should then use the Instruments application to further tune your code on a device. Running your code on an actual device is the only way to tune your code fully. Because the simulator runs in Mac OS X, it has the advantage of a faster CPU and more usable memory, so its performance is generally much better than the performance on an actual device. And using Instruments to trace your code on an actual device may point out additional performance bottlenecks that need tuning.

For more information on using Instruments, see *Instruments User Guide*.

Window and Views

Windows and views are the visual components you use to construct the interface of your iPhone application. Windows provide the background platform for displaying content but views do most of the work of drawing that content and responding to user interactions. Although this chapter covers the concepts associated with both windows and views, it focuses more on views because of their importance to the system.

Because views play such a vital role in iPhone applications, there is no way to cover every aspect of them in a single chapter. This chapter focuses on the basic properties of windows and views, their relationships to each other, and how you create and manipulate them in your application. This chapter does not cover how views respond to touch events or draw custom content. For more information about those subjects, see "Event Handling" (page 77) and "Graphics and Drawing" (page 107) respectively.

Important: The UIKit classes are generally not thread safe. All work related to your application's user interface should be performed on your application's main thread.

What Are Windows and Views?

Like Mac OS X, iPhone OS uses windows and views to present graphical content on the screen. Although there are many similarities between the window and view objects on both platforms, the roles played by both windows and views differ slightly on each platform.

The Role of UIWindow

In contrast with Mac OS X applications, iPhone applications typically have only one **window**, represented by an instance of the `UIWindow` class. Your application creates this window at launch time (or loads it from a nib file), adds one or more views to it, and displays it. After that, you rarely need to refer to the window object again.

In iPhone OS, a window object has no visual adornments such as a close box or title bar and cannot be closed or manipulated directly by the user. All manipulations to a window occur through its programmatic interfaces. The application also uses the window to facilitate the delivery of events to your application. For example, the window object keeps track of its current first responder object and dispatches events to it when asked to do so by the `UIApplication` object.

One thing that experienced Mac OS X developers may find unusual about the `UIWindow` class is its inheritance. In Mac OS X, the parent class of `NSWindow` is `NSResponder`. In iPhone OS, the parent class of `UIWindow` is `UIView`. Thus, in iPhone OS, a window is also a view object. Despite its parentage, you typically treat windows in iPhone OS the same as you would in Mac OS X. That is, you typically do not manipulate the view-related properties of a `UIWindow` object directly.

When creating your application window, you should always set its initial frame size to fill the entire screen. If you load your window from a nib file, Interface Builder does not permit you to create a window smaller than the screen size. If you create your window programmatically, however, you must specifically pass in the desired frame rectangle at creation time. There is no reason to pass in any rectangle other than the screen rectangle, which you can get from the `UIScreen` object as shown here:

```
UIWindow* aWindow = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]] autorelease];
```

Although iPhone OS supports layering windows on top of each other, your application should never create more than one window. The system itself uses additional windows to display the system status bar, important alerts, and other types of messages on top of your application's windows. If you want to display alerts on top of your content, use the alert views provided by UIKit rather than creating additional windows.

The Role of UIView

A **view**, an instance of the `UIView` class, defines a rectangular area on the screen. In iPhone applications, views play a key role in both presenting your interface and responding to interactions with that interface. Each view object has the responsibility of rendering content within its rectangular area and for responding to touch events in that area. This dual behavior means that views are the primary mechanism for interacting with the user in your application. In a Model-View-Controller application, view objects are the View portion of the application.

In addition to displaying its own contents and handling events, a view may also manage one or more subviews. A **subview** is simply a view object embedded inside the frame of the original view object, which is referred to as the parent view or **superview**. Views arranged in this manner form what is known as a **view hierarchy** and may contain any number of views. Views can also be nested at arbitrarily deep levels by adding subviews to subviews. The organization of views inside the view hierarchy controls what appears on screen, as each subview is displayed on top of its parent view. The organization also controls how the views react to events and changes. Each parent view is responsible for managing its direct subviews, by adjusting their position and size as needed and even responding to events that its subviews do not handle.

Because view objects are the main way your application interacts with the user, they have a number of responsibilities. Here are just a few:

- Drawing and animation
  - Views draw content in their rectangular area.
  - Some view properties can be animated to new values.
- Layout and subview management
  - Views manage a list of subviews.
  - Views define their own resizing behaviors in relation to their parent view.
  - Views can manually change the size and position of their subviews as needed.
  - Views can convert points in their coordinate system to the coordinate systems of other views or the window.
- Event handling
  - Views receive touch events.

- Views participate in the responder chain.

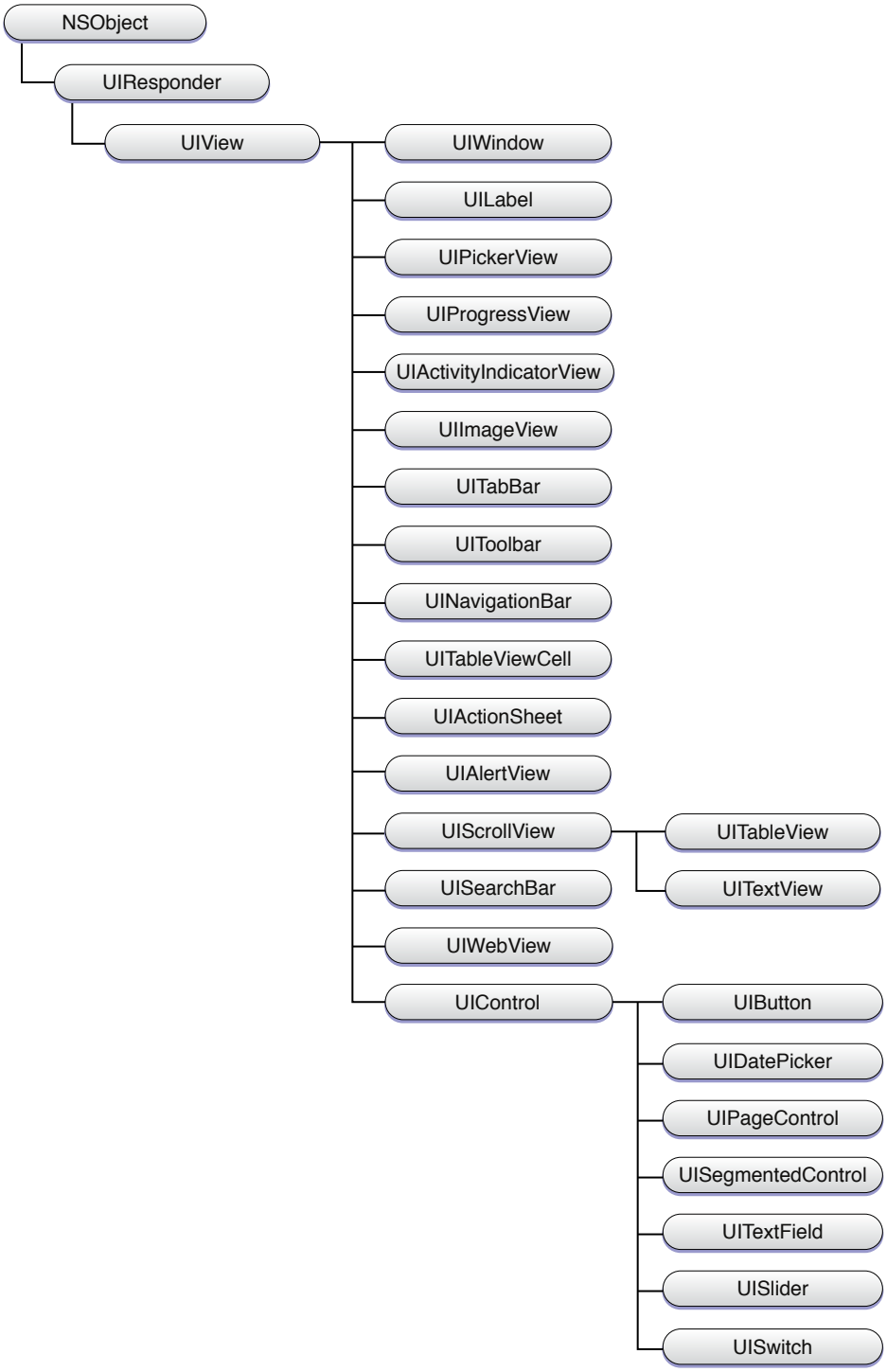
In iPhone applications, views work closely with view controllers to manage several aspects of the views' behavior. View controllers handle the loading and unloading of views, interface rotations caused by the user physically rotating the device, and interactions with the high-level navigation objects used to construct complex user interfaces. For more information, see "The Role of View Controllers" (page 52).

Most of this chapter is dedicated to explaining these responsibilities and showing you how to tie your own custom code into the existing `UIView` behaviors.

UIKit View Classes

The `UIView` class defines the basic properties of a view but not its visual representation. Instead, UIKit uses subclasses to define the specific appearance and behavior for standard system elements such as text fields, buttons, and toolbars. Figure 2-1 shows the class hierarchy diagram for all of the views in UIKit. With the exception of the `UIView` and `UIControl` classes, most of the views in this hierarchy are designed to be used as-is or in conjunction with a delegate object.

Figure 2-1 View class hierarchy



This view hierarchy can be broken down into the following broad categories:

- Containers

Container views enhance the function of other views or provide additional visual separation of the content. For example, the `UIScrollView` class is used to display views whose contents are too large to fit onscreen all at once. The `UITableView` class is a subclass of `UIScrollView` that manages lists of data. Because table rows are selectable, tables are commonly used for hierarchical navigation too—for example, to drill down into a hierarchy of objects.

A `UIToolbar` object is a special type of container that visually groups one or more button-like items. A toolbar typically appears along the bottom of the screen. The Safari, Mail, and Photos applications all use toolbars to display buttons representing frequently used commands. Toolbars can be shown all the time or only as needed by the application.

- Controls

  Controls are used to create most of a typical application's user interface. A control is a special type of view that inherits from the `UIControl` superclass. Controls typically display a specific value and handle all of the user interactions required to modify that value. Controls also use standard system paradigms, such as target-action and delegation, to notify your application when user interactions occur. Controls include buttons, text fields, sliders, and switches.

- Display views

  Although controls and many other types of views provide interactive behavior, some views simply display information. The UIKit classes that exhibit this behavior include `UIImageView`, `UILabel`, `UIProgressView`, and `UIActivityIndicatorView`.

- Text and web views

  Text and web views provide a more sophisticated way to display multiline text content in your application. The `UITextView` class supports the display and editing of multiple lines of text in a scrollable area. The `UIWebView` class provides a way to display HTML content, which lets you incorporate graphics and advanced text-formatting options and lay out your content in custom ways.

- Alert views and action sheets

  Alert views and action sheets are used to get the user's attention immediately. They present a message to the user, along with one or more optional buttons that the user can use to respond to the message. Alert views and action sheets are similar in function but look and behave differently. For example, the `UIAlertView` class displays a blue alert box that pops up on the screen and the `UIActionSheet` class displays a box that slides in from the bottom of the screen.

- Navigation views

  Tab bars and navigation bars work in conjunction with view controllers to provide tools for navigating from one screen of your user interface to another. You typically do not create `UITabBar` and `UINavigationBar` items directly but configure them through the appropriate controller interface or using Interface Builder instead.

- The window

  A window provides a surface for drawing content and is the root container for all other views. There is typically only one window per application. For more information, see "The Role of UIWindow" (page 47).

In addition to views, UIKit provides view controllers to manage those objects. For more information, see "The Role of View Controllers" (page 52).

The Role of View Controllers

Applications running in iPhone OS have many options for organizing their content and presenting it to the user. An application that contains a lot of content might divide that content up into multiple screens' worth of information. At runtime, each screen would then be backed by a set of view objects responsible for displaying the data for that particular screen. The views for a single screen would themselves be backed by a view controller object, whose job is to manage the data displayed by those views and coordinate updates with the rest of the application.

The `UIViewController` class is responsible for creating the set of views it manages and for flushing them from memory during low-memory situations. View controllers also provide automatic responses for some standard system behaviors. For example, in response to a change in the device's orientation, the view controller can resize its managed views to fit the new orientation, if that orientation is supported. You can also use view controllers to display new views modally on top of the current view.

In addition to the base `UIViewController` class, UIKit includes more advanced subclasses for handling some of the sophisticated interface behaviors common to the platform. In particular, navigation controllers manage the display of multiple hierarchical screens worth of content. Tab bar controllers let the user switch between different sets of screens, each of which represents a different operating mode for the application.

For information on how to use view controllers to manage the views in your user interface, see *View Controller Programming Guide for iPhone OS*.

View Architecture and Geometry

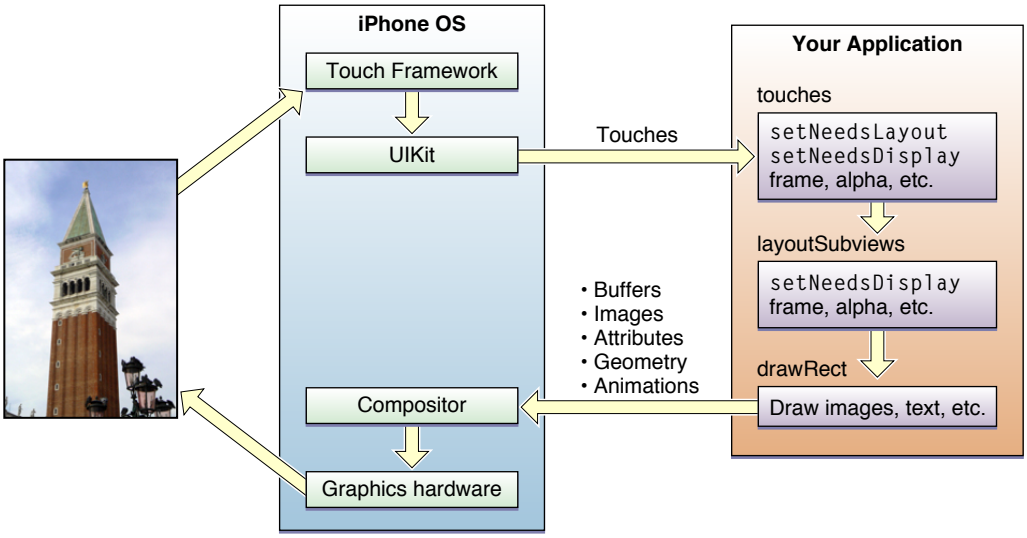
Because views are focal objects in iPhone applications, it is important to understand a little about how views interact with other parts of the system. The standard view classes in UIKit provide a considerable amount of behavior to your application for free. They also provide well-defined integration points where you can customize that behavior and do what you need to do for your application.

The following sections explain the standard behavior of views and call out the places where you can integrate your custom code. For information about the integration points of specific classes, see the reference document for that class. You can get a list of all the class reference documents in *UIKit Framework Reference*.

The View Interaction Model

Any time a user interacts with your user interface, or your own code programmatically changes something, a complex sequence of events takes place inside of UIKit to handle that interaction. At specific points during that sequence, UIKit calls out to your view classes and gives them a chance to respond on behalf of your application. Understanding these callout points is important to understanding where your views fit into the system. Figure 2-2 shows the basic sequence of events that starts with the user touching the screen and ends with the graphics system updating the screen content in response. Programmatic events follow the same basic steps without the initial user interaction.

Figure 2-2 UIKit interactions with your view objects



The following steps break the event sequence in Figure 2-2 (page 53) down even further and explain what happens at each stage and how your application might want to react in response.

1. The user touches the screen.
2. The hardware reports the touch event to the UIKit framework.
3. The UIKit framework packages the touch into a `UIEvent` object and dispatches it to the appropriate view. (For a detailed explanation of how UIKit delivers events to your views, see "Event Delivery" (page 78).)
4. The event-handling methods of your view might respond to the event by doing any of the following:
   - Adjust the properties (frame, bounds, alpha, and so on) of the view or its subviews.
   - Mark the view (or its subviews) as needing a change in its layout.
   - Mark the view (or its subviews) as needing to be redrawn.
   - Notify a controller about changes to some piece of data.

   Of course, it is up to the view to decide which of these things must be done and call the appropriate methods to do it.
5. If a view is marked as requiring layout, UIKit calls the view's `layoutSubviews` method.

   You can override this method in your custom views and use it to adjust the position and size of any subviews. For example, a view that provides a large scrollable area would need to use several subviews as "tiles" rather than create one large view, which is not likely to fit in memory anyway. In its implementation of this method, the view would hide any subviews that are now offscreen or reposition them and use them to draw newly exposed content. As part of this process, the view can also mark the new tiles as needing to be redrawn.
6. If any part of the view is marked as needing to be redrawn, UIKit calls the view's `drawRect:` method.

UIKit calls this method for only those views that need it. Each view's implementation of this method should redraw the specified area as quickly as possible. Each view should draw only its own contents and not the contents of any subviews. Views should not attempt to make any further changes to their properties or layout at this point.

7. Any updated views are composited with the rest of visible content and sent to the graphics hardware for display.

8. The graphics hardware transfers the rendered content to the screen.

Note: The preceding update model applies primarily to applications that use native views and drawing techniques. If your application draws its content using OpenGL ES, you would typically configure a single full-screen view and then draw directly to your OpenGL graphics context. Your view would still handle touch events, but it would not need to lay out subviews or implement a `drawRect:` method. For more information about using OpenGL ES, see "Drawing with OpenGL ES" (page 116).

Given the preceding set of steps, the primary integration points for your own custom views are as follows:

1. These event-handling methods:
   - `touchesBegan:withEvent:`
   - `touchesMoved:withEvent:`
   - `touchesEnded:withEvent:`
   - `touchesCancelled:withEvent:`
2. The `layoutSubviews` method
3. The `drawRect:` method

These are the methods that most custom views implement to get the behavior they want; you may not need to override all of them. For example, if you are implementing a view whose size never changes, you might not need to override the `layoutSubviews` method. Similarly, if you are implementing a view that displays simple content, such as text and images, you can often avoid drawing altogether by simply embedding `UIImageView` and `UILabel` objects as subviews.

It is also important to remember that these are the primary integration points but not the only ones. Several methods of the `UIView` class are designed to be override points for subclassers. You should look at the method descriptions in *UIView Class Reference* to see which methods might be appropriate for you to override in your custom implementations.

The View Rendering Architecture

Although you use views to represent content onscreen, the `UIView` class itself actually relies heavily on another object for much of its basic behavior. Each view object in UIKit is backed by a Core Animation layer object, which is an instance of the `CALayer` class. This layer class provides the fundamental support for the layout and rendering of a view's contents and for compositing and animating that content.

In contrast with Mac OS X (in which Core Animation support is optional) iPhone OS integrates Core Animation into the heart of the view rendering implementation. Although Core Animation has a central role, UIKit streamlines the programming experience by providing a transparent layer on top of Core Animation. This transparent layer eliminates the need to access Core Animation layers directly most of the time, instead letting you access similar behaviors using the methods and properties of the `UIView` class. Where Core Animation becomes important, however, is when the `UIView` class does not provide everything you need. At that point, you can dive down into the Core Animation layers and do some pretty sophisticated rendering for your application.

The following sections provide an introduction to Core Animation and describe some of the features it provides to you for free through the `UIView` class. For more detailed information about how to use Core Animation for advanced rendering, see *Core Animation Programming Guide*.

Core Animation Basics

Core Animation takes advantage of hardware acceleration and an optimized architecture to implement fast rendering and real-time animations. The first time a view's `drawRect:` method is called, the layer captures the results into a bitmap. Subsequent redraw calls use this cached bitmap whenever possible to avoid calling the `drawRect:` method, which can be expensive. This process allows Core Animation to optimize its compositing operations and deliver the desired performance.

Core Animation stores the layers associated with your view objects in a hierarchy referred to as the **layer tree**. Like views, each layer in the layer tree has a single parent and can have any number of embedded sublayers. By default, objects in the layer tree are organized exactly like the views in your view hierarchy. You can add layers, however, without adding a corresponding view. You might do this to implement special visual effects for which a view is not required.

Layer objects are actually the driving force behind the rendering and layout system in iPhone OS, and most view properties are actually thin wrappers for properties on the underlying layer object. When you change the property of a layer in the layer tree (directly using the `CALayer` object), the changed value is reflected immediately in the layer object. If the change triggers a corresponding animation, however, that change may not be reflected onscreen immediately; instead, it must be animated onto the screen over time. To manage these sorts of animations, Core Animation maintains two additional sets of layer objects in what are referred to as the **presentation tree** and the **render tree**.

The presentation tree reflects the state of the layers as they are currently presented to the user. When you animate the changing of a layer value, the presentation layer reflects the old value until the animation commences. As the animation progresses, Core Animation updates the value in the presentation-tree layer based on the current frame of the animation. The render tree then works together with the presentation tree to render the changes on the screen. Because the render tree runs in a separate process or thread, the work it does does not impact your application's main run loop. While both the layer tree and the presentation tree are public, the render tree is a private API.

The placement of layer objects behind your views has many important implications for the performance of your drawing code. The upside to using layers is that most geometry changes to your views do not require redrawing. For example, changing the position and size of a view does not require the system to redraw the contents of a view; it can simply reuse the cached bitmap created by the layer. Animating this cached content is significantly more efficient than trying to redraw that content every time.

The downside to using layers is that the additional cached data can add memory pressure to your application. If your application creates too many views or creates very large views, you could run out of memory quickly. You should not be afraid to use views in your application, but do not create new view objects if you have existing views that can be reused. In other words, pursue approaches that minimize the number of views you keep in memory at the same time.

For a more detailed overview of Core Animation, the object trees, and how you create animations, see *Core Animation Programming Guide*.

Changing the Layer of a View

Because views are required to have an associated layer object in iPhone OS, the `UIView` class creates this layer automatically at initialization time. You can access the layer that is created through the `layer` property of the view, but you cannot change the layer object after the view is created.

If you want a view to use a different type of layer, you must override the view's `layerClass` class method and return the class object for the layer you want it to use. The most common reason to return a different layer class is to implement an OpenGL-based application. To use OpenGL drawing commands, the layer for the underlying view must be an instance of the `CAEAGLLayer` class. This type of layer interacts with the OpenGL rendering calls to present the desired content on the screen.

Important: You should never modify the `delegate` property of a view's layer; that property stores a pointer to the view and should be considered private. Similarly, because a view can operate as the delegate for only one layer, you must not assign it as the delegate for any other layer objects. Doing so will cause your application to crash.

Animation Support

One of the benefits of having a layer object behind every view in iPhone OS is that you can animate content more easily. Remember that animation is not necessarily about creating visual eye candy. Animations provide the user with a context for any changes that occur in your application's user interface. For example, when you use a transition to move from one screen to another, you are indicating to users that the screens are related. The system provides automatic support for many of the most commonly used animations, but you can also create animations for other parts of your interface.

Many properties of the `UIView` class are considered to be animatable. An **animatable** property is one for which there is semiautomatic support for animating from one value to another. You must still tell UIKit that you want to perform the animation, but Core Animation assumes full responsibility for running the animation once it has begun. Among the properties you can animate on a `UIView` object are the following:

```
frame
bounds
center
transform
alpha
```

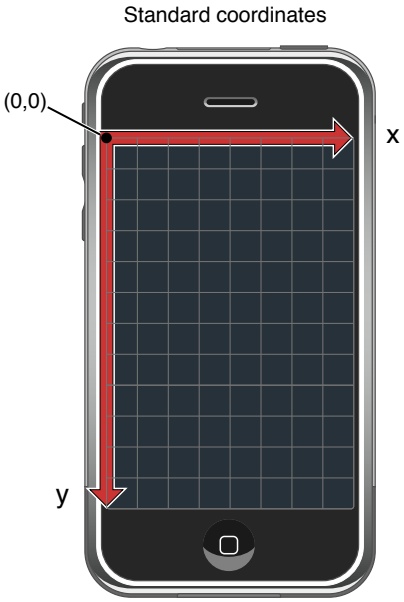
Even though other view properties are not directly animatable, you can create explicit animations for some of them. Explicit animations require you to do more of the work in managing the animation and the rendered contents, but they still use the underlying Core Animation infrastructure to obtain good performance.

For more information about creating animations using the `UIView` class, see "Animating Views" (page 69). For more information about creating explicit animations, see *Core Animation Programming Guide*.

View Coordinate Systems

Coordinates in UIKit are based on a coordinate system whose origin is in the top-left corner and whose coordinate axes extend down and to the right from that point. Coordinate values are represented using floating-point numbers, which allow for precise layout and positioning of content and allow for resolution independence. Figure 2-3 (page 57) shows this coordinate system relative to the screen, but this coordinate system is also used by the `UIWindow` and `UIView` classes. This particular orientation was chosen to make it easier to lay out controls and content in user interfaces, even though it differs from the default coordinate systems in use by Quartz and Mac OS X.

Figure 2-3 View coordinate system



As you write your interface code, be aware of the coordinate system currently in effect. Every window and view object maintains its own local coordinate system. All drawing in a view occurs relative to the view's local coordinate system. The frame rectangle for each view, however, is specified using the coordinate system of its parent view, and coordinates delivered as part of an event object are specified relative to the coordinate system of the enclosing window. For convenience, the `UIWindow` and `UIView` classes each provide methods to convert back and forth between the coordinate systems of different objects.

Although the coordinate system used by Quartz does not use the top-left corner as the origin point, for many Quartz calls this is not a problem. Before invoking your view's `drawRect:` method, UIKit automatically configures the drawing environment to use a top-left origin. Quartz calls made within this environment draw correctly in your view. The only time you need to consider these different coordinate systems is when you set up the drawing environment yourself using Quartz.

For more information about coordinate systems, Quartz, and drawing in general, see "Graphics and Drawing" (page 107).

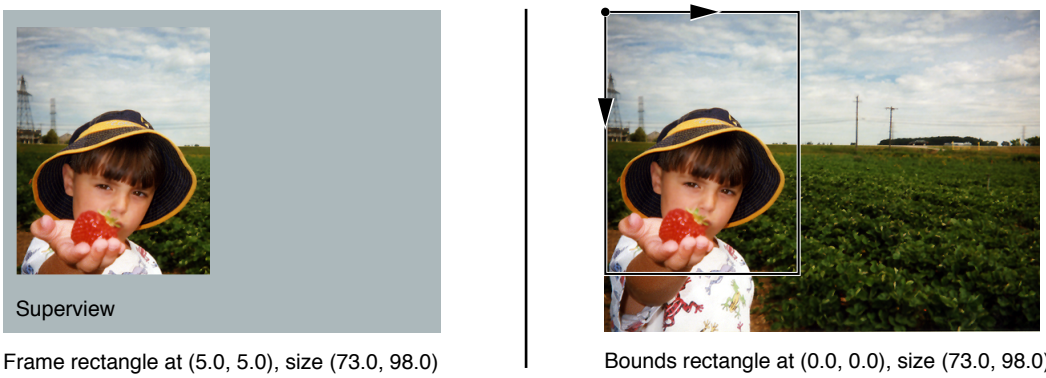
The Relationship of the Frame, Bounds, and Center

A view object tracks its size and location using its `frame`, `bounds`, and `center` properties. The `frame` property contains a rectangle, the **frame rectangle**, that specifies the view's location and size relative to its parent view's coordinate system. The `bounds` property contains a rectangle, the **bounds rectangle**, that defines the view's position and size relative to its own local coordinate system. And although the origin of the bounds rectangle is typically set to (0, 0), it need not be. The `center` property contains the **center point** of the frame rectangle.

You use the `frame`, `bounds`, and `center` properties for different purposes in your code. Because the bounds rectangle represents the view's local coordinate system, you use it most often during drawing or event-handling code when you need to know where in your view something happened. The center point represents the known center point of your view and is always the best way to manipulate the position of your view. The frame rectangle is a convenience value that is computed using the `bounds` and `center` point and is valid only when the view's transform is set to the identity transform.

Figure 2-4 shows the relationship between the frame and bounds rectangles. The complete image on the right is drawn in the view starting at (0, 0). Because the size of the bounds does not match the full size of the image, however, only part of the image outside the bounds rectangle is clipped automatically. When the view is composited with its parent view, the position of the view inside its parent is determined by the origin of the view's frame rectangle, which in this case is (5, 5). As a result, the view's contents appear shifted down and to the right from the parent view's origin.

Figure 2-4 Relationship between a view's frame and bounds



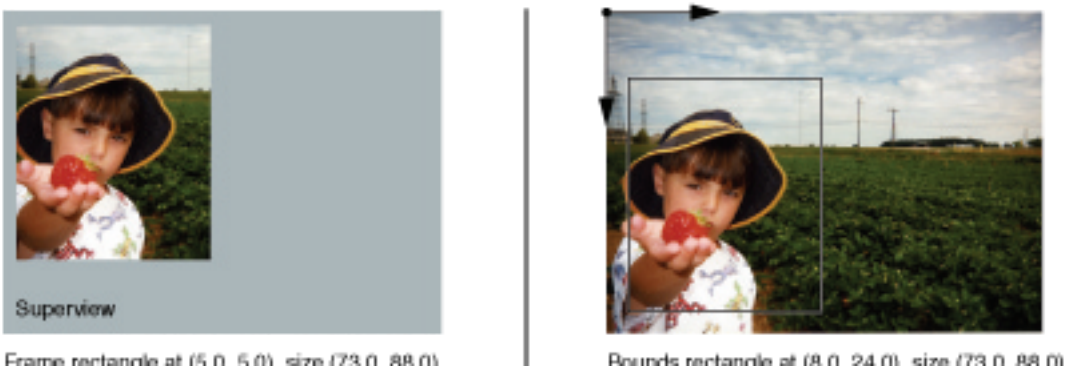
When there is no transform applied to the view, the location and size of the view are determined by these three interrelated properties. The `frame` property of a view is set when a view object is created programmatically using the `initWithFrame:` method. That method also initializes the `bounds` rectangle to originate at (0.0, 0.0) and have the same size as the view's frame. The `center` property is then set to the center point of the frame.

Although you can set the values of these properties independently, setting the value for one changes the others in the following ways:

- When you set the `frame` property, the size of the `bounds` property is set to match the size of the `frame` property. The `center` property is also adjusted to match the center point of the new frame.
- When you set the `center` property, the origin of the `frame` changes accordingly.
- When you set the size of the `bounds` rectangle, the size of the `frame` rectangle changes to match.

You can change the `bounds` origin without changing the other two properties. When you do, the view displays the portion of the underlying image that you have identified. In Figure 2-4 (page 58), the original `bounds` origin is set to (0.0, 0.0). In Figure 2-5, that origin is moved to (8.0, 24.0). As a result, a different portion of the underlying image is displayed by the view. Because the frame rectangle did not change, however, the new content is displayed in the same location inside the parent view as before.

Figure 2-5 Altering a view's bounds



Note: By default, a view's frame is not clipped to its parent view's frame. If you want to force a view to clip its subviews, set the view's `clipsToBounds` property to `YES`.

Coordinate System Transformations

Although coordinate system transformations are commonly used in a view's `drawRect:` method to facilitate drawing, in iPhone OS, you can also use them to implement visual effects for your view. For example, the `UIView` class includes a `transform` property that lets you apply different types of translation, scaling, and zooming effects to the entire view. By default, the value of this property is the identity transform, which causes no changes to the view. To add transformations, get the `CGAffineTransform` structure stored in this property, use the corresponding Core Graphics functions to apply the transformations, and then assign the modified transform structure back to the view's `transform` property.

Note: When applying transforms to a view, all transformations are performed relative to the center point of the view.

Translating a view shifts all subviews along with the drawing of the view's content. Because coordinate systems of subviews inherit and build on these alterations, scaling also affects the drawing of the subviews. For more information about how to control the scaling of view content, see "Content Modes and Scaling" (page 60).

Important: If the `transform` property is not the identity transform, the value of the `frame` property is undefined and must be ignored. After setting the transform, use the `bounds` and `center` properties to get the position and size of the view.

For information about using transforms in conjunction with your `drawRect:` method, see "Coordinates and Coordinate Transforms" (page 108). For information about the functions you use to modify the `CGAffineTransform` structure, see *CGAffineTransform Reference*.

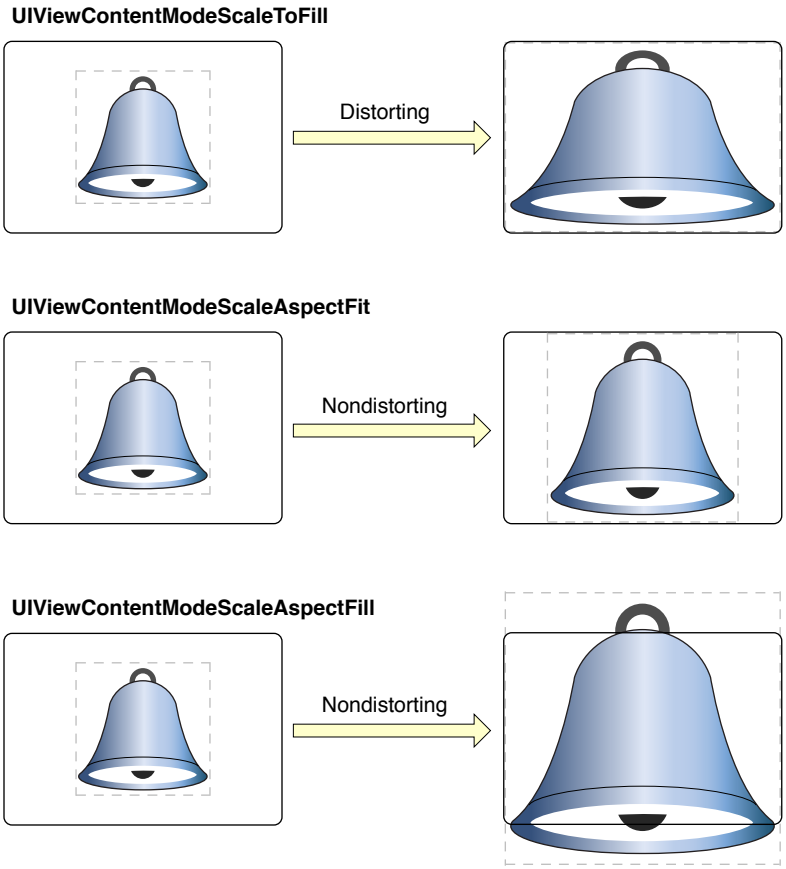
Content Modes and Scaling

When you change the bounds of a view or apply a scaling factor to the `transform` property of a view, the frame rectangle is changed by a commensurate amount. Depending on the content mode associated with the view, the view's content may also be scaled or repositioned to account for the changes. The view's `contentMode` property determines the effect that bounds changes and scaling operations have on the view. By default, the value of this property is set to `UIViewContentModeScaleToFill`, which always causes the view's contents to be scaled to fit the new frame size. For example, Figure 2-6 shows what happens when the horizontal scaling factor of the view is doubled.

Figure 2-6 View scaled using the scale-to-fill content mode

Scaling of your view's content occurs because the first time a view is shown, its rendered contents are cached in the underlying layer. Rather than force the view to redraw itself every time its bounds change or a scaling factor is applied, UIKit uses the view's content mode to determine how to display the cached content. Figure 2-7 compares the results of changing the bounds of a view or applying a scaling factor to it using several different content modes.

Figure 2-7 Content mode comparisons



Although applying a scaling factor always scales the view's contents, there are content modes that do not scale the view's contents when the bounds of the view change. Several `UIViewContentMode` constants (such as `UIViewContentModeTop` and `UIViewContentModeBottomRight`) display the current content in different corners or along different edges of the view. There is also a mode for displaying the content centered inside the view. Changing the bounds rectangle with one of these content modes in place simply moves the existing contents to the appropriate location inside the new bounds rectangle.

Do consider using content modes when you want to implement resizable controls in your application; by doing so you can avoid both control distortion and the writing of custom drawing code. Buttons and segmented controls are particularly suitable for content mode–based drawing. They typically use several images to create the appearance of the control. In addition to having two fixed-size end cap images, a button that can grow horizontally uses a stretchable center image that is only 1 pixel wide. By displaying each image in its own image view and setting the content mode of the stretchable middle image to `UIViewContentModeScaleToFill`, the button can grow in size without distorting the appearance of the end caps. More importantly, the images associated with each image view can be cached by Core Animation and animated without any custom drawing code, which results in much better performance.

Although content modes are good to avoid redrawing the contents of your view, you can also use the `UIViewContentModeRedraw` content mode when you specifically want control over the appearance of your view during scaling and resizing operations. Setting your view's content mode to this value forces Core Animation to invalidate your view's contents and call your view's `drawRect:` method rather than scale or resize them automatically.

Autoresizing Behaviors

When you change the frame rectangle of a view, the position and size of embedded subviews often needs to change to match the new size of the original view. If the `autoresizesSubviews` property of a view is set to `YES`, its subviews are automatically resized according to the values in the `autoresizingMask` property. Often, simply configuring the autoresizing mask for a view provides the appropriate behavior for an application. Otherwise, it is the application's responsibility to reposition and resize the subviews by overriding the `layoutSubviews` method.

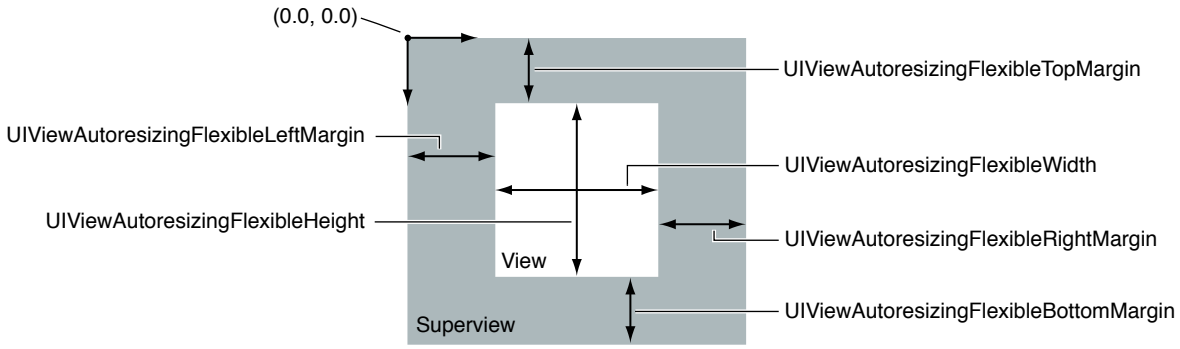
To set a view's autoresizing behaviors, combine the desired autoresizing constants using a bitwise OR operator and assign the resulting value to the view's `autoresizingMask` property. Table 2-1 lists the autoresizing constants and describes how each one affects the size and placement of a given view. For example, to keep a view pinned to the lower-left corner of its superview, add the `UIViewAutoresizingFlexibleRightMargin` and `UIViewAutoresizingFlexibleTopMargin` constants and assign them to the `autoresizingMask` property. When more than one aspect along an axis is made flexible, the resize amount is distributed evenly among them.

Table 2-1 Autoresizing mask constants

Autoresizing mask	Description
`UIViewAutoresizingNone`	If set, the view doesn't autoresize.
`UIViewAutoresizing-FlexibleHeight`	If set, the view's height changes proportionally to the change in the superview's height. Otherwise, the view's height does not change relative to the superview's height.
`UIViewAutoresizing-FlexibleWidth`	If set, the view's width changes proportionally to the change in the superview's width. Otherwise, the view's width does not change relative to the superview's width.
`UIViewAutoresizing-FlexibleLeftMargin`	If set, the view's left edge is repositioned proportionally to the change in the superview's width. Otherwise, the view's left edge remains in the same position relative to the superview's left edge.
`UIViewAutoresizing-FlexibleRightMargin`	If set, the view's right edge is repositioned proportionally to the change in the superview's width. Otherwise, the view's right edge remains in the same position relative to the superview.
`UIViewAutoresizing-FlexibleBottomMargin`	If set, the view's bottom edge is repositioned proportionally to the change in the superview's height. Otherwise, the view's bottom edge remains in the same position relative to the superview.
`UIViewAutoresizing-FlexibleTopMargin`	If set, the view's top edge is repositioned proportionally to the change in the superview's height. Otherwise, the view's top edge remains in the same position relative to the superview.

Figure 2-8 provides a graphical representation of the position of the constant values. When one of these constants is omitted, the view's layout is fixed in that aspect; when a constant is included in the mask, the view's layout is flexible in that aspect.

Figure 2-8 View autoresizing mask constants



If you are using Interface Builder to configure your views, you can set the autoresizing behavior for each view by using the Autosizing controls in the Size inspector. Although the flexible width and height constants from the preceding figure have the same behavior as the Interface Builder springs located in the same position have, the behavior of the margin constants is effectively reversed. In other words, to apply the flexible right margin autoresizing behavior to a view in Interface Builder, you must leave the space on that side of the Autosizing control empty, not place a strut there. Fortunately, Interface Builder provides an animation to show you how changes to the autoresizing behaviors affect your view.

If the `autoresizesSubviews` property of a view is set to `NO`, any autoresizing behaviors set on the immediate subviews of that view are ignored. Similarly, if a subview's autoresizing mask is set to `UIViewAutoresizingNone`, the subview does not change size and so its immediate subviews are never resized either.

Note: For autoresizing to work correctly, the view's `transform` property must be set to the identity transform. The behavior is undefined if it is not.

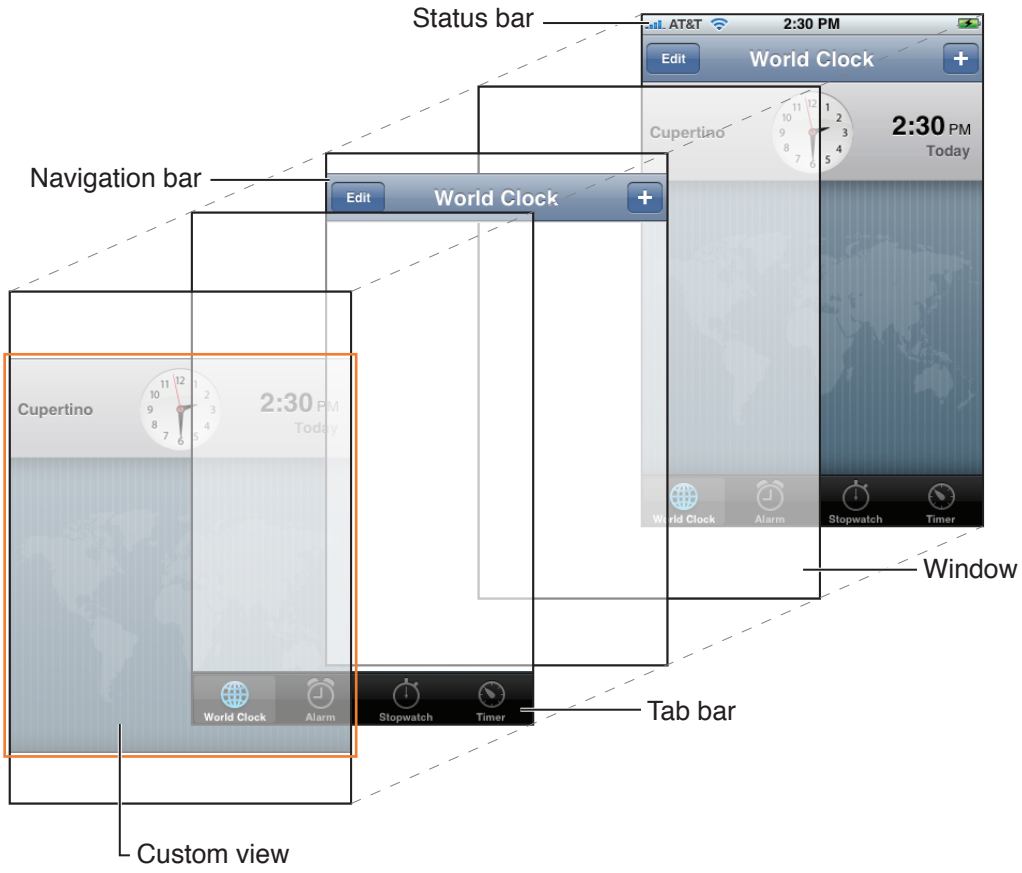
Although autoresizing behaviors may be suitable for some layout needs, if you want more control over the layout of your views, you should override the `layoutSubviews` method in the appropriate view classes. For more information about managing the layout of your views, see "Responding to Layout Changes" (page 71).

Creating and Managing the View Hierarchy

Managing the view hierarchy of your user interface is a crucial part of developing your application's user interface. How you organize your views defines not only the way your application appears visually but also how your application responds to changes. The parent-child relationships in the view hierarchy help define the chain of objects that is responsible for handling touch events in your application. When the user rotates the device, parent-child relationships also help define how each view's size and position are altered by changes to the user interface orientation.

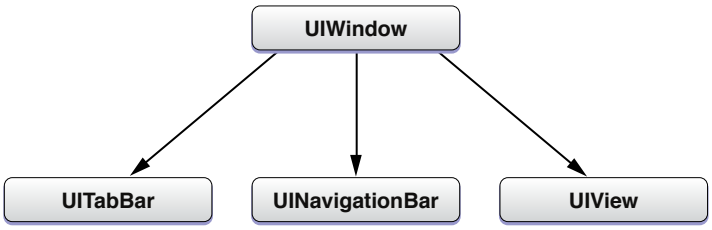
Figure 2-9 shows a simple example of how the layering of views creates a desired visual effect. In the case of the Clock application, tab-bar and navigation-bar views are mixed together with a custom view to implement the overall interface.

Figure 2-9 Layered views in the Clock application



If you look at the object relationships for the views in the Clock application, you see that they look something like the relationships shown in "Changing the Layer of a View." The window object acts as the root view for the application's tab bar, navigation bar, and custom view.

Figure 2-10 View hierarchy for the Clock application



There are several ways to build view hierarchies in iPhone applications, including graphically in Interface Builder and programmatically in your code. The following sections show you how to assemble your view hierarchies and, having done that, how to find views in the hierarchy and convert between different view coordinate systems.

Creating a View Object

The simplest way to create views is to use Interface Builder and load them from the resulting nib file. From Interface Builder's graphical environment, you can drag new views out of the library and drop them onto a window or another view and build your view hierarchies quickly. Because Interface Builder uses live view objects, when you build your interface graphically you see exactly how it will appear when you load it at runtime. And there is no need to write tedious code to allocate and initialize each view in your view hierarchy.

If you prefer not to use Interface Builder and nib files to create your views, you can create them programmatically. To create a new view object, allocate memory for the view object and send that object an `initWithFrame:` message to initialize it. For example, to create a new instance of the `UIView` class, which you could use as a container for other views, you would use the following code:

```
CGRect  viewRect = CGRectMake(0, 0, 100, 100);
UIView* myView = [[UIView alloc] initWithFrame:viewRect];
```

Note: Although all system objects support the `initWithFrame:` message, some may have a preferred initialization method that you should use instead. For information about any custom initialization methods, see the reference documentation for the class.

The frame rectangle that you specify when you initialize the view represents the position and size of the view relative to its intended parent view. You must add views to a window or to another view to make them appear on the screen. When you do, UIKit uses the frame rectangle you specify to place the view inside its parent. For information on how to add views to your view hierarchy, see "Adding and Removing Subviews" (page 65).

Adding and Removing Subviews

Interface Builder is the most convenient way to build view hierarchies because it lets you see exactly how those views will appear at runtime. It then saves the view objects and their hierarchical relationships in a nib file, which the system uses at runtime to recreate the objects and relationships in your application. When a nib file is loaded, the system automatically calls the `UIView` methods needed to recreate the view hierarchy.

If you prefer not to use Interface Builder and nib files to create your view hierarchies, you can create them programmatically instead. A view that has required subviews should create them in its own `initWithFrame:` method to ensure that they are present and initialized with the view. Subviews that are part of your application design (and not required for the operation of your view) should be created outside of your view's initialization code. In iPhone applications, the two most common places to create views and subviews programmatically are the `applicationDidFinishLaunching:` method of your application delegate and the `loadView` method of your view controllers.

To manipulate views in the view hierarchy, you use the following methods:

- To add a subview to a parent, call the `addSubview:` method of the parent view. This method adds the subview to the end of the parent's list of subviews.
- To insert a subview in the middle of the parent's list of subviews, call any of the `insertSubview:...` methods of the parent view.
- To reorder existing subviews inside their parent, call the `bringSubviewToFront:`, `sendSubviewToBack:`, or `exchangeSubviewAtIndex:withSubviewAtIndex:` methods of the parent view. Using these methods is faster than removing the subviews and reinserting them.

- To remove a subview from its parent, call the `removeFromSuperview` method of the subview (not the parent view).

When adding subviews, the current frame rectangle of the subview is used as the initial position of that view inside its parent. You can change that position at any time by changing the `frame` property of the subview. Subviews whose frame lies outside of their parent's visible bounds are not clipped by default. To enable clipping, you must set the `clipsToBounds` property of the parent view to `YES`.

Listing 2-1 shows a sample `applicationDidFinishLaunching:` method of an application delegate object. In this example, the application delegate creates its entire user interface programmatically at launch time. The interface consists of two generic `UIView` objects, which display primary colors. Each view is then embedded inside a window, which is also a subclass of `UIView` and can therefore act as a parent view. Because parents retain their subviews, this method releases the newly created views to prevent them from being overretained.

Listing 2-1 Creating a window with views

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // Create the window object and assign it to the
    // window instance variable of the application delegate.
    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    window.backgroundColor = [UIColor whiteColor];

    // Create a simple red square
    CGRect redFrame = CGRectMake(10, 10, 100, 100);
    UIView *redView = [[UIView alloc] initWithFrame:redFrame];
    redView.backgroundColor = [UIColor redColor];

    // Create a simple blue square
    CGRect blueFrame = CGRectMake(10, 150, 100, 100);
    UIView *blueView = [[UIView alloc] initWithFrame:blueFrame];
    blueView.backgroundColor = [UIColor blueColor];

    // Add the square views to the window
    [window addSubview:redView];
    [window addSubview:blueView];

    // Once added to the window, release the views to avoid the
    // extra retain count on each of them.
    [redView release];
    [blueView release];

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

Important: When you're considering memory management, think of the subviews as any other collection object. Specifically, when you insert a view as a subview using `addSubview:`, that subview is retained by its superview. Inversely, when you remove the subview from its superview using the `removeFromSuperview` method, the subview is autoreleased. Releasing views after adding them to your view hierarchy prevents them being overretained, which could cause memory leaks.

For more information about Cocoa memory management conventions, see *Memory Management Programming Guide for Cocoa*.

When you add a subview to a parent view, UIKit sends several messages to both the parent and child to let them know what is happening. You can override methods such as `willMoveToSuperview:`, `willMoveToWindow:`, `willRemoveSubview:`, `didAddSubview:`, `didMoveToSuperview`, and `didMoveToWindow` in your custom views to process changes before and after they occur and to update the state information in your view accordingly.

After you create a view hierarchy, you can use the `superview` property of a view to get its parent or the `subviews` property to get its children. You can also use the `isDescendantOfView:` method to determine whether a view is in the view hierarchy of a parent view. Because the root view in a view hierarchy has no parent, its `superview` property is set to `nil`. For views currently onscreen, the window object is typically the root view of the hierarchy.

You can use the `window` property of a view to get a pointer to the window that currently contains the view (if any). This property is set to `nil` if the view is not currently attached to a window.

Converting Coordinates in the View Hierarchy

At various times, particularly when handling events, an application may need to convert coordinate values from one frame of reference to another. For example, touch events usually report the touch location using the coordinate system of the window, but view objects need that information in the local coordinate system of the view, which may be different. The `UIView` class defines the following methods for converting coordinates to and from the view's local coordinate system:

```
convertPoint:fromView:
convertRect:fromView:
convertPoint:toView:
convertRect:toView:
```

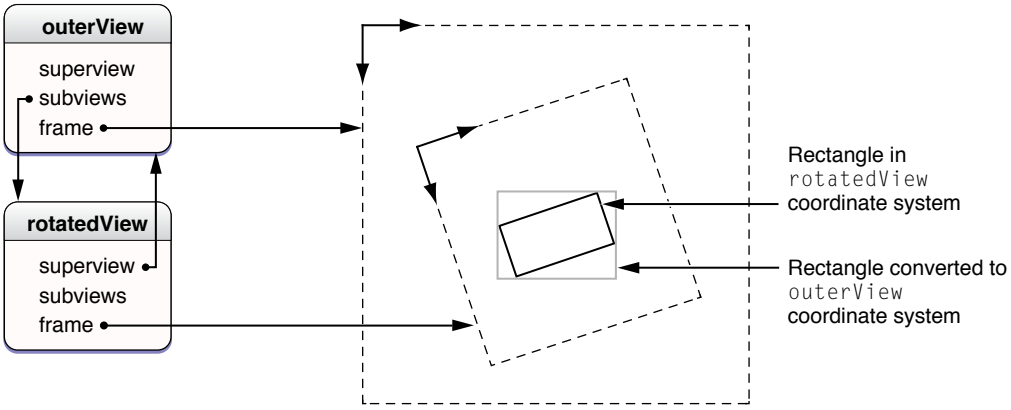
The `convert...:fromView:` methods convert coordinates to the view's local coordinate system, while the `convert...:toView:` methods convert coordinates from the view's local coordinate system to the coordinate system of the specified view. If you specify `nil` as the reference view for any of the methods, the conversions are made to and from the coordinate system of the window that contains the view.

In addition to the `UIView` conversion methods, the `UIWindow` class also defines several conversion methods. These methods are similar to the `UIView` versions except that instead of converting to and from a view's local coordinate system, these methods convert to and from the window's coordinate system.

```
convertPoint:fromWindow:
convertRect:fromWindow:
convertPoint:toWindow:
convertRect:toWindow:
```

Coordinate conversions are straightforward when neither view is rotated or when dealing only with points. When converting rectangles or sizes between views with different rotations, the geometric structure must be altered in a reasonable way so that the resulting coordinates are correct. When converting a rectangle, the `UIView` class assumes that you want to guarantee coverage of the original screen area. To this end, the converted rectangle is enlarged so that when located in the appropriate view, it completely covers the original rectangle. Figure 2-11 shows the conversion of a rectangle in the `rotatedView` object's coordinate system to that of its superview, `outerView`.

Figure 2-11 Converting values in a rotated view



When converting size information, `UIView` simply treats it as a delta offset from (0.0, 0.0) that you need to convert from one view to another. Though the offset distance remains the same, the balance along the two axes shifts according to the rotation. When converting sizes, UIKit always returns sizes that consist of positive numbers.

Tagging Views

The `UIView` class contains a `tag` property that you can use to tag individual view objects with an integer value. You can use tags to uniquely identify views inside your view hierarchy and to perform searches for those views at runtime. (Tag-based searches are faster than iterating the view hierarchy yourself.) The default value for the `tag` property is `0`.

To search for a tagged view, use the `viewWithTag:` method of `UIView`. This method searches the receiver's subviews using a depth-first search, starting with the receiver itself.

Modifying Views at Runtime

As applications receive input from the user, they adjust their user interface in response to that input. An application might rearrange the views in its interface, refresh existing views that contain changed data, or load an entirely new set of views. When deciding which techniques to use, consider your interface and what you are trying to achieve. How you initiate these techniques, however, is the same for all applications. The following sections describe these techniques and how you use them to update your user interface at runtime.

Note: For background information about how UIKit moves events and messages between itself and your custom code, see "The View Interaction Model" (page 52) before proceeding.

Animating Views

Animations provide fluid visual transitions between different states of your user interface. In iPhone OS, animations are used extensively to reposition views, change their size, and even change their alpha value to make them fade in or out. Because this support is crucial for making easy-to-use applications, UIKit simplifies the process of creating animations by integrating support for them directly into the `UIView` class.

The `UIView` class defines several properties that are inherently **animatable**—that is, the view provides built-in support for animating changes in the property from their current value to a new value. Although the work needed to perform the animation is handled for you automatically by the `UIView` class, you must still let the view know that you want the animation to happen. You do this by wrapping changes to the given property in an animation block.

An **animation block** starts with a call to the `beginAnimations:context:` class method of `UIView` and ends with a call to the `commitAnimations` class method. Between these calls, you configure the animation parameters and change the properties you want to animate. As soon as you call the `commitAnimations` method, UIKit performs the animations, animating changes from their current values to the new values you just set. Animation blocks can be nested, but nested animations do not start until the outermost animation block is committed.

Table 2-2 lists the animatable properties of the `UIView` class.

Table 2-2 Animatable properties

Property	Description
`frame`	The view's frame rectangle, in superview coordinates.
`bounds`	The view's bounding rectangle, in view coordinates.
`center`	The center of the frame, in superview coordinates.
`transform`	The transform applied to the view, relative to the center of its bounds.
`alpha`	The view's alpha value, which determines the view's level of transparency.

Configuring Animation Parameters

In addition to changing property values inside an animation block, you can configure additional parameters that determine how you want the animation to proceed. You do this by calling the following class methods of `UIView`:

- Use the `setAnimationStartDate:` method to set the start date of the animations after the `commitAnimations` method returns. The default behavior is to schedule the animation for immediate execution on the animation thread.
- Use the `setAnimationDelay:` method to set a delay between the time the `commitAnimations` method returns and the animations actually begin.

- Use the `setAnimationDuration:` method to set the number of seconds over which the animations occur.
- Use the `setAnimationCurve:` method to set the relative speed of the animations over their course. For example, the animations can gradually speed up at the beginning, gradually slow down near the end, or remain the same speed throughout.
- Use the `setAnimationRepeatCount:` method to set the number of times the animations repeat.
- Use the `setAnimationRepeatAutoreverses:` method to specify whether the animations reverse automatically when they reach their target value. Combined with the `setAnimationRepeatCount:` method, you can use this method to toggle each property between its initial and final values smoothly over a period of time.

The `commitAnimations` class method returns immediately and before the animations begin. UIKit performs animations in a separate thread and away from your application's main event loop. The `commitAnimations` method posts its animations to this separate thread where they are queued up until they are ready to execute. By default, Core Animation finishes the currently running animation block before starting animations currently on the queue. You can override this behavior and start your animation immediately, however, by passing `YES` to the `setAnimationBeginsFromCurrentState:` class method within your animation block. This causes the current in-flight animation to stop and the new animation to begin from the current state.

By default, all animatable property changes within an animation block are animated. If you want to prevent some changes made within the block from being animated, use the `setAnimationsEnabled:` method to disable animations temporarily, make your changes, and then reenable them. Any changes made after a `setAnimationsEnabled:` call with the value `NO` are not animated until a matching call with the value `YES` occurs or you commit the animation block. Use the `areAnimationsEnabled` method to determine whether animations are currently enabled.

Configuring an Animation Delegate

You can assign a delegate to an animation block and use that delegate to receive messages when the animations begin and end. You might do this to perform additional tasks immediately before and after the animation. You set the delegate using the `setAnimationDelegate:` class method of `UIView`, and use the `setAnimationWillStartSelector:` and `setAnimationDidStopSelector:` methods to specify the selectors that will receive the messages. The signatures of the corresponding methods are as follows:

```
- (void)animationWillStart:(NSString *)animationID context:(void *)context;
- (void)animationDidStop:(NSString *)animationID finished:(NSNumber *)finished
 context:(void *)context;
```

The *`animationID`* and *`context`* parameters for both methods are the same parameters that were passed to the `beginAnimations:context:` method at the beginning of the animation block:

- *`animationID`* - an application-supplied string used to identify animations in an animation block.
- *`context`* - another application-supplied object you can use to pass additional information to the delegate.

The `setAnimationDidStopSelector:` selector method has an additional argument—a Boolean value that is `YES` if the animation ran to completion and was not canceled or stopped prematurely by another animation.

Responding to Layout Changes

Whenever the layout of your views changes, UIKit applies each view's autoresizing behaviors and then calls its `layoutSubviews` method to give it a chance to adjust the geometry of its contained subviews further. Layout changes can occur when any of the following happens:

- The size of a view's bounds rectangle changes.
- The content offset value—that is, the origin of the visible content region—of a scroll view changes.
- The transform associated with the view changes.
- The set of Core Animation sublayers associated with the view's layer changes.
- Your application forces layout to occur by calling the `setNeedsLayout` or `layoutIfNeeded` methods of the view.
- Your application forces layout by calling the `setNeedsLayout` method of the view's underlying layer object.

A view's autoresizing behaviors handle the initial job of positioning any subviews. Applying these behaviors guarantees that your views are close to their intended size. For information about how autoresizing behaviors affect the size and position of your views, see "Autoresizing Behaviors" (page 62).

Sometimes, you might want to adjust the layout of subviews manually using `layoutSubviews`, rather than rely exclusively on autoresizing behaviors. For example, if you are implementing a custom control that is built from several subview elements, by adjusting the subviews manually you can precisely configure the appearance for your control over a range of sizes. Alternatively, a view representing a large scrollable content area could display that content by tiling a set of subviews. During scrolling, views going off one edge of the screen would be recycled and repositioned at the incoming screen edge along with any new content.

Note: You can also use the `layoutSubviews` method to adjust the size and position of custom `CALayer` objects attached as sublayers to your view's layer. Managing custom layer hierarchies behind your view lets you perform advanced animations directly using Core Animation. For more information about using Core Animation to manage layer hierarchies, see *Core Animation Programming Guide*.

When writing your layout code, be sure to test your code in each of your application's supported interface orientations. Applications that support both landscape and portrait orientations should verify that layout is handled properly in each orientation. Similarly, your application should be prepared to deal with other system changes, such as the height of the status bar changing. This occurs when a user uses your application while on an active phone call and then hangs up. At hang-up time, the managing view controller may resize its view to account for the shrinking status bar size. Such a change would then filter down to the rest of the views in your application.

Redrawing Your View's Content

Occasionally, changes to your application's data model require that you also change the corresponding user interface. To make those changes, you mark the corresponding views as dirty and in need of an update (using either the `setNeedsDisplay` or `setNeedsDisplayInRect:` methods). Marking views as dirty, as opposed to simply creating a graphics context and drawing, gives the system a chance to process drawing operations more efficiently. For example, if you mark several regions of the same view as dirty during a given cycle, the

system coalesces the dirty regions into a single call to the view's `drawRect:` method. As a result, only one graphics context is created to draw all of the affected regions. This practice is much more efficient than creating several graphics contexts in quick succession.

Views that implement a `drawRect:` method should always check the rectangle passed to the method and use it to limit the scope of their drawing operations. Because drawing is a relatively expensive operation, limiting drawing in this way is a good way to improve performance.

By default, geometry changes to a view do not automatically cause the view to be redrawn. Instead, most geometry changes are handled automatically by Core Animation. Specifically, when you change the `frame`, `bounds`, `center`, or `transform` properties of the view, Core Animation applies the geometry changes to the cached bitmap associated with the view's layer. In many cases, this approach is perfectly acceptable, but if you find the results undesirable, you can force UIKit to redraw your view instead. To prevent Core Animation from applying geometry changes implicitly, set your view's `contentMode` property to `UIViewContentModeRedraw`. For more information about content modes, see "Content Modes and Scaling" (page 60).

Hiding Views

You can hide or show a view by changing the value in the view's `hidden` property. Setting this property to `YES` hides the view; setting it to `NO` shows it. Hiding a view also hides any embedded subviews as if their own `hidden` property were set.

When you hide a view, it remains in the view hierarchy, but its contents are not drawn and it does not receive touch events. Because it remains in the view hierarchy, a hidden view continues to participate in autoresizing and other layout operations. If you hide a view that is currently the first responder, the view does not automatically resign its first responder status. Events targeted at the first responder are still delivered to the hidden view. For more information about the responder chain, see "Responder Objects and the Responder Chain" (page 78).

Creating a Custom View

The `UIView` class provides the underlying support for displaying content on the screen and for handling touch events, but its instances draw nothing but a background color using an alpha value and its subviews. If your application needs to display custom content or handle touch events in a specific manner, you must create a custom subclass of `UIView`.

The following sections describe some of the key methods and behaviors you might implement in your custom view objects. For additional subclassing information, see *UIView Class Reference*.

Initializing Your Custom View

Every new view object you define should include a custom `initWithFrame:` initializer method. This method is responsible for initializing the class at creation time and putting your view object into a known state. You use this method when creating instances of your view programmatically in your code.

Listing 2-2 shows a skeletal implementation of a standard `initWithFrame:` method. This method calls the inherited implementation of the method first and then initializes the instance variables and state information of the class before returning the initialized object. Calling the inherited implementation is traditionally performed first so that if there is a problem, you can simply abort your own initialization code and return `nil`.

Listing 2-2 Initializing a view subclass

```
- (id)initWithFrame:(CGRect)aRect {
    self = [super initWithFrame:aRect];
    if (self) {
          // setup the initial properties of the view
          ...
       }
    return self;
}
```

If you plan to load instances of your custom view class from a nib file, you should be aware that in iPhone OS, the nib-loading code does not use the `initWithFrame:` method to instantiate new view objects. Instead, it uses the `initWithCoder:` method that is defined as part of the `NSCoding` protocol.

Even if your view adopts the `NSCoding` protocol, Interface Builder does not know about your view's custom properties and therefore does not encode those properties into the nib file. As a result, your own `initWithCoder:` method does not have the information it needs to properly initialize the class when it is loaded from a nib file. To solve this problem, you can implement the `awakeFromNib` method in your class and use it to initialize your class specifically when it is loaded from a nib file.

Drawing Your View's Content

As you make changes to your view's content, you notify the system that parts of that view need to be redrawn using the `setNeedsDisplay` or `setNeedsDisplayInRect:` methods. When the application returns to its run loop, it coalesces any drawing requests and computes the specific parts of your interface that need to be updated. It then begins traversing your view hierarchy and sending `drawRect:` messages to the views that require updates. The traversal starts with the root view of your hierarchy and proceeds down through the subviews, processing them from back to front. Views that display custom content inside their visible bounds must implement the `drawRect:` method to render that content.

Before calling your view's `drawRect:` method, UIKit configures the drawing environment for your view. It creates a graphics context and adjusts its coordinate system and clipping region to match the coordinate system and bounds of your view. Thus, by the time your `drawRect:` method is called, you can simply begin drawing using UIKit classes and functions, Quartz functions, or a combination of them all. If you need to access the current graphics context, you can get a pointer to it using the `UIGraphicsGetCurrentContext` function.

Important: The current graphics context is valid only for the duration of one call to your view's `drawRect:` method. UIKit may create a different graphics context for each subsequent call to this method, so you should not try to cache the object and use it later.

Listing 2-3 shows a simple implementation of a `drawRect:` method that draws a 10-pixel-wide red border around the view. Because UIKit drawing operations use Quartz for their underlying implementations, you can mix drawing calls as shown here and still get the results you expect.

Listing 2-3 A drawing method

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGRect    myFrame = self.bounds;

    CGContextSetLineWidth(context, 10);

    [[UIColor redColor] set];
    UIRectFrame(myFrame);
}
```

If you know that your view's drawing code always covers the entire surface of the view with opaque content, you can improve the overall efficiency of your drawing code by setting the `opaque` property of your view to `YES`. When you mark a view as opaque, UIKit avoids drawing content that is located immediately behind your view. This not only reduces the amount of time spent drawing but also minimizes the work that must be done to composite that content together. You should set this property to `YES` only if you know your view provides opaque content. If your view cannot guarantee that its contents are always opaque, you should set the property to `NO`.

Another way to improve drawing performance, especially during scrolling, is to set the `clearsContextBeforeDrawing` property of your view to `NO`. When this property is set to `YES`, UIKIt automatically fills the area to be updated by your `drawRect:` method with transparent black before calling your method. Setting this property to `NO` eliminates the overhead for that fill operation but puts the burden on your application to completely redraw the portions of your view inside the update rectangle passed to your `drawRect:` method. Such an optimization is usually a good tradeoff during scrolling, however.

Responding to Events

The `UIView` class is a subclass of `UIResponder` and is therefore capable of receiving touch events corresponding to user interactions with the view's contents. Touch events start at the view in which the touch occurred and are passed up the responder chain until they are handled. Because views are themselves responders, they participate in the responder chain and therefore can receive touch events dispatched to them from any of their associated subviews.

Views that handle touch events typically implement all of the following methods, which are described in more detail in "Event Handling" (page 77).

```
touchesBegan:withEvent:
touchesMoved:withEvent:
touchesEnded:withEvent:
touchesCancelled:withEvent:
```

Remember that, by default, views respond to only one touch at a time. If the user puts a second finger down, the system ignores the touch event and does not report it to your view. If you plan to track multifinger gestures from your view's event-handler methods, you need to reenable multi-touch events by setting the `multipleTouchEnabled` property of your view to `YES`.

Some views, such as labels and images, disable event handling altogether initially. You can control whether a view handles events at all by changing the value of the view's `userInteractionEnabled` property. You might temporarily set this property to `NO` to prevent the user from manipulating the contents of your view while a long operation is pending. To prevent events from reaching any of your views, you can also use the

`beginIgnoringInteractionEvents` and `endIgnoringInteractionEvents` methods of the `UIApplication` object. These methods affect the delivery of events for the entire application, not just for a single view.

As it handles touch events, UIKit uses the `hitTest:withEvent:` and `pointInside:withEvent:` methods of `UIView` to determine whether a touch event occurred in a given view. Although you rarely need to override these methods, you could do so to implement custom touch behaviors for your view. For example, you could override these methods to prevent subviews from handling touch events.

Cleaning Up After Your View

If your view class allocates any memory, stores references to any custom objects, or holds resources that must be released when the view is released, you must implement a `dealloc` method. The system calls the `dealloc` method when your view's retain count reaches zero and your view is about to be deallocated itself. Your implementation of this method should release the objects and resources it holds and then call the inherited implementation, as shown in Listing 2-4.

Listing 2-4 Implementing the `dealloc` method

```
- (void)dealloc {
    // Release a retained UIColor object
    [color release];

    // Call the inherited implementation
    [super dealloc];
}
```

Event Handling

This chapter describes the types of events in iPhone OS, and explains how to handle them. It also discusses how to copy and paste data within an application or between applications using the facilities of the `UIPasteboard` class, which was introduced in iPhone OS 3.0.

Important: The UIKit classes are generally not thread safe. All work involving your application's event-handling code should be performed on your application's main thread.

Events and Event Types

An **event** is an object that represents a user event—that is, a user action that iPhone OS detects, such as the touch of a finger or a shake of the device. In Cocoa Touch, events are instances of the `UIEvent` class. An event object may encapsulate state related to the user event, such as the associated touches. As a user event takes place—for example, as fingers touch the screen and move across its surface—the system continually sends event objects to an application for handling.

iPhone OS currently supports two types of events: touch events and motion events. The `UIEvent` class has been extended for iPhone OS 3.0 to accommodate not only these types of events but future kinds of events as well. It declares the `enum` constants shown in Listing 3-1.

Listing 3-1 Event-type and event-subtype constants

```
typedef enum {
    UIEventTypeTouches,
    UIEventTypeMotion,
} UIEventType;

typedef enum {
    UIEventSubtypeNone        = 0,
    UIEventSubtypeMotionShake = 1,
} UIEventSubtype;
```

Each event has one of these event type and subtype constants associated with it, which you can access through the `type` and `subtype` properties of `UIEvent`. The event type includes both touch events and motion events. In iPhone OS 3.0, there is only a shake-motion subtype (`UIEventSubtypeMotionShake`); touch events always have a subtype of `UIEventSubtypeNone`.

You should never retain a `UIEvent` object in your code. If you need to preserve the current state of an event object for later evaluation, you should copy and store those bits of state in an appropriate manner (using an instance variable or a dictionary object, for example).

Event Delivery

The delivery of an event to an object for handling occurs along a specific path. As described in "Core Application Architecture" (page 17), when users touch the screen of a device, iPhone OS recognizes the set of touches and packages them in a `UIEvent` object that it places in the active application's event queue. If the system interprets the shaking of the device as a motion event, an event object representing that event is also placed in the application's event queue. The singleton `UIApplication` object managing the application takes an event from the top of the queue and dispatches it for handling. Typically, it sends the event to the application's key window—the window currently the focus for user events—and the `UIWindow` object representing that window sends the event to an initial object for handling. That object is different for touch events and motion events.

- **Touch events**. The window object uses hit-testing and the responder chain to find the view to receive the touch event. In hit-testing, a window calls `hitTest:withEvent:` on the top-most view of the view hierarchy; this method proceeds by recursively calling `pointInside:withEvent:` on each view in the view hierarchy that returns `YES`, proceeding down the hierarchy until it finds the subview within whose bounds the touch took place. That view becomes the hit-test view.

  If the hit-test view cannot handle the event, the event travels up the responder chain as described in "Responder Objects and the Responder Chain" (page 78) until the system finds a view that can handle it. A touch object (described in "Touch Events" (page 81)) is associated with its hit-test view for its lifetime, even if the touch represented by the object subsequently moves outside the view. "Hit-Testing" (page 92) discusses some of the programmatic implications of hit-testing.
- **Motion events**. The window object sends the motion event to the first responder for handling. (The first responder is described in "Responder Objects and the Responder Chain."

Although the hit-test view and the first responder are often the same view object, they do not have to be the same.

The `UIApplication` object and each `UIWindow` object dispatches events in the `sendEvent:` method. (These classes declare a method with the same signature). Because these methods are funnel points for events coming into an application, you can subclass `UIApplication` or `UIWindow` and override the `sendEvent:` method to monitor events (which is something few applications would need to do). If you override these methods, be sure to call the superclass implementation (that is, `[super sendEvent:theEvent]`); never tamper with the distribution of events.

Responder Objects and the Responder Chain

The preceding discussion mentions the concept of responders. What is a responder object and how does it fit into the architecture for event delivery?

A **responder object** is an object that can respond to events and handle them. `UIResponder` is the base class for all responder objects, also known as, simply, responders. It defines the programmatic interface not only for event handling but for common responder behavior. `UIApplication`, `UIView`, and all UIKit classes that descend from `UIView` (including `UIWindow`) inherit directly or indirectly from `UIResponder`, and thus their instances are responder objects.

The **first responder** is the responder object in an application (usually a `UIView` object) that is designated to be the first recipient of events other than touch events. A `UIWindow` object sends the first responder these events in messages, giving it the first shot at handling them. To receive these messages, the responder object

must implement `canBecomeFirstResponder` to return `YES`; it must also receive a `becomeFirstResponder` message (which it can invoke on itself). The first responder is the first view in a window to receive the following type of events and messages:

- Motion events—via calls to the `UIResponder` motion-handling methods described in "Motion Events" (page 95)
- Action messages—sent when the user manipulates a control (such as a button or slider) and no target is specified for the action message
- Editing-menu messages—sent when users tap the commands of the editing menu (described in "Copy, Cut, and Paste Operations" (page 97))

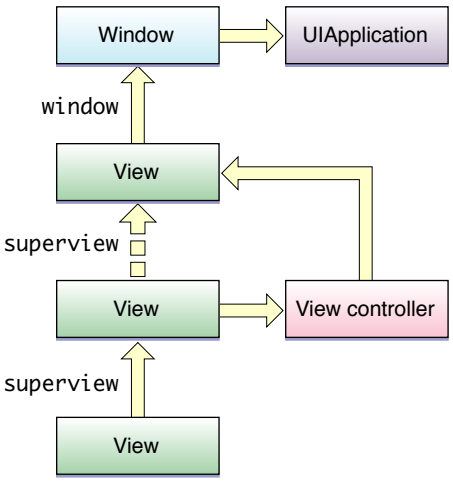
The first responder also plays a role in text editing. A text view or text field that is the focus of editing is made the first responder, which causes the virtual keyboard to appear.

Note: Applications must explicitly set a first responder to handle motion events, action messages, and editing-menu messages; UIKit automatically sets the text field or text view a user taps to be the first responder.

If the first responder or the hit-test view doesn't handle an event, it may pass the event (via message) to the **next responder** in the **responder chain** to see if it can handle it.

The responder chain is a linked series of responder objects along which an event or action message (or editing-menu message) is passed. It allows responder objects to transfer responsibility for handling an event to other, higher-level objects. An event proceeds up the responder chain as the application looks for an object capable of handling the event. Because the hit-test view is also a responder object, an application may also take advantage of the responder chain when handing touch events. The responder chain consists of a series of "next responders" in the sequence depicted in Figure 3-1.

Figure 3-1 The responder chain in iPhone OS



When the system delivers an event, it first sends it to a specific view. For touch events, that view is the one returned by `hitTest:withEvent:`; for motion events and action messages, that view is the first responder. If the initial view doesn't handle the event, it travels up the responder chain along a particular path:

1. The hit-test view or first responder passes the event or action message to its view controller if it has one; if the view doesn't have a view controller, it passes the event or action message to its superview.

2. If a view or its view controller cannot handle the event or action message, it passes it to the superview of the view.
3. Each subsequent superview in the hierarchy follows the pattern described in the first two steps if it cannot handle the event or action message.
4. The topmost view in the view hierarchy, if it doesn't handle the event or action message, passes it to the `UIWindow` object for handling.
5. The `UIWindow` object, if it doesn't handle the event or action message, passes it to the singleton `UIApplication` object.

If the application object cannot handle the event or action message, it discards it.

If you implement a custom view to handle events or action messages, you should not forward the event or message to `nextResponder` directly to send it up the responder chain. Instead invoke the superclass implementation of the current event-handling method—let UIKit handle the traversal of the responder chain.

Regulating Event Delivery

UIKit gives applications programmatic means to simplify event handling or to turn off the stream of events completely. The following list summarizes these approaches:

- **Turning off delivery of touch events**. By default, a view receives touch events, but you can set its `userInteractionEnabled` property to `NO` to turn off delivery of events. A view also does not receive events if it's hidden or if it's transparent.
- **Turning off delivery of touch events for a period**. An application can call the `UIApplication` method `beginIgnoringInteractionEvents` and later call the `endIgnoringInteractionEvents` method. The first method stops the application from receiving touch event messages entirely; the second method is called to resume the receipt of such messages. You sometimes want to turn off event delivery while your code is performing animations.
- **Turning on delivery of multiple touches**. By default, a view ignores all but the first touch during a multi-touch sequence. If you want the view to handle multiple touches you must enable this capability for the view. You do this programmatically by setting the `multipleTouchEnabled` property of your view to `YES`, or in Interface Builder by setting the related attribute in the inspector for the related view.
- **Restricting event delivery to a single view**. By default, a view's `exclusiveTouch` property is set to `NO`, which means that this view does not block other views in a window from receiving touches. If you set the property to `YES`, you mark the view so that, if it is tracking touches, it is the only view in the window that is tracking touches. Other views in the window cannot receive those touches. However, a view that is marked "exclusive touch" does not receive touches that are associated with other views in the same window. If a finger contacts an exclusive-touch view, then that touch is delivered only if that view is the only view tracking a finger in that window. If a finger touches a non-exclusive view, then that touch is delivered only if there is not another finger tracking in an exclusive-touch view.
- **Restricting event delivery to subviews**. A custom `UIView` class can override `hitTest:withEvent:` to restrict the delivery of multi-touch events to its subviews. See "Hit-Testing" (page 92) for a discussion of this technique.

Touch Events

Touch events in iPhone OS are based on a Multi-Touch model. Instead of using a mouse and a keyboard, users touch the screen of the device to manipulate objects, enter data, and otherwise convey their intentions. iPhone OS recognizes one or more fingers touching the screen as part of a **Multi-Touch sequence**. This sequence begins when the first finger touches down on the screen and ends when the last finger is lifted from the screen. iPhone OS tracks fingers touching the screen throughout a multi-touch sequence and records the characteristics of each of them, including the location of the finger on the screen and the time the touch occurred. Applications often recognize certain combinations of touches as gestures and respond to them in ways that are intuitive to users, such as zooming in on content in response to a pinching gesture and scrolling through content in response to a flicking gesture.

Note: A finger on the screen affords a much different level of precision than a mouse pointer. When a user touches the screen, the area of contact is actually elliptical and tends to be offset below the point where the user thinks he or she touched. This "contact patch" also varies in size and shape based on which finger is touching the screen, the size of the finger, the pressure of the finger on the screen, the orientation of the finger, and other factors. The underlying Multi-Touch system analyzes all of this information for you and computes a single touch point.

Many classes in UIKit handle multi-touch events in ways that are distinctive to objects of the class. This is especially true of subclasses of `UIControl`, such as `UIButton` and `UISlider`. Objects of these subclasses—known as control objects—are receptive to certain types of gestures, such as a tap or a drag in a certain direction; when properly configured, they send an action message to a target object when that gesture occurs. Other UIKit classes handle gestures in other contexts; for example, `UIScrollView` provides scrolling behavior for table views, text views, and other views with large content areas.

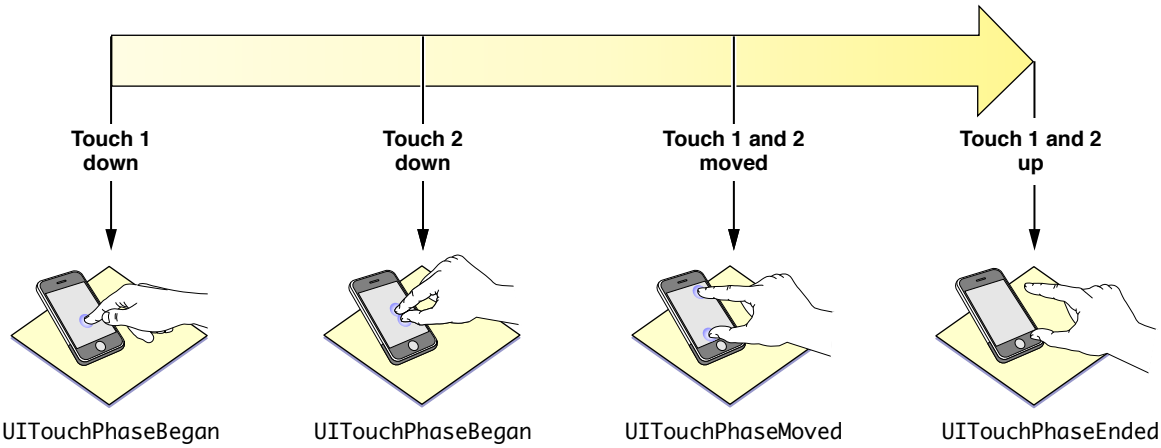
Some applications may not need to handle events directly; instead, they can rely on the classes of UIKit for that behavior. However, if you create a custom subclass of `UIView`—a common pattern in iPhone OS development—and if you want that view to respond to certain touch events, you need to implement the code required to handle those events. Moreover, if you want a UIKit object to respond to events differently, you have to create a subclass of that framework class and override the appropriate event-handling methods.

Events and Touches

In iPhone OS, a **touch** is the presence or movement of a finger on the screen that is part of a unique **multi-touch sequence**. For example, a pinch-close gesture has two touches: two fingers on the screen moving toward each other from opposite directions. There are simple single-finger gestures, such as a tap, or a double-tap, a drag, or a flick (where the user quickly swipes a finger across the screen). An application might recognize even more complicated gestures; for example, an application might have a custom control in the shape of a dial that users "turn" with multiple fingers to fine-tune some variable.

A `UIEvent` object of type `UIEventTypeTouches` represents a touch event. The system continually sends these touch-event objects (or simply, touch events) to an application as fingers touch the screen and move across its surface. The event provides a snapshot of all touches during a multi-touch sequence, most importantly the touches that are new or have changed for a particular view. As depicted in Figure 3-2, a multi-touch sequence begins when a finger first touches the screen. Other fingers may subsequently touch the screen, and all fingers may move across the screen. The sequence ends when the last of these fingers is lifted from the screen. An application receives event objects during each phase of any touch.

Figure 3-2 A multi-touch sequence and touch phases

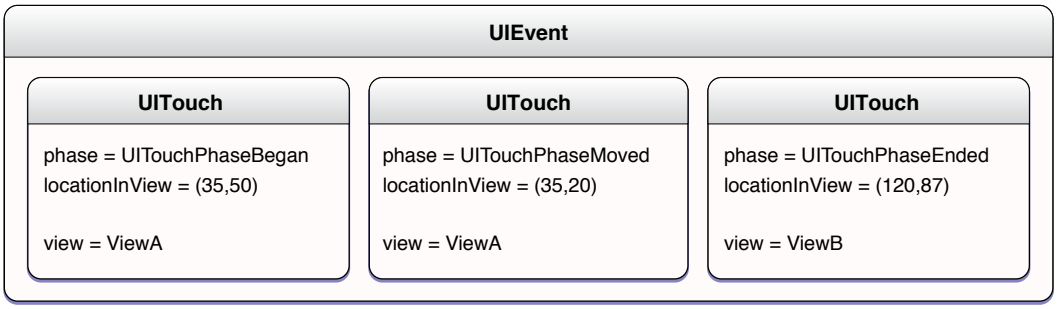


Touches, which are represented by `UITouch` objects, have both temporal and spatial aspects. The temporal aspect, called a phase, indicates when a touch has just begun, whether it is moving or stationary, and when it ends—that is, when the finger is lifted from the screen.

The spatial aspect of touches concerns their association with the object in which they occur as well as their location in it. When a finger touches the screen, the touch is associated with the underlying window and view and maintains that association throughout the life of the event. If multiple touches arrive at once, they are treated together only if they are associated with the same view. Likewise, if two touches arrive in quick succession, they are treated as a multiple tap only if they are associated with the same view. A touch object stores the current location and previous location (if any) of the touch in its view or window.

An event object contains all touch objects for the current multi-touch sequence and can provide touch objects specific to a view or window (see Figure 3-3). A touch object is persistent for a given finger during a sequence, and UIKit mutates it as it tracks the finger throughout it. The touch attributes that change are the phase of the touch, its location in a view, its previous location, and its timestamp. Event-handling code may evaluate these attributes to determine how to respond to a touch event.

Figure 3-3 Relationship of a `UIEvent` object and its `UITouch` objects



Because the system can cancel a multi-touch sequence at any time, an event-handling application must be prepared to respond appropriately. Cancellations can occur as a result of overriding system events, such as an incoming phone call.

Handling Multi-Touch Events

To handle multi-touch events, you must first create a subclass of a responder class. This subclass could be any one of the following:

- A custom view (subclass of `UIView`)
- A subclass of `UIViewController` or one of its UIKit subclasses
- A subclass of a UIKit view or control class, such as `UIImageView` or `UISlider`
- A subclass of `UIApplication` or `UIWindow` (although this would be rare)

A view controller typically receives, via the responder chain, touch events initially sent to its view.

For instances of your subclass to receive multi-touch events, your subclass must implement one or more the `UIResponder` methods for touch-event handling, described below. in addition, the view must be visible (neither transparent or hidden) and must have its `userInteractionEnabled` property set to `YES`, which is the default.

The following sections describe the touch-event handling methods, describe approaches for handling common gestures, show an example of a responder object that handles a complex sequence of multi-touch events, discuss event forwarding, and suggest some techniques for event handling.

The Event-Handling Methods

During a multi-touch sequence, the application dispatches a series of event messages to the target responder. To receive and handle these messages, the class of a responder object must implement at least one of the following methods declared by `UIResponder`, and, in some cases, all of these methods:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

The application sends these messages when there are new or changed touches for a given touch phase:

- It sends the `touchesBegan:withEvent:` message when one or more fingers touch down on the screen.
- It sends the `touchesMoved:withEvent:` message when one or more fingers move.
- It sends the `touchesEnded:withEvent:` message when one or more fingers lift up from the screen.
- It sends the `touchesCancelled:withEvent:` message when the touch sequence is cancelled by a system event, such as an incoming phone call.

Each of these methods is associated with a touch phase; for example, `touchesBegan:withEvent:` is associated with `UITouchPhaseBegan`. You can get the phase of any `UITouch` object by evaluating its `phase` property.

Each message that invokes an event-handling method passes in two parameters. The first is a set of `UITouch` objects that represent new or changed touches for the given phase. The second parameter is a `UIEvent` object representing this particular event. From the event object you can get *all* touch objects for the event or a subset of those touch objects filtered for specific views or windows. Some of these touch objects represent touches that have not changed since the previous event message or that have changed but are in different phases.

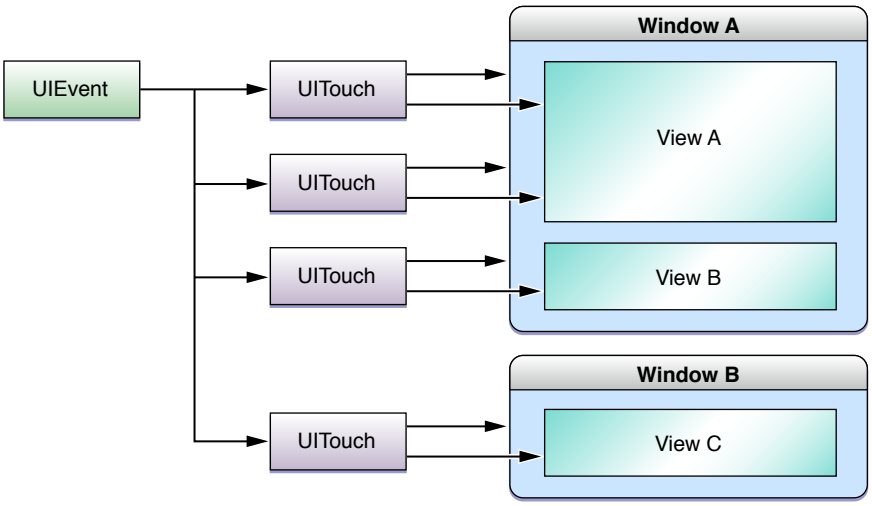
Basics of Touch-Event Handling

You frequently handle an event for a given phase by getting one or more of the `UITouch` objects in the passed-in set, evaluating their properties or getting their locations, and proceeding accordingly. The objects in the set represent those touches that are new or have changed for the phase represented by the implemented event-handling method. If any of the touch objects will do, you can send the `NSSet` object an `anyObject` message; this is the case when the view receives only the first touch in a multi-touch sequence (that is, the `multipleTouchEnabled` property is set to `NO`).

An important `UITouch` method is `locationInView:`, which, if passed a parameter of `self`, yields the location of the touch in the coordinate system of the receiving view. A parallel method tells you the previous location of the touch (`previousLocationInView:`. Properties of the `UITouch` instance tell you how many taps have been made (`tapCount`), when the touch was created or last mutated (`timestamp`), and what phase it is in (`phase`).

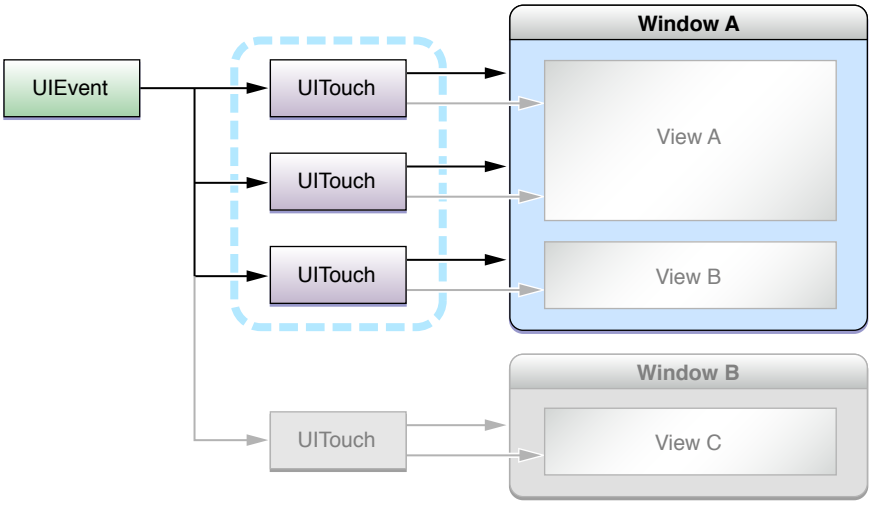
If for some reason you are interested in touches in the current multi-touch sequence that have not changed since the last phase or that are in a phase other than the ones in the passed-in set, you can request them from the passed-in `UIEvent` object. The diagram in Figure 3-4 depicts a `UIEvent` object that contains four touch objects. To get all these touch objects, you would invoke the `allTouches` on the event object.

Figure 3-4 All touches for a given touch event



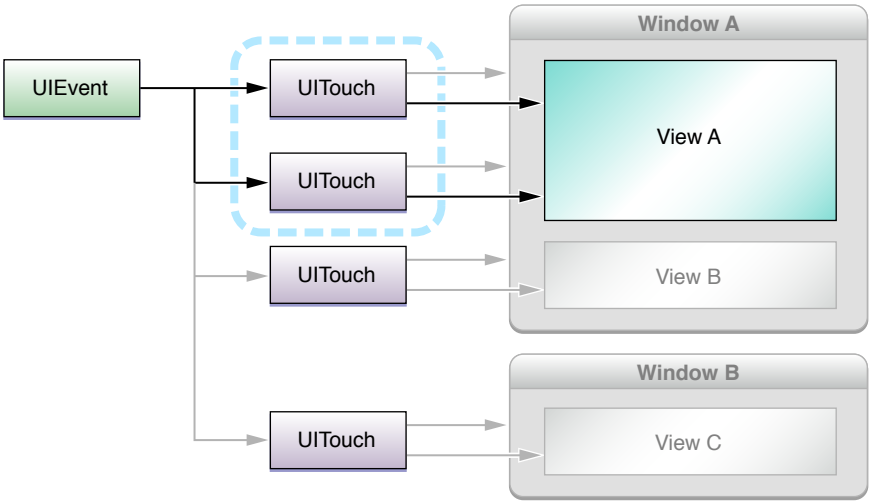
If on the other hand you are interested in only those touches associated with a specific window (Window A in Figure 3-5), you would send the `UIEvent` object a `touchesForWindow:` message.

Figure 3-5 All touches belonging to a specific window



If you want to get the touches associated with a specific view, you would call `touchesForView:` on the event object, passing in the view object (View A in Figure 3-6).

Figure 3-6 All touches belonging to a specific view



If a responder creates persistent objects while handling events during a multi-touch sequence, it should implement `touchesCancelled:withEvent:` to dispose of those objects when the system cancels the sequence. Cancellation often occurs when an external event—for example, an incoming phone call—disrupts the current application's event processing. Note that a responder object should also dispose of those same objects when it receives the last `touchesEnded:withEvent:` message for a multi-touch sequence. (See "Forwarding Touch Events" (page 93) to find out how to determine the last `UITouchPhaseEnded` touch object in a multi-touch sequence.)

Important: If your custom responder class is a subclass of `UIView` or `UIViewController`, you should implement all of the methods described in "The Event-Handling Methods" (page 83). If your class is a subclass of any other UIKit responder class, you do not need to override all of the event-handling methods; however, in those methods that you do override, be sure to call the superclass implementation of the method (for example, `super touchesBegan:touches withEvent:theEvent];`). The reason for this guideline is simple: All views that process touches, including your own, expect (or should expect) to receive a full touch-event stream. If you prevent a UIKit responder object from receiving touches for a certain phase of an event, the resulting behavior may be undefined and probably undesirable.

Handling Tap Gestures

A very common gesture in iPhone applications is the tap: the user taps an object on the screen with his or her finger. A responder object can handle a single tap in one way, a double-tap in another, and possibly a triple-tap in yet another way. To determine the number of times the user tapped a responder object, you get the value of the `tapCount` property of a `UITouch` object.

The best places to find this value are the methods `touchesBegan:withEvent:` and `touchesEnded:withEvent:`. In many cases, the latter method is preferred because it corresponds to the touch phase in which the user lifts a finger from a tap. By looking for the tap count in the touch-up phase (`UITouchPhaseEnded`), you ensure that the finger is really tapping and not, for instance, touching down and then dragging.

Listing 3-2 shows how to determine whether a double-tap occurred in one of your views.

Listing 3-2 Detecting a double-tap gesture

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {
        if (touch.tapCount >= 2) {
            [self.superview bringSubviewToFront:self];
        }
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

A complication arises when a responder object wants to handle a single-tap *and* a double-tap gesture in different ways. For example, a single tap might select the object and a double tap might display a view for editing the item that was double-tapped. How is the responder object to know that a single tap is not the first part of a double tap? Listing 3-3 (page 87) illustrates an implementation of the event-handling methods that increases the size of the receiving view upon a double-tap gesture and decreases it upon a single-tap gesture.

The following is a commentary on this code:

1. In `touchesEnded:withEvent:`, when the tap count is one, the responder object sends itself a `performSelector:withObject:afterDelay:` message. The selector identifies another method implemented by the responder to handle the single-tap gesture; the second parameter is an `NSValue` or `NSDictionary` object that holds some state of the `UITouch` object; the delay is some reasonable interval between a single- and a double-tap gesture.

   Note: Because a touch object is mutated as it proceeds through a multi-touch sequence, you cannot retain a touch and assume that its state remains the same. (And you cannot copy a touch object because `UITouch` does not adopt the `NSCopying` protocol.) Thus if you want to preserve the state of a touch object, you should store those bits of state in a `NSValue` object, a dictionary, or a similar object. (The code in Listing 3-3 stores the location of the touch in a dictionary but does not use it; this code is included for purposes of illustration.)

2. In `touchesBegan:withEvent:`, if the tap count is two, the responder object cancels the pending delayed-perform invocation by calling the `cancelPreviousPerformRequestsWithTarget:` method of `NSObject`, passing itself as the argument. If the tap count is not two, the method identified by the selector in the previous step for single-tap gestures is invoked after the delay.

3. In `touchesEnded:withEvent:`, if the tap count is two, the responder performs the actions necessary for handling double-tap gestures.

Listing 3-3 Handling a single-tap gesture and a double-tap gesture

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    if (aTouch.tapCount == 2) {
        [NSObject cancelPreviousPerformRequestsWithTarget:self];
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];
    if (theTouch.tapCount == 1) {
        NSDictionary *touchLoc = [NSDictionary dictionaryWithObject:
            [NSValue valueWithCGPoint:[theTouch locationInView:self]]
forKey:@"location"];
        [self performSelector:@selector(handleSingleTap:) withObject:touchLoc
afterDelay:0.3];
    } else if (theTouch.tapCount == 2) {
        // Double-tap: increase image size by 10%"
        CGRect myFrame = self.frame;
        myFrame.size.width += self.frame.size.width * 0.1;
        myFrame.size.height += self.frame.size.height * 0.1;
        myFrame.origin.x -= (self.frame.origin.x * 0.1) / 2.0;
        myFrame.origin.y -= (self.frame.origin.y * 0.1) / 2.0;
        [UIView beginAnimations:nil context:NULL];
        [self setFrame:myFrame];
        [UIView commitAnimations];
    }
}
```

```
- (void)handleSingleTap:(NSDictionary *)touches {
    // Single-tap: decrease image size by 10%"
    CGRect myFrame = self.frame;
    myFrame.size.width -= self.frame.size.width * 0.1;
    myFrame.size.height -= self.frame.size.height * 0.1;
    myFrame.origin.x += (self.frame.origin.x * 0.1) / 2.0;
    myFrame.origin.y += (self.frame.origin.y * 0.1) / 2.0;
    [UIView beginAnimations:nil context:NULL];
    [self setFrame:myFrame];
    [UIView commitAnimations];
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    /* no state to clean up, so null implementation */
}
```

Handling Swipe and Drag Gestures

Horizontal and vertical swipes are a simple type of gesture that you can track easily from your own code and use to perform actions. To detect a swipe gesture, you have to track the movement of the user's finger along the desired axis of motion, but it is up to you to determine what constitutes a swipe. In other words, you need to determine whether the user's finger moved far enough, if it moved in a straight enough line, and if it went fast enough. You do that by storing the initial touch location and comparing it to the location reported by subsequent touch-moved events.

Listing 3-4 shows some basic tracking methods you could use to detect horizontal swipes in a view. In this example, the view stores the initial location of the touch in a `startTouchPosition` instance variable. As the user's finger moves, the code compares the current touch location to the starting location to determine whether it is a swipe. If the touch moves too far vertically, it is not considered to be a swipe and is processed differently. If it continues along its horizontal trajectory, however, the code continues processing the event as if it were a swipe. The processing routines could then trigger an action once the swipe had progressed far enough horizontally to be considered a complete gesture. To detect swipe gestures in the vertical direction, you would use similar code but would swap the x and y components.

Listing 3-4 Tracking a swipe gesture in a view

```
#define HORIZ_SWIPE_DRAG_MIN  12
#define VERT_SWIPE_DRAG_MAX    4

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    // startTouchPosition is an instance variable
    startTouchPosition = [touch locationInView:self];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint currentTouchPosition = [touch locationInView:self];

    // To be a swipe, direction of touch must be horizontal and long enough.
    if (fabsf(startTouchPosition.x - currentTouchPosition.x) >=
HORIZ_SWIPE_DRAG_MIN &&
        fabsf(startTouchPosition.y - currentTouchPosition.y) <=
VERT_SWIPE_DRAG_MAX)
    {
        // It appears to be a swipe.
```

```
        if (startTouchPosition.x < currentTouchPosition.x)
            [self myProcessRightSwipe:touches withEvent:event];
        else
            [self myProcessLeftSwipe:touches withEvent:event];
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    startTouchPosition = 0.0;
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    startTouchPosition = 0.0;
}
```

Listing 3-5 shows an even simpler implementation of tracking a single touch, but this time for the purposes of dragging the receiving view around the screen. In this instance, the responder class fully implements only the `touchesMoved:withEvent:` method, and in this method computes a delta value between the touch's current location in the view and its previous location in the view. It then uses this delta value to reset the origin of the view's frame.

Listing 3-5 Dragging a view using a single touch

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    CGPoint loc = [aTouch locationInView:self];
    CGPoint prevloc = [aTouch previousLocationInView:self];

    CGRect myFrame = self.frame;
    float deltaX = loc.x - prevloc.x;
    float deltaY = loc.y - prevloc.y;
    myFrame.origin.x += deltaX;
    myFrame.origin.y += deltaY;
    [self setFrame:myFrame];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

Handling a Complex Multi-Touch Sequence

Taps, drags, and swipes are simple gestures, typically involving only a single touch. Handling a touch event consisting of two or more touches is a more complicated affair. You may have to track all touches through all phases, recording the touch attributes that have changed and altering internal state appropriately. There are a couple of things you should do when tracking and handling multiple touches:

- Set the `multipleTouchEnabled` property of the view to `YES`.
- Use a Core Foundation dictionary object (`CFDictionaryRef`) to track the mutations of touches through their phases during the event.

When handling an event with multiple touches, you often store initial bits of each touch's state for later comparison with the mutated `UITouch` instance. As an example, say you want to compare the final location of each touch with its original location. In the `touchesBegan:withEvent:` method, you can obtain the original location of each touch from the `locationInView:` method and store those in a `CFDictionaryRef` object using the addresses of the `UITouch` objects as keys. Then, in the `touchesEnded:withEvent:` method you can use the address of each passed-in `UITouch` object to obtain the object's original location and compare that with its current location. (You should use a `CFDictionaryRef` type rather than an `NSDictionary` object; the latter copies its keys, but the `UITouch` class does not adopt the `NSCopying` protocol, which is required for object copying.)

Listing 3-6 illustrates how you might store beginning locations of `UITouch` objects in a Core Foundation dictionary.

Listing 3-6 Storing the beginning locations of multiple touches

```
- (void)cacheBeginPointForTouches:(NSSet *)touches
{
    if ([touches count] > 0) {
        for (UITouch *touch in touches) {
            CGPoint *point = (CGPoint *)CFDictionaryGetValue(touchBeginPoints,
 touch);
            if (point == NULL) {
                point = (CGPoint *)malloc(sizeof(CGPoint));
                CFDictionarySetValue(touchBeginPoints, touch, point);
            }
            *point = [touch locationInView:view.superview];
        }
    }
}
```

Listing 3-7 illustrates how to retrieve those initial locations stored in the dictionary. It also gets the current locations of the same touches. It uses these values in computing an affine transformation (not shown).

Listing 3-7 Retrieving the initial locations of touch objects

```
- (CGAffineTransform)incrementalTransformWithTouches:(NSSet *)touches {
   NSArray *sortedTouches = [[touches allObjects]
sortedArrayUsingSelector:@selector(compareAddress:)];

   // other code here ...

    UITouch *touch1 = [sortedTouches objectAtIndex:0];
    UITouch *touch2 = [sortedTouches objectAtIndex:1];

    CGPoint beginPoint1 = *(CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch1);
    CGPoint currentPoint1 = [touch1 locationInView:view.superview];
    CGPoint beginPoint2 = *(CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch2);
    CGPoint currentPoint2 = [touch2 locationInView:view.superview];

    // compute the affine transform...
}
```

Although the code example in Listing 3-8 doesn't use a dictionary to track touch mutations, it also handles multiple touches during an event. It shows a custom `UIView` object responding to touches by animating the movement of a "Welcome" placard around the screen as a finger moves it and changing the language of the welcome when the user makes a double-tap gesture. (The code in this example comes from the *MoveMe* sample code project, which you can examine to get a better understanding of the event-handling context.)

Listing 3-8 Handling a complex multi-touch sequence

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    // Only move the placard view if the touch was in the placard view
    if ([touch view] != placardView) {
        // On double tap outside placard view, update placard's display string
        if ([touch tapCount] == 2) {
            [placardView setupNextDisplayString];
        }
        return;
    }
    // "Pulse" the placard view by scaling up then down
    // Use UIView's built-in animation
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.5];
    CGAffineTransform transform = CGAffineTransformMakeScale(1.2, 1.2);
    placardView.transform = transform;
    [UIView commitAnimations];

    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.5];
    transform = CGAffineTransformMakeScale(1.1, 1.1);
    placardView.transform = transform;
    [UIView commitAnimations];

    // Move the placardView to under the touch
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.25];
    placardView.center = [self convertPoint:[touch locationInView:self]
fromView:placardView];
    [UIView commitAnimations];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    // If the touch was in the placardView, move the placardView to its location
    if ([touch view] == placardView) {
        CGPoint location = [touch locationInView:self];
        location = [self convertPoint:location fromView:placardView];
        placardView.center = location;
        return;
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    // If the touch was in the placardView, bounce it back to the center
    if ([touch view] == placardView) {
        // Disable user interaction so subsequent touches don't interfere with
 animation
```

```
        self.userInteractionEnabled = NO;
        [self animatePlacardViewToCenter];
        return;
    }
}
```

Note: Custom views that redraw themselves in response to events they handle generally should only set drawing state in the event-handling methods and perform all of the drawing in the `drawRect:` method. To learn more about drawing view content, see "Graphics and Drawing" (page 107).

To find out when the last finger in a multi-touch sequence is lifted from a view, compare the number of `UITouch` objects in the passed-in set with the number of touches for the view maintained by the passed-in `UIEvent` object. If they are the same, then the multi-touch sequence has concluded. Listing 3-9 illustrates how to do this in code.

Listing 3-9 Determining when the last touch in a multi-touch sequence has ended

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    if ([touches count] == [[event touchesForView:self] count]) {
        // last finger has lifted....
    }
}
```

Remember that the passed-in set contains all touch objects associated with the receiving view that are new or changed for the given phase whereas the touch objects returned from `touchesForView:` includes *all* objects associated with the specified view.

Hit-Testing

Your custom responder can use hit-testing to find the subview or sublayer of itself that is "under" a touch, and then handle the event appropriately. It does this by either calling the `hitTest:withEvent:` method of `UIView` or the `hitTest:` method of `CALayer`; or it can override one of these methods. Responders sometimes perform hit-testing prior to event forwarding (see "Forwarding Touch Events" (page 93)).

Note: The `hitTest:withEvent:` and `hitTest:` methods have some slightly different behaviors.

If you have a custom view with subviews, you need to determine whether you want to handle touches at the subview level or the superview level. If the subviews do not handle touches by implementing `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, or `touchesMoved:withEvent:`, then these messages propagate up the responder chain to the superview. However, because multiple taps and multiple touches are associated with the subviews where they first occurred, the superview won't receive these touches. To ensure reception of all kinds of touches, the superview should override `hitTest:withEvent:` to return itself rather than any of its subviews.

The example in "Forwarding Touch Events" detects when an "Info" image in a layer of the custom view is tapped.

Listing 3-10 Calling `hitTest:` on a view's CALayer object

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    CGPoint location = [[touches anyObject] locationInView:self];
```

```
    CALayer *hitLayer = [[self layer] hitTest:[self convertPoint:location
fromView:nil]];

    if (hitLayer == infoImage) {
        [self displayInfo];
    }
}
```

In Listing 3-11, a responder subclass (in this case, a subclass of `UIWindow`) overrides `hitTest:withEvent:`. It first gets the hit-test view returned by the superclass. Then, if that view is itself, it substitutes the view that is furthest down the view hierarchy.

Listing 3-11 Overriding `hitTest:withEvent:`

```
- (UIView*)hitTest:(CGPoint)point withEvent:(UIEvent *)event {
    UIView *hitView = [super hitTest:point withEvent:event];

    if (hitView == self)
        return [[self subviews] lastObject];
    else
        return hitView;

}
```

Forwarding Touch Events

Event forwarding is a technique used by some applications. You forward touch events by invoking the event-handling methods of another responder object. Although this can be an effective technique, you should use it with caution. The classes of the UIKit framework are not designed to receive touches that are not bound to them; in programmatic terms, this means that the `view` property of the `UITouch` object must hold a reference to the framework object in order for the touch to be handled. If you want to conditionally forward touches to other responders in your application, all of these responders should be instances of your own subclasses of `UIView`.

For example, let's say an application has three custom views: A, B, and C. When the user touches view A, the application's window determines that it is the hit-test view and sends the initial touch event to it. Depending on certain conditions, view A forwards the event to either view B or view C. In this case, views A, B, and C must be aware that this forwarding is going on, and views B and C must be able to deal with touches that are not bound to them.

Event forwarding often requires analysis of touch objects to determine where they should be forwarded. There are several approaches you can take for this analysis:

- With an "overlay" view (such as a common superview), use hit-testing to intercept events for analysis prior to forwarding them to subviews (see "Hit-Testing" (page 92)).
- Override `sendEvent:` in a custom subclass of `UIWindow`, analyze touches, and forward them to the appropriate responders. In your implementation you should always invoke the superclass implementation of `sendEvent:`.
- Design your application so that touch analysis isn't necessary

Listing 3-12 illustrates the second technique, that of overriding `sendEvent:` in a subclass of `UIWindow`. In this example, the object to which touch events are forwarded is a custom "helper" responder that performs affine transformations on the view that is associated with.

Listing 3-12 Forwarding touch events to "helper" responder objects

```
- (void)sendEvent:(UIEvent *)event
{
    for (TransformGesture *gesture in transformGestures) {
        // collect all the touches we care about from the event
        NSSet *touches = [gesture observedTouchesForEvent:event];
        NSMutableSet *began = nil;
        NSMutableSet *moved = nil;
        NSMutableSet *ended = nil;
        NSMutableSet *cancelled = nil;

        // sort the touches by phase so we can handle them similarly to normal
 event dispatch
        for(UITouch *touch in touches) {
            switch ([touch phase]) {
                case UITouchPhaseBegan:
                    if (!began) began = [NSMutableSet set];
                    [began addObject:touch];
                    break;
                case UITouchPhaseMoved:
                    if (!moved) moved = [NSMutableSet set];
                    [moved addObject:touch];
                    break;
                case UITouchPhaseEnded:
                    if (!ended) ended = [NSMutableSet set];
                    [ended addObject:touch];
                    break;
                case UITouchPhaseCancelled:
                    if (!cancelled) cancelled = [NSMutableSet set];
                    [cancelled addObject:touch];
                    break;
                default:
                    break;
            }
        }
        // call our methods to handle the touches
        if (began)     [gesture touchesBegan:began withEvent:event];
        if (moved)     [gesture touchesMoved:moved withEvent:event];
        if (ended)     [gesture touchesEnded:ended withEvent:event];
        if (cancelled) [gesture touchesCancelled:cancelled withEvent:event];
    }
    [super sendEvent:event];
}
```

Notice that in this example that the overriding subclass does something important to the integrity of the touch-event stream: It invokes the superclass implementation of `sendEvent:`.

Note: This code in Listing 3-12 is taken from the *MultiTouchDemo* sample-code project.

Handling Events in Subclasses of UIKit Views and Controls

If you subclass a view or control class of the UIKit framework (for example, `UIImageView` or `UISwitch`) for the purpose of altering or extending event-handling behavior, you should keep the following points in mind:

- Unlike in a custom view, it is not necessary to override each event-handling method.

- Always invoke the superclass implementation of each event-handling method that you do override.
- Do not forward events to UIKit framework objects.

Motion Events

An iPhone or iPod touch device generates motion events when users move the device in a certain way, such as shaking it. Motion events have their origin in the device accelerometer. The system evaluates the accelerometer data and, if it meets certain criteria, interprets it as a gesture. It creates a `UIEvent` object representing this gesture and sends the event object to the currently active application for processing.

Note: Motion events were introduced in iPhone OS 3.0. In this version, only shaking motions are interpreted as gestures and become motion events.

Motion events are much simpler than touch events. The system only tells an application when the motion starts and when it stops, and not when each individual motion occurs. Moreover, whereas a touch event includes a set of touches and their related state, a motion event carries with it no state other than the event type, event subtype, and timestamp. The system interprets motion gestures in such a way as to not conflict with orientation changes.

To receive motion events, the responder object that is to handle them must be the first responder.

Listing 3-13 Becoming first responder

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

- (void)viewDidAppear:(BOOL)animated {
    [self becomeFirstResponder];
}
```

To handle motion events, a class inheriting from `UIResponder` must implement either the `motionBegan:withEvent:` method or `motionEnded:withEvent:` method, or possibly both of these methods (see "Event Handling Best Practices" (page 96)). For example, if an application wants to give horizontal shakes and vertical shakes different meanings, it could cache the current acceleration axis values in `motionBegan:withEvent:`, compare those cached values to the same axis values in `motionEnded:withEvent:`, and act on the results accordingly. A responder should also implement the `motionCancelled:withEvent:` method to respond to events that the system sends to cancel a motion event; these events sometimes reflect the system's determination that the motion is not a valid gesture after all.

Listing 3-14 shows code that handles a shaking-motion event by resetting views that have have been altered (by translation, rotation, and scaling) to their original positions, orientations, and sizes.

Listing 3-14 Handling a motion event

```
- (void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
}
```

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:0.5];
    self.view.transform = CGAffineTransformIdentity;

    for (UIView *subview in self.view.subviews) {
        subview.transform = CGAffineTransformIdentity;
    }
    [UIView commitAnimations];

    for (TransformGesture *gesture in [window allTransformGestures]) {
        [gesture resetTransform];
    }
}

- (void)motionCancelled:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
}
```

An application and its key window deliver a motion event to the window's first responder for handling. If the first responder doesn't handle it, the event progresses up the responder chain in a way similar to touch events until it is either handled or ignored. (See "Event Delivery" (page 78) for details.) However, there is one important difference between touch events and shake-motion events. When the user starts shaking the device, the system sends a motion event to the first responder in a `motionBegan:withEvent:` message; if the first responder doesn't handle the event, it travels up the responder chain. If the shaking lasts less than a second or so, the system sends a `motionEnded:withEvent:` message to the first responder. But if the shaking lasts longer, or if the system determines the motion is not a shake, the first responder receives a `motionCancelled:withEvent:` message.

If a shake-motion event travels up the responder chain to the window without being handled, and the `applicationSupportsShakeToEdit` property of `UIApplication` is set to `YES`, iPhone OS displays a sheet with Undo and Redo commands. By default, this property is set to `YES`.

Event Handling Best Practices

When handling events, both touch events and motion events, there are a few recommended techniques and patterns you should follow.

- Always implement the event-cancellation methods.

  In your implementation, you should restore the state of the view to what it was before the current multi-touch sequence, freeing any transient resources set up for handling the event. If you don't implement the cancellation method your view could be left in an inconsistent state. In some cases, another view might receive the cancellation message.

- If you handle events in a subclass of `UIView`, `UIViewController`, or (in rare cases) `UIResponder`,
  - You should implement all of the event-handling methods (even if it is a null implementation).
  - Do not call the superclass implementation of the methods.

- If you handle events in a subclass of any other UIKit responder class,
  - You do not have to implement all of the event-handling methods.

- ❏ But in the methods you do implement, be sure to call the superclass implementation. For example,

```
[super touchesBegan:theTouches withEvent:theEvent];
```

- Do not forward events to other responder objects of the UIKit framework.

  The responders that you forward events to should be instances of your own subclasses of `UIView`, and all of these objects must be aware that event-forwarding is taking place and that, the case of touch events, they may receive touches that are not bound to them.
- Custom views that redraw themselves in response to events should only set drawing state in the event-handling methods and perform all of the drawing in the `drawRect:` method.
- Do not explicitly send events up the responder (via `nextResponder`); instead, invoke the superclass implementation and let the UIKit handle responder-chain traversal.

Copy, Cut, and Paste Operations

Beginning with iPhone OS 3.0, users can now copy text, images, or other data in one application and paste that data to another location within the same application or in a different application. You can, for example, copy a person's address in an email message and paste it into the appropriate field in the Contacts application. The UIKit framework currently implements copy-cut-paste in the `UITextView`, `UITextField`, and `UIWebView` classes. If you want this behavior in your own applications, you can either use objects of these classes or implement copy-cut-paste yourself.

The following sections describe the programmatic interfaces of the UIKit that you use for copy, cut, and paste operations and explain how they are used.

Note: For usage guidelines related to copy and paste operations, see "Supporting Copy and Paste" in *iPhone Human Interface Guidelines*.

UIKit Facilities for Copy-Paste Operations

Several classes and an informal protocol of the UIKit framework give you the methods and mechanisms you need to implement copy, cut, and paste operations in your application:

- The `UIPasteboard` class provides pasteboards: protected areas for sharing data within an application or between applications. The class offers methods for writing and reading items of data to and from a pasteboard.
- The `UIMenuController` class displays an editing menu above or below the selection to be copied, cut, or pasted into. The commands of the editing menu are (potentially) Copy, Cut, Paste, Select, and Select All.
- The `UIResponder` class declares the method `canPerformAction:withSender:`. Responder classes can implement this method to show and remove commands of the editing menu based on the current context.

- The `UIResponderStandardEditActions` informal protocol declares the interface for handling copy, cut, paste, select, and select-all commands. When users tap one of the commands in the editing menu, the corresponding `UIResponderStandardEditActions` method is invoked.

Pasteboard Concepts

A pasteboard is a standardized mechanism for exchanging data within applications or between applications. The most familiar use for pasteboards is handling copy, cut, and paste operations:

- When a user selects data in an application and chooses the Copy (or Cut) menu command, the selected data is placed onto a pasteboard.
- When the user chooses the Paste menu command (either in the same or a different application), the data on a pasteboard is copied to the current application from the pasteboard.

In iPhone OS, a pasteboard is also used to support Find operations. Additionally, you may use pasteboards to transfer data between applications using custom URL schemes instead of copy, cut, and paste commands; see "Communicating with Other Applications" (page 34) for information about this technique.

Regardless of the operation, the basic tasks you perform with a pasteboard object are to write data to a pasteboard and to read data from a pasteboard. Although these tasks are conceptually simple, they mask a number of important details. The main complexity is that there may be a number of ways to represent data, and this complexity leads to considerations of efficiency. These and other issues are discussed in the following sections.

Named Pasteboards

Pasteboards may be public or private. Public pasteboard are called **system pasteboards**; private pasteboards are created by applications, and hence are called **application pasteboards**. Pasteboards must have unique names. `UIPasteboard` defines two system pasteboards, each with its own name and purpose:

- `UIPasteboardNameGeneral` is for cut, copy, and paste operations involving a wide range of data types. You can obtain a singleton object representing the General pasteboard by invoking the `generalPasteboard` class method.
- `UIPasteboardNameFind` is for search operations. The string currently typed by the user in the search bar (`UISearchBar`) is written to this pasteboard, and thus can be shared between applications. You can obtain an object representing the Find pasteboard by calling the `pasteboardWithName:create:` class method, passing in `UIPasteboardNameFind` for the name.

Typically you use one of the system-defined pasteboards, but if necessary you can create your own application pasteboard using `pasteboardWithName:create:` If you invoke `pasteboardWithUniqueName`, `UIPasteboard` gives you a uniquely-named application pasteboard. You can discover the name of a pasteboard through its `name` property.

Pasteboard Persistence

Pasteboards can be marked persistent so that they continue to exist beyond the termination of applications that use them. Pasteboards that aren't persistent are removed when the application that created them quits. System pasteboards are persistent. Application pasteboards by default are not persistent. To make an application pasteboard persistent, set its `persistent` property to `YES`. A persistent application pasteboard is removed when a user uninstalls the owning application.

Pasteboard Owner and Items

The object that last put data onto the pasteboard is referred to as the pasteboard **owner**. Each piece of data placed onto a pasteboard is considered a pasteboard **item**. The pasteboard can hold single or multiple items. Applications can place or retrieve as many items as they wish. For example, say a user selection in a view contains both text and an image. The pasteboard lets you copy the text and the image to the pasteboard as separate items. An application reading multiple items from a pasteboard can choose to take only those items that is supports (the text, but not the image, for example).

Important: When an application writes data to a pasteboard, even if it is just a single item, that data replaces the current contents of the pasteboard. Although you may use the `addItems:` method of `UIPasteboard` to append items, the write methods of the class do not append items to the current contents of the pasteboard.

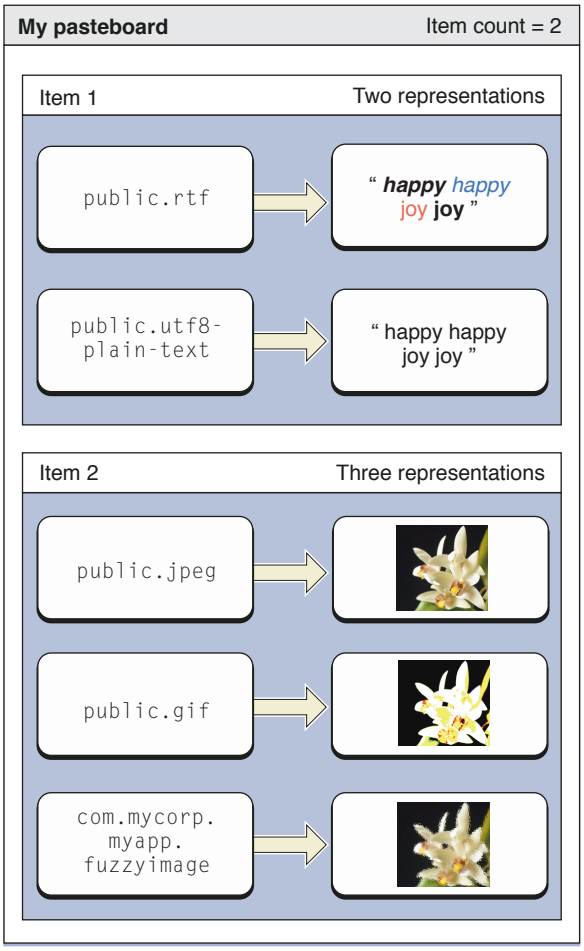
Representations and UTIs

Pasteboard operations are often carried out between two different applications. Neither application is required to know about the other, including the kinds of data it can handle. To maximize the potential for sharing, a pasteboard can hold multiple **representations** of the same pasteboard item. For example, a rich text editor might provide HTML, PDF, and plain-text representations of the copied data. An item on a pasteboard includes all representations of that data item that the application can provide.

Each representation of a pasteboard item is typically identified by a Unique Type Identifier (UTI). (A UTI is simply a string that uniquely identifies a particular data type. The UTI provides a common means to identify data types. If you have a custom data type you wish to support, you must create a unique identifier for it. For this, you could use reverse-DNS notation for your representation-type string to ensure uniqueness; for example, a custom representation type could be `com.myCompany.myApp.myType`. For more information on UTIs, see *Uniform Type Identifiers Overview*.

For example, suppose an application supported selection of rich text and images. It may want to place on a pasteboard both rich text and Unicode versions of a text selection and different representations of an image selection. Each representation of each item is stored with its own data, as shown in Figure 3-7.

Figure 3-7 Pasteboard items and representations



In general, to maximize the potential for sharing, pasteboard items should include as many different representations as possible.

A pasteboard reader must find the data type that best suits its capabilities (if any). Typically, this means selecting the richest type available. For example, a text editor might provide HTML (rich text) and plain-text representations of copied text data. An application that supports rich text should retrieve the HTML representation and an application that only supports plain text should retrieve the plain-text version.

Change Count

The change count is a per-pasteboard variable that increments every time the contents of the pasteboard changes—specifically, when items are added, modified, or removed. By examining the change count (through the `changeCount` property), an application can determine whether the current data in the pasteboard is the same as the data it last received. Every time the change count is incremented, the pasteboard sends a notification to interested observers.

Selection and Menu Management

To copy or cut something in a view, that "something" must be selected. It can be a range of text, an image, a URL, a color, or any other representation of data, including custom objects. To implement copy-and-paste behavior in your custom view, you must manage the selection of objects in that view yourself. If the user selects an object in the view by making a certain touch gesture (for example, a double-tap) you must handle that event, internally record the selection (and deselect any previous selection), and perhaps visually indicate the new selection in the view. If it is possible for users to select multiple objects in your view for copy-cut-paste operations, you must implement that multiple-selection behavior.

Note: Touch events and techniques for handling them are discussed in "Touch Events" (page 81).

When your application determines that the user has requested the editing menu—which could be the action of making a selection—you should complete the following steps to display the menu:

1. Call the `sharedMenuController` class method of `UIMenuController` to get the global menu-controller instance.
2. Compute the boundaries of the selection and with the resulting rectangle call the `setTargetRect:inView:` method. The editing menu is displayed above or below this rectangle, depending how close the selection is to the top or bottom of the screen.
3. Call the `setMenuVisible:animated:` method (with `YES` for both arguments) to animate the display of the editing menu above or below the selection.

Listing 3-15 illustrates how you might display the edit menu in an implementation of the `touchesEnded:withEvent:` method. (Note that the example omits the section of code that handles the selection.) This code snippet also shows that the custom view sending itself a `becomeFirstResponder` message to ensure that it is the first responder for the subsequent copy, cut, and paste operations.

Listing 3-15 Displaying the editing menu

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];

    if ([theTouch tapCount] == 2  && [self becomeFirstResponder]) {

        // selection management code goes here...

        // bring up editing menu.
        UIMenuController *theMenu = [UIMenuController sharedMenuController];
       CGRect selectionRect = CGRectMake(currentSelection.x, currentSelection.y,
 SIDE, SIDE);
        [theMenu setTargetRect:selectionRect inView:self];
        [theMenu setMenuVisible:YES animated:YES];

    }
}
```

The menu initially includes all commands for which the first responder has corresponding `UIResponderStandardEditActions` method implementations (`copy:`, `paste:`, and so on). Before the menu is displayed, however, the system sends a `canPerformAction:withSender:` message to the first responder, which in many cases is the custom view itself. In its implementation of this method, the responder

evaluates whether the command (indicated by the selector in the first argument) is applicable in the current context. For example, if the selector is `paste:` and there is no data in the pasteboard of a type the view can handle, the responder should return `NO` to suppress the Paste command. If the first responder does not implement the `canPerformAction:withSender:` method, or does not handle the given command, the message travels up the responder chain.

Listing 3-16 shows an implementation of the `canPerformAction:withSender:` method that looks for message matching the `copy:`, `copy:`, and `paste:` selectors; it enables or disables the Copy, Cut, and Paste menu commands based on the current selection context and, for paste, the contents of the pasteboard.

Listing 3-16 Conditionally enabling menu commands

```
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender {
    BOOL retValue = NO;
    ColorTile *theTile = [self colorTileForOrigin:currentSelection];

    if (action == @selector(paste:) )
        retValue = (theTile == nil) &&
             [[UIPasteboard generalPasteboard] containsPasteboardTypes:
             [NSArray arrayWithObject:ColorTileUTI]];
    else if ( action == @selector(cut:) || action == @selector(copy:) )
        retValue = (theTile != nil);
    else
        retValue = [super canPerformAction:action withSender:sender];
    return retValue;
}
```

Note that the final `else` clause in this method calls the superclass implementation to give any superclasses a chance to handle commands that the subclass chooses to ignore.

Note that a menu command, when acted upon, can change the context for other menu commands. For example, if the user selects all objects in the view, the Copy and Cut commands should be included in the menu. In this case the responder can, while the menu is still visible, call `update` on the menu controller; this results in the reinvocation of `canPerformAction:withSender:` on the first responder.

Copying and Cutting the Selection

When users tap the Copy or Cut command of the editing menu, the system invokes the `copy:` or `cut:` method (respectively) of the responder object that implements it. Usually the first responder—your custom view—implements these methods, but if the first responder doesn't implement them, the message travels up the responder in the usual fashion. Note that the `UIResponderStandardEditActions` informal protocol declares these methods.

Note: Because `UIResponderStandardEditActions` is an informal protocol, any class in your application can implement its methods. But to take advantage of the default behavior for traversing the responder chain, the class implementing the methods should inherit from `UIResponder` and should be installed in the responder chain.

In response to a `copy:` or `cut:` message, you write the object or data represented by the selection to the pasteboard in as many different representations as you can. This operation involves the following steps (which assume a single pasteboard item):

1. From the selection, identify or obtain the object or the binary data corresponding to the object.

   Binary data must be encapsulated in an `NSData` object. If you're going to write another type of object to the pasteboard, it must be a property-list object—that is, an object of one of the following classes: `NSString`, `NSArray`, `NSDictionary`, `NSDate`, `NSNumber`, or `NSURL`. (For more on property-list objects, see *Property List Programming Guide*.)

2. If possible, generate one or more other representations of the object or data.

   For example, if in the previous step you created a `UIImage` object representing a selected image, you could use the `UIImageJPEGRepresentation` and `UIImagePNGRepresentation` functions to convert the image to a different representation.

3. Obtain a pasteboard object.

   In many cases, this is the general pasteboard, which you can get through the `generalPasteboard` class method.

4. Assign a suitable UTI for each representation of data written to the pasteboard item.

   See "Pasteboard Concepts" (page 98) for a discussion of this subject.

5. Write the data to the first pasteboard item for each representation type:

   - To write a data object, send a `setData:forPasteboardType:` message to the pasteboard object.
   - To write a property-list object, send a `setValue:forPasteboardType:` message to the pasteboard object.

6. If the command is Cut (`cut:` method), remove the object represented by the selection from the application's data model and update your view.

Listing 3-17 shows implementations of the `copy:` and `cut:` methods. The `cut:` method invokes the `copy:` method and then removes the selected object from the view and the data model. Note that the `cut:` method archives a custom object to obtain an `NSData` object that it can pass to the pasteboard in `setData:forPasteboardType:`.

Listing 3-17 Copying and cutting operations

```
- (void)copy:(id)sender {
    UIPasteboard *gpBoard = [UIPasteboard generalPasteboard];
    ColorTile *theTile = [self colorTileForOrigin:currentSelection];
    if (theTile) {
       NSData *tileData = [NSKeyedArchiver archivedDataWithRootObject:theTile];
        if (tileData)
            [gpBoard setData:tileData forPasteboardType:ColorTileUTI];
    }
}

- (void)cut:(id)sender {
     [self copy:sender];
     ColorTile *theTile = [self colorTileForOrigin:currentSelection];

     if (theTile) {
         CGPoint tilePoint = theTile.tileOrigin;
         [tiles removeObject:theTile];
```

```
            CGRect tileRect = [self rectFromOrigin:tilePoint inset:TILE_INSET];
            [self setNeedsDisplayInRect:tileRect];
        }
    }
```

Pasting the Selection

When users tap the Paste command of the editing menu, the system invokes the `paste:` method of the responder object that implements it. Usually the first responder—your custom view—implements this method, but if the first responder doesn't implement it, the message travel up the responder in the usual fashion. The `paste:` method is declared by the `UIResponderStandardEditActions` informal protocol.

In response to a `paste:` message, you read an object from the pasteboard in a representation that your application supports. Then you add the pasted object to the application's data model and display the new object in the view in the user-indicated location. This operation involves the following steps (which assume a single pasteboard item):

1. Obtain a pasteboard object.

   In many cases, this is the general pasteboard, which you can get through the `generalPasteboard` class method.

2. Verify that the first pasteboard item contains data in a representation that your application can handle by calling the `containsPasteboardTypes:` method or the `pasteboardTypes` method and then examining the returned array of types.

   Note that you should have already performed this step in your implementation of `canPerformAction:withSender:`.

3. If the first item of the pasteboard contains data that the application can handle, call one of the following methods to read it:

   - `dataForPasteboardType:` if the data to be read is encapsulated in an `NSData` object.
   - `valueForPasteboardType:` if the data to be read is encapsulated in a property-list object (see "Copying and Cutting the Selection" (page 102)).

4. Add the object to the application's data model.

5. Display a representation of the object in the user interface at the location specified by the user.

Listing 3-18 is an example of an implementation of the `paste:` method. It does the reverse of the combined `cut:` and `copy:` methods. The custom view first sees whether the general pasteboard holds its custom representation of data; if it does, it then reads the data from the pasteboard, adds it to the application's data model, and marks part of itself—the current selection—for redrawing.

Listing 3-18 Pasting data from the pasteboard to a selection

```
- (void)paste:(id)sender {
    UIPasteboard *gpBoard = [UIPasteboard generalPasteboard];
    NSArray *pbType = [NSArray arrayWithObject:ColorTileUTI];
    ColorTile *theTile = [self colorTileForOrigin:currentSelection];
    if (theTile == nil && [gpBoard containsPasteboardTypes:pbType]) {
```

```
        NSData *tileData = [gpBoard dataForPasteboardType:ColorTileUTI];
        ColorTile *theTile = (ColorTile *)[NSKeyedUnarchiver
unarchiveObjectWithData:tileData];
         if (theTile) {
             theTile.tileOrigin = self.currentSelection;
             [tiles addObject:theTile];
             CGRect tileRect = [self rectFromOrigin:currentSelection
inset:TILE_INSET];
             [self setNeedsDisplayInRect:tileRect];
         }
     }
}
```

Dismissing the Editing Menu

When your implementation of the `cut:`, `copy:` or `paste:` command returns, the editing menu is automatically hidden. You can keep it visible with the following line of code:

```
[UIMenuController setMenuController].menuVisible = YES;
```

The system may hide the editing menu at any time. For example, it hides the menu when an alert is displayed or the user taps in another area of the screen. If you have state or a display that depends on whether the editing menu is visible, you should listen for the notification named `UIMenuControllerWillHideMenuNotification` and take an appropriate action.

Graphics and Drawing

High-quality graphics are an important part of your application's user interface. Providing high-quality graphics not only makes your application look good, but it also makes your application look like a natural extension to the rest of the system. iPhone OS provides two primary paths for creating high-quality graphics in your system: OpenGL or native rendering using Quartz, Core Animation, and UIKit.

The OpenGL frameworks are geared primarily toward game development or applications that require high frame rates. OpenGL is a C-based interface used to create 2D and 3D content on desktop computers. iPhone OS supports OpenGL drawing through the OpenGL ES framework, which provides support for both the OpenGL ES 2.0 and OpenGL ES v1.1 specifications. OpenGL ES is designed specifically for use on embedded hardware systems and differs in many ways from desktop versions of OpenGL.

For developers who want a more object-oriented drawing approach, iPhone OS provides Quartz, Core Animation, and the graphics support in UIKit. Quartz is the main drawing interface, providing support for path-based drawing, anti-aliased rendering, gradient fill patterns, images, colors, coordinate-space transformations, and PDF document creation, display, and parsing. UIKit provides Objective-C wrappers for Quartz images and color manipulations. Core Animation provides the underlying support for animating changes in many UIKit view properties and can also be used to implement custom animations.

This chapter provides an overview of the drawing process for iPhone applications, along with specific drawing techniques for each of the supported drawing technologies. This chapter also provides tips and guidance on how to optimize your drawing code for the iPhone OS platform.

Important: The UIKit classes are generally not thread safe. All drawing-related operations should be performed on your application's main thread.

The UIKit Graphics System

In iPhone OS, all drawing—regardless of whether it involves OpenGL, Quartz, UIKit, or Core Animation—occurs within the confines of a `UIView` object. Views define the portion of the screen in which drawing occurs. If you use system-provided views, this drawing is handled for you automatically. If you define custom views, however, you must provide the drawing code yourself. For applications that draw using OpenGL, once you set up your rendering surface, you use the drawing model specified by OpenGL.

For Quartz, Core Animation, and UIKit, you use the drawing concepts described in the following sections.

The View Drawing Cycle

The basic drawing model for `UIView` objects involves updating content on demand. The `UIView` class makes the update process easier and more efficient, however, by gathering the update requests you make and delivering them to your drawing code at the most appropriate time.

Whenever a portion of your view needs to be redrawn, the `UIView` object's built-in drawing code calls its `drawRect:` method. It passes this method a rectangle indicating the portion of your view that needs to be redrawn. You override this method in your custom view subclasses and use it to draw the contents of your view. The first time your view is drawn, the rectangle passed to the `drawRect:` method contains your view's entire visible area. During subsequent calls, however, this rectangle represents only the portion of the view that actually needs to be redrawn. There are several actions that can trigger a view update:

- Moving or removing another view that was partially obscuring your view
- Making a previously hidden view visible again by setting its `hidden` property to `NO`
- Scrolling a view off the screen and then back on
- Explicitly calling the `setNeedsDisplay` or `setNeedsDisplayInRect:` method of your view

After calling your `drawRect:` method, the view marks itself as updated and waits for new actions to arrive and trigger another update cycle. If your view displays static content, then all you need to do is respond to changes in your view's visibility caused by scrolling and the presence of other views. If you update your view's content periodically, however, you must determine when to call the `setNeedsDisplay` or `setNeedsDisplayInRect:` method to trigger an update. For example, if you were updating content several times a second, you might want to set up a timer to update your view. You might also update your view in response to user interactions or the creation of new content in your view.

Coordinates and Coordinate Transforms

As described in "View Coordinate Systems" (page 57), the origin of a window or view is located in its top-left corner, and positive coordinate values extend down and to the right of this origin. When you write your drawing code, you use this coordinate system to specify the location of individual points for the content you draw.

If you need to make changes to the default coordinate system, you do so by modifying the current transformation matrix. The **current transformation matrix (CTM)** is a mathematical matrix that maps points in your view's coordinate system to points on the device's screen. When your view's `drawRect:` method is first called, the CTM is configured so that the origin of the coordinate system matches the your view's origin and its positive axes extend down and to the right. However, you can change the CTM by adding scaling, rotation, and translation factors to it and thereby change the size, orientation, and position of the default coordinate system relative to the underlying view or window.

Modifying the CTM is the standard technique used to draw content in your view because it involves much less work. If you want to draw a 10 x 10 square starting at the point (20, 20) in the current drawing system, you could create a path that moves to (20, 20) and then draws the needed set of lines to complete the square. If you decide later that you want to move that square to the point (10, 10), however, you would have to recreate the path with the new starting point. In fact, you would have to recreate the path every time you changed the origin. Creating paths is a relatively expensive operation, but creating a square whose origin is at (0, 0) and modifying the CTM to match the desired drawing origin is cheap by comparison.

In the Core Graphics framework, there are two ways to modify the CTM. You can modify the CTM directly using the CTM manipulation functions defined in *CGContext Reference*. You can also create a `CGAffineTransform` structure, apply any transformations you want, and then concatenate that transform onto the CTM. Using an affine transform lets you group transformations and then apply them to the CTM all at once. You can also evaluate and invert affine transforms and use them to modify point, size, and rectangle values in your code. For more information on using affine transforms, see *Quartz 2D Programming Guide* and *CGAffineTransform Reference*.

Graphics Contexts

Before calling your custom `drawRect:` method, the view object automatically configures its drawing environment so that your code can start drawing immediately. As part of this configuration, the `UIView` object creates a graphics context (a `CGContextRef` opaque type) for the current drawing environment. This graphics context contains the information the drawing system needs to perform any subsequent drawing commands. It defines basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and several others.

You can create custom graphics context objects in situations where you want to draw somewhere other than your view. In Quartz, you primarily do this when you want to capture a series of drawing commands and use them to create an image or a PDF file. To create the context, you use the `CGBitmapContextCreate` or `CGPDFContextCreate` function. Once you have the context, you can pass it to the drawing functions needed to create your content.

When creating custom contexts, the coordinate system for those contexts is different than the native coordinate system used by iPhone OS. Instead of the origin being in the upper-left corner of the drawing surface, it is in the lower-left corner and the axes point up and to the right. The coordinates you specify in your drawing commands must take this into consideration or the resulting image or PDF file may appear wrong when rendered.

Important: Because you use a lower-left origin when drawing into a bitmap or PDF context, you must compensate for that coordinate system when rendering the resulting content into a view. In other words, if you create an image and draw it using the `CGContextDrawImage` function, the image will appear upside down by default. To correct for this, you must invert the y axis of the CTM (by multiplying it by `-1`) and shift the origin from the lower-left corner to the upper-left corner of the view.

If you use a `UIImage` object to wrap a `CGImageRef` you create, you do not need to modify the CTM. The `UIImage` object automatically compensates for the inverted coordinate system of the `CGImageRef` type.

For more information about graphics contexts, modifying the graphics state information, and using graphics contexts to create custom content, see *Quartz 2D Programming Guide*. For a list of functions used in conjunction with graphics contexts, see *CGContext Reference*, *CGBitmapContext Reference*, and *CGPDFContext Reference*.

Points Versus Pixels

The Quartz drawing system uses a vector-based drawing model. Compared to a raster-based drawing model, in which drawing commands operate on individual pixels, drawing commands in Quartz are specified using a fixed-scale drawing space, known as the **user coordinate space**. iPhone OS then maps the coordinates in this drawing space onto the actual pixels of the device. The advantage of this model is that graphics drawn using vector commands continue to look good when scaled up or down using an affine transform.

In order to maintain the precision inherent in a vector-based drawing system, drawing coordinates are specified using floating-point values instead of integers. The use of floating-point values for coordinates makes it possible for you to specify the location of your program's content very precisely. For the most part, you do not have to worry about how those values are eventually mapped to the device's screen.

The user coordinate space is the environment that you use for all of your drawing commands. The units of this space are measured in points. The **device coordinate space** refers to the native coordinate space of the device, which is measured in pixels. By default, one point in user coordinate space is equal to one pixel in device space, which results in 1 point equaling 1/160th of an inch. You should not assume that this 1-to-1 ratio will always be the case, however.

Color and Color Spaces

iPhone OS supports the full range of color spaces available in Quartz; however, most applications should need only the RGB color space. Because iPhone OS is designed to run on embedded hardware and display graphics on a screen, the RGB color space is the most appropriate one to use.

The `UIColor` object provides convenience methods for specifying color values using RGB, HSB, and grayscale values. When creating colors in this way, you never need to specify the color space. It is determined for you automatically by the `UIColor` object.

You can also use the `CGContextSetRGBStrokeColor` and `CGContextSetRGBFillColor` functions in the Core Graphics framework to create and set colors. Although the Core Graphics framework includes support for creating colors using other color spaces, and for creating custom color spaces, using those colors in your drawing code is not recommended. Your drawing code should always use RGB colors.

Supported Image Formats

Table 4-1 lists the image formats supported directly by iPhone OS. Of these formats, the PNG format is the one most recommended for use in your applications.

Table 4-1 Supported image formats

Format	Filename extensions
Portable Network Graphic (PNG)	`.png`
Tagged Image File Format (TIFF)	`.tiff`, `.tif`
Joint Photographic Experts Group (JPEG)	`.jpeg`, `.jpg`
Graphic Interchange Format (GIF)	`.gif`
Windows Bitmap Format (DIB)	`.bmp`, `.BMPf`
Windows Icon Format	`.ico`
Windows Cursor	`.cur`
XWindow bitmap	`.xbm`

Drawing Tips

The following sections provide tips on how to write quality drawing code while ensuring that your application looks appealing to end users.

Deciding When to Use Custom Drawing Code

Depending on the type of application you are creating, it may be possible to use little or no custom drawing code. Although immersive applications typically make extensive use of custom drawing code, utility and productivity applications can often use standard views and controls to display their content.

The use of custom drawing code should be limited to situations where the content you display needs to change dynamically. For example, a drawing application would need to use custom drawing code to track the user's drawing commands and a game would be updating the screen constantly to reflect the changing game environment. In those situations, you would need to choose an appropriate drawing technology and create a custom view class to handle events and update the display appropriately.

On the other hand, if the bulk of your application's interface is fixed, you can render the interface in advance to one or more image files and display those images at runtime using `UIImageView` objects. You can layer image views with other content as needed to build your interface. For example, you could use `UILabel` objects to display configurable text and include buttons or other controls to provide interactivity.

Improving Drawing Performance

Drawing is a relatively expensive operation on any platform, and optimizing your drawing code should always be an important step in your development process. Table 4-2 lists several tips for ensuring that your drawing code is as optimal as possible. In addition to these tips, you should always use the available performance tools to test your code and remove hotspots and redundancies.

Table 4-2 Tips for improving drawing performance

Tip	Action
Draw minimally	During each update cycle, you should update only the portions of your view that actually changed. If you are using the `drawRect:` method of `UIView` to do your drawing, use the update rectangle passed to that method to limit the scope of your drawing. For OpenGL drawing, you must track updates yourself.
Mark opaque views as such	Compositing a view whose contents are opaque requires much less effort than compositing one that is partially transparent. To make a view opaque, the contents of the view must not contain any transparency and the `opaque` property of the view must be set to `YES`.
Remove alpha channels from opaque PNG files	If every pixel of a PNG image is opaque, removing the alpha channel avoids the need to blend the layers containing that image. This simplifies compositing of the image considerably and improves drawing performance.

Tip	Action
Reuse table cells and views during scrolling	Creating new views during scrolling should be avoided at all costs. Taking the time to create new views reduces the amount of time available for updating the screen, which leads to uneven scrolling behavior.
Avoid clearing the previous content during scrolling	By default, UIKit clears a view's current context buffer prior to calling its `drawRect:` method to update that same area. If you are responding to scrolling events in your view, clearing this region repeatedly during scrolling updates can be expensive. To disable the behavior, you can change the value in the `clearsContextBeforeDrawing` property to `NO`.
Minimize graphics state changes while drawing	Changing the graphics state requires effort by the window server. If you need to draw content that uses similar state information, try to draw that content together to reduce the number of state changes needed.

Maintaining Image Quality

Providing high-quality images for your user interface should be a priority in your design. Images provide a reasonably efficient way to display complicated graphics and should be used wherever they are appropriate. When creating images for your application, keep the following guidelines in mind:

- **Use the PNG format for images.** The PNG format provides high-quality image content and is the preferred image format for iPhone OS. In addition, iPhone OS includes an optimized drawing path for PNG images that is typically more efficient than other formats.
- **Create images so that they do not need resizing.** If you plan to use an image at a particular size, be sure to create the corresponding image resource at that size. Do not create a larger image and scale it down to fit, because scaling requires additional CPU cycles and requires interpolation. If you need to present an image at variable sizes, include multiple versions of the image at different sizes and scale down from an image that is relatively close to the target size.

Drawing with Quartz and UIKit

Quartz is the general name for the native window server and drawing technology in iPhone OS. The Core Graphics framework is at the heart of Quartz, and is the primary interface you use for drawing content. This framework provides data types and functions for manipulating the following:

- Graphics contexts
- Paths
- Images and bitmaps
- Transparency layers
- Colors, pattern colors, and color spaces
- Gradients and shadings
- Fonts

- PDF content

UIKit builds on the basic features of Quartz by providing a focused set of classes for graphics-related operations. The UIKit graphics classes are not intended as a comprehensive set of drawing tools—Core Graphics already provides that. Instead, they provide drawing support for other UIKit classes. UIKit support includes the following classes and functions:

- `UIImage`, which implements an immutable class for displaying images
- `UIColor`, which provides basic support for device colors
- `UIFont`, which provides font information for classes that need it
- `UIScreen`, which provides basic information about the screen
- Functions for generating a JPEG or PNG representation of a `UIImage` object
- Functions for drawing rectangles and clipping the drawing area
- Functions for changing and getting the current graphics context

For information about the classes and methods that comprise UIKit, see *UIKit Framework Reference*. For more information about the opaque types and functions that comprise the Core Graphics framework, see *Core Graphics Framework Reference*.

Configuring the Graphics Context

By the time your `drawRect:` method is called, your view's built-in drawing code has already created and configured a default graphics context for you. You can retrieve a pointer to this graphics context by calling the `UIGraphicsGetCurrentContext` function. This function returns a reference to a `CGContextRef` type, which you pass to Core Graphics functions to modify the current graphics state. Table 4-3 lists the main functions you use to set different aspects of the graphics state. For a complete list of functions, see *CGContext Reference*. This table also lists UIKit alternatives where they exist.

Table 4-3 Core graphics functions for modifying graphics state

Graphics state	Core Graphics functions	UIKit alternatives

Current transformation matrix (CTM)	`CGContextRotateCTM`
	`CGContextScaleCTM`
	`CGContextTranslateCTM`
	`CGContextConcatCTM`
		None

Clipping area	`CGContextClipToRect`	None

Line: Width, join, cap, dash, miter limit	`CGContextSetLineWidth`
	`CGContextSetLineJoin`
	`CGContextSetLineCap`
	`CGContextSetLineDash`
	`CGContextSetMiterLimit`
		None

Graphics state	Core Graphics functions	UIKit alternatives
Accuracy of curve estimation (flatness)	`CGContextSetFlatness`	None
Anti-aliasing setting	`CGContextSetAllowsAntialiasing`	None
Color: Fill and stroke settings	`CGContextSetRGBFillColor` `CGContextSetRGBStrokeColor`	`UIColor` class
Alpha value (transparency)	`CGContextSetAlpha`	None
Rendering intent	`CGContextSetRenderingIntent`	None
Color space: Fill and stroke settings	`CGContextSetFillColorSpace` `CGContextSetStrokeColorSpace`	None

Text: Font, font size, character spacing, text drawing mode	`CGContextSetFont`
	`CGContextSetFontSize`
	`CGContextSetCharacterSpacing`
		`UIFont` class

Blend mode	`CGContextSetBlendMode`	The `UIImage` class and various drawing functions let you specify which blend mode to use.

The graphics context contains a stack of saved graphics states. When Quartz creates a graphics context, the stack is empty. Using the `CGContextSaveGState` function pushes a copy of the current graphics state onto the stack. Thereafter, modifications you make to the graphics state affect subsequent drawing operations but do not affect the copy stored on the stack. When you are done making modifications, you can return to the previous graphics state by popping the saved state off the top of the stack using the `CGContextRestoreGState` function. Pushing and popping graphics states in this manner is a fast way to return to a previous state and eliminates the need to undo each state change individually. It is also the only way to restore some aspects of the state, such as the clipping path, back to their original settings.

For general information about graphics contexts and using them to configure the drawing environment, see Graphics Contexts in *Quartz 2D Programming Guide*.

Creating and Drawing Images

iPhone OS provides support for loading and displaying images using both the UIKit and Core Graphics frameworks. How you determine which classes and functions to use to draw images depends on how you intend to use them. Whenever possible, though, it is recommended that you use the classes of UIKit for representing images in your code. Table 4-4 lists some of the usage scenarios and the recommended options for handling them.

Table 4-4 Usage scenarios for images

Scenario	Recommended usage
Display an image as the content of a view	Use a `UIImageView` class to load and display the image. This option assumes that your view's only content is an image. You can still layer other views on top of the image view to draw additional controls or content.
Display an image as an adornment for part of a view	Load and draw the image using the `UIImage` class.
Save some bitmap data into an image object	Use the `UIGraphicsBeginImageContext` function to create a new image-based graphics context. After creating this context, you can draw your image contents into it and then use the `UIGraphicsGetImageFromCurrentImageContext` function to generate an image based on what you drew. (If desired, you can even continue drawing and generate additional images.) When you are done creating images, use the `UIGraphicsEndImageContext` function to close the graphic context. If you prefer using Core Graphics, you can use the `CGBitmapContextCreate` function to create a bitmap graphics context and draw your image contents into it. When you finish drawing, use the `CGBitmapContextCreateImage` function to create a `CGImageRef` from the bitmap context. You can draw the Core Graphics image directly or use this it to initialize a `UIImage` object.
Save an image as a JPEG or PNG file	Create a `UIImage` object from the original image data. Call the `UIImageJPEGRepresentation` or `UIImagePNGRepresentation` function to get an `NSData` object, and use that object's methods to save the data to a file.

The following example shows how to load an image from your application's bundle. You can subsequently use this image object to initialize a `UIImageView` object, or you can store it and draw it explicitly in your view's `drawRect:` method.

```
NSString* imagePath = [[NSBundle mainBundle] pathForResource:@"myImage"
ofType:@"png"];
UIImage* myImageObj = [[UIImage alloc] initWithContentsOfFile:imagePath];
```

To draw an image explicitly in your view's `drawRect:` method, you can use any of the drawing methods available in `UIImage`. These methods let you specify where in your view you want to draw the image and therefore do not require you to create and apply a separate transform prior to drawing. Assuming you stored the previously loaded image in a member variable called `anImage`, the following example draws that image at the point (10, 10) in the view.

```
- (void)drawRect:(CGRect)rect
{
    // Draw the image
    [anImage drawAtPoint:CGPointMake(10, 10)];
}
```

Important: If you use the `CGContextDrawImage` function to draw bitmap images directly, the image data is inverted along the y axis by default. This is because Quartz images assume a coordinate system with a lower-left corner origin and positive coordinate axes extending up and to the right from that point. Although you can apply a transform before drawing, the simpler (and recommended) way to draw Quartz images is to wrap them in a `UIImage` object, which compensates for this difference in coordinate spaces automatically. For more information on creating and drawing images using Core Graphics, see *Quartz 2D Programming Guide*.

Creating and Drawing Paths

A path is a description of a 2D geometric scene that uses a sequence of lines and Bézier curves to represent that scene. UIKit includes the `UIRectFrame` and `UIRectFill` functions (among others) for drawing simple paths such as rectangles in your views. Core Graphics also includes convenience functions for creating simple paths such as rectangles and ellipses. For more complex paths, you must create the path yourself using the functions of the Core Graphics framework.

To create a path, you use the `CGContextBeginPath` function to configure the graphics context to receive path commands. After calling that function, you use other path-related functions to set the path's starting point, draw lines and curves, add rectangles and ellipses, and so on. When you are done specifying the path geometry, you can paint the path directly or create a `CGPathRef` or `CGMutablePathRef` data type to store a reference to that path for later use.

When you want to draw a path in your view, you can stroke it, fill it, or do both. Stroking a path with a function such as `CGContextStrokePath` creates a line centered on the path using the current stroke color. Filling the path with the `CGContextFillPath` function uses the current fill color or fill pattern to fill the area enclosed by the path's line segments.

For more information on how to draw paths, including information about how you specify the points for complex path elements, see Paths in *Quartz 2D Programming Guide*. For information on the functions you use to create paths, see *CGContext Reference* and *CGPath Reference*.

Creating Patterns, Gradients, and Shadings

The Core Graphics framework includes additional functions for creating patterns, gradients, and shadings. You use these types to create non monochrome colors and use them to fill the paths you create. Patterns are created from repeating images or content. Gradients and shadings provide different ways to create smooth transitions from color to color.

The details for creating and using patterns, gradients, and shadings are all covered in *Quartz 2D Programming Guide*.

Drawing with OpenGL ES

The **Open Graphics Library (OpenGL)** is a cross-platform C-based interface used to create 2D and 3D content on desktop systems. It is typically used by games developers or anyone needing to perform drawing with high frame rates. You use OpenGL functions to specify primitive structures such as points, lines, and polygons

and the textures and special effects to apply to those structures to enhance their appearance. The functions you call send graphics commands to the underlying hardware, where they are then rendered. Because rendering is done mostly in hardware, OpenGL drawing is usually very fast.

OpenGL for Embedded Systems is a pared-down version of OpenGL that is designed for mobile devices and takes advantage of modern graphics hardware. If you want to create OpenGL content for iPhone OS–based devices—that is, iPhone or iPod Touch—you'll use OpenGL ES. The OpenGL ES framework (`OpenGLES.framework`) provided with iPhone OS supports both the OpenGL ES v1.1 and OpenGL ES v2.0 specifications.

For more information about OpenGL ES support in iPhone OS, see *OpenGL ES Programming Guide for iPhone OS*.

Applying Core Animation Effects

Core Animation is an Objective-C framework that provides infrastructure for creating fluid, real-time animations quickly and easily. Core Animation is not a drawing technology itself, in the sense that it does not provide primitive routines for creating shapes, images, or other types of content. Instead, it is a technology for manipulating and displaying content that you created using other technologies.

Most applications can benefit from using Core Animation in some form in iPhone OS. Animations provide feedback to the user about what is happening. For example, when the user navigates through the Settings application, screens slide in and out of view based on whether the user is navigating further down the preferences hierarchy or back up to the root node. This kind of feedback is important and provides contextual information for the user. It also enhances the visual style of an application.

In most cases, you may be able to reap the benefits of Core Animation with very little effort. For example, several properties of the `UIView` class (including the view's frame, center, color, and opacity—among others) can be configured to trigger animations when their values change. You have to do some work to let UIKit know that you want these animations performed, but the animations themselves are created and run automatically for you. For information about how to trigger the built-in view animations, see "Animating Views" (page 69).

When you go beyond the basic animations, you must interact more directly with Core Animation classes and methods. The following sections provide information about Core Animation and show you how to work with its classes and methods to create typical animations in iPhone OS. For additional information about Core Animation and how to use it, see *Core Animation Programming Guide*.

About Layers

The key technology in Core Animation is the layer object. Layers are lightweight objects that are similar in nature to views, but that are actually model objects that encapsulate geometry, timing, and visual properties for the content you want to display. The content itself is provided in one of three ways:

- You can assign a `CGImageRef` to the `contents` property of the layer object.
- You can assign a delegate to the layer and let the delegate handle the drawing.
- You can subclass `CALayer` and override one of the display methods.

When you manipulate a layer object's properties, what you are actually manipulating is the model-level data that determines how the associated content should be displayed. The actual rendering of that content is handled separately from your code and is heavily optimized to ensure it is fast. All you must do is set the layer content, configure the animation properties, and then let Core Animation take over.

For more information about layers and how they are used, see *Core Animation Programming Guide*.

About Animations

When it comes to animating layers, Core Animation uses separate animation objects to control the timing and behavior of the animation. The `CAAnimation` class and its subclasses provide different types of animation behaviors that you can use in your code. You can create simple animations that migrate a property from one value to another, or you can create complex keyframe animations that track the animation through the set of values and timing functions you provide.

Core Animation also lets you group multiple animations together into a single unit, called a transaction. The `CATransaction` object manages the group of animations as a unit. You can also use the methods of this class to set the duration of the animation.

For examples of how to create custom animations, see *Animation Types and Timing Programming Guide*.

Text and Web

The text system in iPhone OS was designed with the modest needs of mobile users in mind. The text system was designed to handle the single and multi-line text input that are commonly used in email and SMS programs. The text system also supports Unicode and has several different input methods, making it easy to display and read text in many different languages.

Important: The UIKit classes are generally not thread safe. All work related to your application's user interface should be performed on your application's main thread.

About Text and Web Support

The text system in iPhone OS provides a tremendous amount of power while still being very simple to use. The UIKit framework includes several high-level classes for managing the display and input of text. This framework also includes a more advanced class for displaying HTML and JavaScript-based content.

The following sections describe the basic support for text and web content in iPhone OS. For more information about each of the classes listed in this section, see *UIKit Framework Reference*.

Text Views

The UIKit framework provides three primary classes for displaying text content:

- `UILabel` displays static text strings
- `UITextField` displays a single line of editable text
- `UITextView` displays multiple lines of editable text

These classes support the display of arbitrarily large amounts of text, although labels and text fields are typically used for relatively small amounts of text. To make the displayed text easier to read on the smaller screens of iPhone OS–based devices, however, these classes do not support the kinds of advanced formatting you might find in desktop operating systems like Mac OS X. All three classes still allow you to specify the font information, including size and styling options, that you might otherwise want, but the font information you specify is applied to all of the text associated with the object.

Figure 5-1 shows examples of the available text classes as they appear on screen. These examples were taken from the *UICatalog* sample application, which demonstrates many of the views and controls available in UIKit. The image on the left shows several different styles of text fields while the image on the right shows a single text view. The callouts displayed on the gray background are themselves `UILabel` objects embedded inside the table cells used to display the different views. There is also a `UILabel` object with the text "Left View" at the bottom of the screen on the left.

Figure 5-1 Text classes in the UICatalog application

When working with editable text views, you should always provide a delegate object to manage the editing session. Text views send several different notifications to the delegate to let them know when editing begins, when it ends, and to give them a chance to override some editing actions. For example, the delegate can decide if the current text contains a valid value and prevent the editing session from ending if it does not. When editing does finally end, you also use the delegate to get the resulting text value and update your application's data model.

Because there are slight differences in their intended usage, the delegate methods for each text view are slightly different. A delegate that supports the `UITextField` class implements the methods of the `UITextFieldDelegate` protocol. Similarly, a delegate that supports the `UITextView` class implements the methods of the `UITextViewDelegate` protocol. In both cases, you are not required to implement any of the protocol methods but if you do not, the text field is not going to be of much use to you. For more information on the methods in these two protocols, see *UITextFieldDelegate Protocol Reference* and *UITextViewDelegate Protocol Reference*.

Web View

The `UIWebView` class lets you integrate what is essentially a miniature web browser into your application's user interface. The `UIWebView` class makes full use of the same web technologies used to implement Safari in iPhone OS, including full support for HTML, CSS, and JavaScript content. The class also supports many of the built-in gestures that users are familiar with in Safari. For example, you can double-click and pinch to zoom in and out of the page and you can scroll around the page by dragging your finger.

In addition to displaying content, you can also use a web view object to gather input from the user through the use of web forms. Like the other text classes in UIKit, if you have an editable text field on a form in your web page, tapping that field brings up a keyboard so that the user can enter text. Because it is an integral part of the web experience, the web view itself manages the displaying and dismissing of the keyboard for you.

Figure 5-2 shows an example of a `UIWebView` object from the the *UICatalog* sample application, which demonstrates many of the views and controls available in UIKit. Because it just displays HTML content, if you want the user to be able to navigate pages much like they would in a web browser, you need to add controls to do so. For example, the web view in the figure occupies the space below the text field containing the target URL and does not contain the text field itself.

Figure 5-2 A web view

A web view provides information about when pages are loaded, and whether there were any load errors, through its associated delegate object. A web delegate is an object that implements one or more methods of the `UIWebViewDelegate` protocol. Your implementations of the delegate methods can respond to failures or perform other tasks related to the loading of a web page. For more information about the methods of the `UIWebViewDelegate` protocol, see *UIWebViewDelegate Protocol Reference*.

Keyboards and Input Methods

Whenever the user taps in an object capable of accepting text input, the object asks the system to display an appropriate keyboard. Depending on the needs of your program and the user's preferred language, the system might display one of several different keyboards. Although your application cannot control the user's preferred language (and thus the keyboard's input method), it can control attributes of the keyboard that indicate its intended use, such as the configuration of any special keys and its behaviors.

You configure the attributes of the keyboard directly through the text objects of your application. The `UITextField` and `UITextView` classes both conform to the `UITextInputTraits` protocol, which defines the properties for configuring the keyboard. Setting these properties programmatically or in the Interface Builder inspector window causes the system to display the keyboard of the designated type.

Note: Although the `UIWebView` class does not support the `UITextInputTraits` protocol directly, you can configure some keyboard attributes for text input elements. In particular, you can include `autocorrect` and `autocapitalization` attributes in the definition of an input element to specify the keyboard's behaviors, as shown in the following example.

```
<input type="text" size="30" autocorrect="off" autocapitalization="on">
```

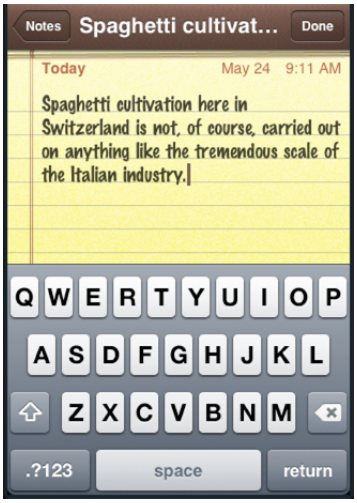
You cannot specify the keyboard type in input elements. The web view displays a custom keyboard that is based on the default keyboard but includes some additional controls for navigating between form elements.

The default keyboard configuration is designed for general text input. Figure 5-3 displays the default keyboard along with several other keyboard configurations. The default keyboard displays an alphabetical keyboard initially but the user can toggle it and display numbers and punctuation as well. Most of the other keyboards offer similar features as the default keyboard but provide additional buttons that are specially suited to particular tasks. However, the phone and numerical keyboards offer a dramatically different layout that is tailored towards numerical input.

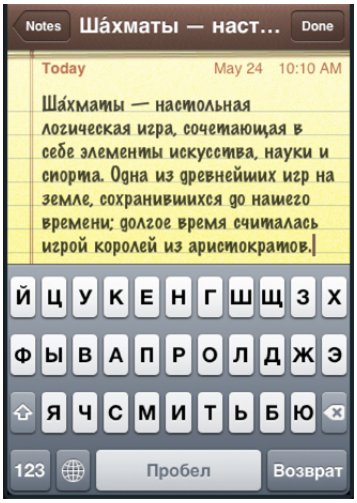
Figure 5-3 Several different keyboard types

To facilitate the language preferences of different users, iPhone OS also supports different input methods and keyboard layouts for different languages, some of which are shown in Figure 5-4. The input method and layout for the keyboard is determined by the user's language preferences.

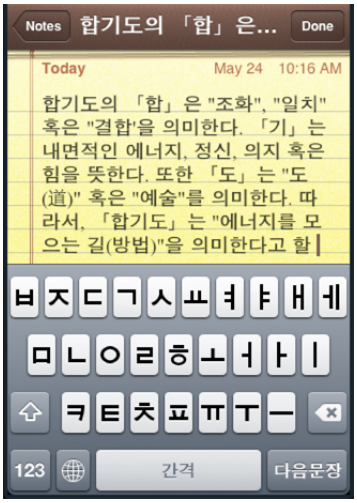
Figure 5-4 Several different keyboards and input methods



English



Russian



Korean

Japanese - Romanji

Japanese - Kana

Chinese - Handwriting

Managing the Keyboard

Although many UIKit objects display the keyboard automatically in response to user interactions, your application still has some responsibilities for configuring and managing the keyboard. The following sections describe those responsibilities.

Receiving Keyboard Notifications

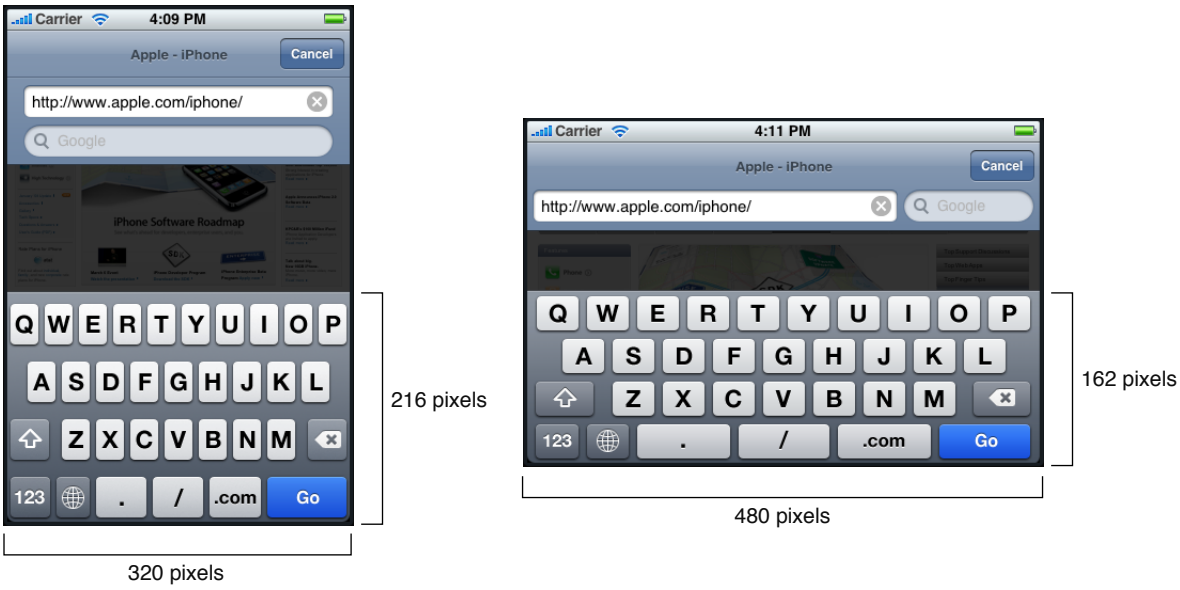
When the keyboard is shown or hidden, iPhone OS sends out the following notifications to any registered observers:

- `UIKeyboardWillShowNotification`
- `UIKeyboardDidShowNotification`
- `UIKeyboardWillHideNotification`
- `UIKeyboardDidHideNotification`

The system sends keyboard notifications when the keyboard first appears, when it disappears, any time the owner of the keyboard changes, or any time your application's orientation changes. In each situation, the system sends only the appropriate subset of messages. For example, if the owner of the keyboard changes, the system sends a `UIKeyboardWillHideNotification` message, but not a `UIKeyboardDidHideNotification` message, to the current owner because the change never causes the keyboard to be hidden. The delivery of the `UIKeyboardWillHideNotification` is simply a way to alert the current owner that it is about to lose the keyboard focus. Changing the orientation of the keyboard does send both will and did hide notifications, however, because the keyboards for each orientation are different and thus the original must be hidden before the new one is displayed.

Each keyboard notification includes information about the size and position of the keyboard on the screen. You should always use the information in these notifications as opposed to assuming the keyboard is a particular size or in a particular location. The size of the keyboard is not guaranteed to be the same from one input method to another and may also change between different releases of iPhone OS. In addition, even for a single language and system release, the keyboard dimensions can vary depending on the orientation of your application. For example, Figure 5-5 shows the relative sizes of the URL keyboard in both the portrait and landscape modes. Using the information inside the keyboard notifications ensures that you always have the correct size and position information.

Figure 5-5 Relative keyboard sizes in portrait and landscape modes



Note: The rectangle contained in the `UIKeyboardBoundsUserInfoKey` of the info dictionary should be used only for the size information it contains. Do not use the origin of the rectangle (which is always {0.0, 0.0}) in rectangle-intersection operations. Because the keyboard is animated into position, the actual bounding rectangle of the keyboard changes over time. The starting and ending positions of the keyboard are therefore stored in the info dictionary under the `UIKeyboardCenterBeginUserInfoKey` and `UIKeyboardCenterEndUserInfoKey` keys, and from these values you can compute the origin.

One reason to use keyboard notifications is so that you can reposition content that is obscured by the keyboard when it is visible. For information on how to handle this scenario, see "Moving Content That Is Located Under the Keyboard" (page 127).

Displaying the Keyboard

When the user taps a view, the system automatically designates that view as the first responder. When this happens to a view that contains editable text, the view initiates an editing session for that text. At the beginning of that editing session, the view asks the system to display the keyboard, if it is not already visible. If the keyboard is already visible, the change in first responder causes text input from the keyboard to be redirected to the newly tapped view.

Because the keyboard is displayed automatically when a view becomes the first responder, you often do not need to do anything to display it. However, you can programmatically display the keyboard for an editable text view by calling that view's `becomeFirstResponder` method. Calling this method makes the target view the first responder and begins the editing process just as if the user had tapped on the view.

If your application manages several text-based views on a single screen, it is a good idea to track which view is currently the first responder so that you can dismiss the keyboard later.

Dismissing the Keyboard

Although it typically displays the keyboard automatically, the system does not dismiss the keyboard automatically. Instead, it is your application's responsibility to dismiss the keyboard at the appropriate time. Typically, you would do this in response to a user action. For example, you might dismiss the keyboard when the user taps the Return or Done button on the keyboard or taps some other button in your application's interface. Depending on how you configured the keyboard, you might need to add some additional controls to your user interface to facilitate the keyboard's dismissal.

To dismiss the keyboard, you call the `resignFirstResponder` method of the text-based view that is currently the first responder. When a text view resigns its first responder status, it ends its current editing session, notifies its delegate of that fact, and dismisses the keyboard. In other words, if you have a variable called `myTextField` that points to the `UITextField` object that is currently the first responder, dismissing the keyboard is as simple as doing the following:

```
[myTextField resignFirstResponder];
```

Everything from that point on is handled for you automatically by the text object.

Moving Content That Is Located Under the Keyboard

When asked to display the keyboard, the system slides it in from the bottom of the screen and positions it over your application's content. Because it is placed on top of your content, it is possible for the keyboard to be placed on top of the text object that the user wanted to edit. When this happens, you must adjust your content so that the target object remains visible.

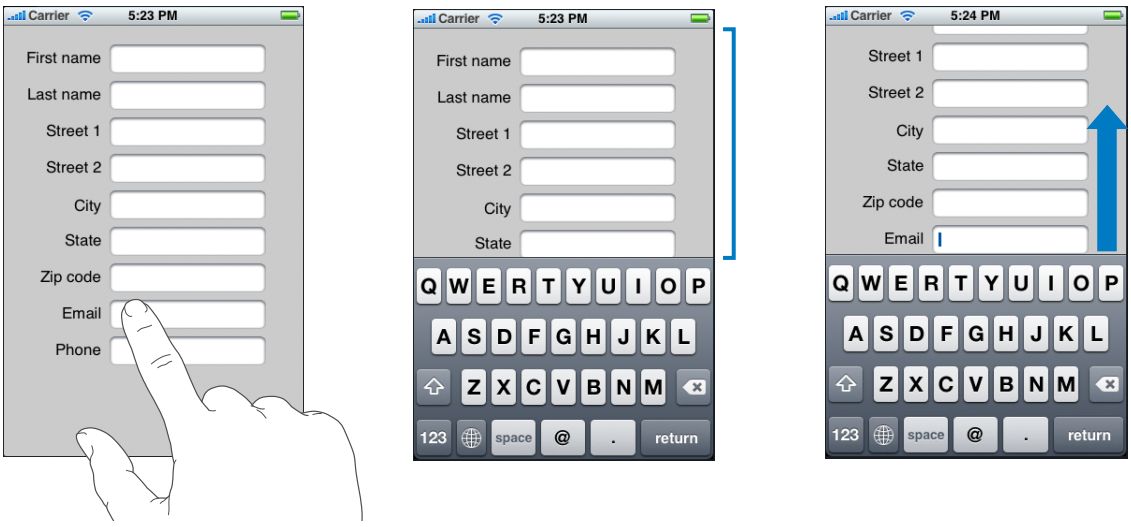
Adjusting your content typically involves temporarily resizing one or more views and positioning them so that the text object remains visible. The simplest way to manage text objects with the keyboard is to embed them inside a `UIScrollView` object (or one of its subclasses like `UITableView`). When the keyboard is displayed, all you have to do is resize the scroll view and scroll the desired text object into position. Thus, in response to a `UIKeyboardDidShowNotification`, your handler method would do the following:

1. Get the size of the keyboard.
2. Subtract the keyboard height from the height of your scroll view.
3. Scroll the target text field into view.

Note: As of iPhone OS 3.0, the `UITableViewController` class automatically resizes and repositions its table view when there is in-line editing of text fields. See "View Controllers and Navigation-Based Applications" in *Table View Programming Guide for iPhone OS*.

Figure 5-6 illustrates the preceding steps for a simple application that embeds several text fields inside a `UIScrollView` object. When the keyboard appears, the notification handler method resizes the scroll view and then uses the `scrollRectToVisible:animated:` method of `UIScrollView` to scroll the tapped text field (in this case the email field) into view.

Figure 5-6 Adjusting content to accommodate the keyboard



1. User taps email field
2. Scroll view resized to new height
3. Email field scrolled into view

Note: When setting up your own scroll views, be sure to configure the autoresizing rules for any content views appropriately. In the preceding figure, the text fields are actually subviews of a generic `UIView` object, which is itself a subview of the `UIScrollView` object. If the generic view's `UIViewAutoresizingFlexibleWidth` and `UIViewAutoresizingFlexibleHeight` autoresizing options are set, changing the scroll view's frame size also changes the frame of the generic view, which could yield undesirable results. Disabling these options for that view ensures that the view retains its size and its contents are scrolled correctly.

Listing 5-1 shows the code for registering to receive keyboard notifications and shows the handler methods for those notifications. This code is implemented by the view controller that manages the scroll view, and the `scrollView` variable is an outlet that points to the scroll view object. Each handler method gets the keyboard size from the info dictionary of the notification and adjusts the scroll view height by the corresponding amount. In addition, the `keyboardWasShown:` method scrolls the rectangle of the active text field, which is stored in a custom variable (called `activeField` in this example) that is a member variable of the view controller and set in the `textFieldDidBeginEditing:` delegate method, which is itself shown in Listing 5-2 (page 129). (In this example, the view controller also acts as the delegate for each of the text fields.)

Listing 5-1 Handling the keyboard notifications

```
// Call this method somewhere in your view controller setup code.
- (void)registerForKeyboardNotifications
{
    [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(keyboardWasShown:)
            name:UIKeyboardDidShowNotification object:nil];

    [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(keyboardWasHidden:)
            name:UIKeyboardDidHideNotification object:nil];
}

// Called when the UIKeyboardDidShowNotification is sent.
- (void)keyboardWasShown:(NSNotification*)aNotification
{
    if (keyboardShown)
        return;

    NSDictionary* info = [aNotification userInfo];

    // Get the size of the keyboard.
    NSValue* aValue = [info objectForKey:UIKeyboardBoundsUserInfoKey];
    CGSize keyboardSize = [aValue CGRectValue].size;

    // Resize the scroll view (which is the root view of the window)
    CGRect viewFrame = [scrollView frame];
    viewFrame.size.height -= keyboardSize.height;
    scrollView.frame = viewFrame;

    // Scroll the active text field into view.
    CGRect textFieldRect = [activeField frame];
    [scrollView scrollRectToVisible:textFieldRect animated:YES];

    keyboardShown = YES;
}
```

```
// Called when the UIKeyboardDidHideNotification is sent
- (void)keyboardWasHidden:(NSNotification*)aNotification
{
    NSDictionary* info = [aNotification userInfo];

    // Get the size of the keyboard.
    NSValue* aValue = [info objectForKey:UIKeyboardBoundsUserInfoKey];
    CGSize keyboardSize = [aValue CGRectValue].size;

    // Reset the height of the scroll view to its original value
    CGRect viewFrame = [scrollView frame];
    viewFrame.size.height += keyboardSize.height;
    scrollView.frame = viewFrame;

    keyboardShown = NO;
}
```

The `keyboardShown` variable from the preceding listing is a Boolean value used to track whether the keyboard is already visible. If your interface has multiple text fields, the user can tap between them to edit the values in each one. When that happens, however, the keyboard does not disappear but the system does still generate `UIKeyboardDidShowNotification` notifications each time editing begins in a new text field. By tracking whether the keyboard was actually hidden, this code prevents the scroll view from being reduced in size more than once.

Listing 5-2 shows some additional code used by the view controller to set and clear the `activeField` variable in the preceding example. During initialization, each text field in the interface sets the view controller as its delegate. Therefore, when a text field becomes active, it calls these methods. For more information on text fields and their delegate notifications, see *UITextField Class Reference*.

Listing 5-2 Additional methods for tracking the active text field.

```
- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    activeField = textField;
}

- (void)textFieldDidEndEditing:(UITextField *)textField
{
    activeField = nil;
}
```

There are other ways you can scroll the edited area in a scroll view above an obscuring keyboard. Instead of altering the frame of the scroll view, you can extend the height of the content view by the height of the keyboard and then scroll the edited area into view. The `UIScrollView` class has a `contentSize` property that you can set for this purpose; you can also adjust the frame of the content view, as shown in Listing 5-3. This code also uses the `setContentOffset:animated:` method to scroll the edited field into view, in this case scrolling it just below the top of the scroll view instead of just above the keyboard.

Listing 5-3 Adjusting the frame of the content view and scrolling a field above the keyboard

```
- (void)keyboardWasShown:(NSNotification*)aNotification
{
    if (keyboardShown)
        return;
```

```
    NSDictionary* info = [aNotification userInfo];
    CGSize kbSize = [[info objectForKey:UIKeyboardBoundsUserInfoKey]
CGRectValue].size;
    CGRect bkgndRect = activeField.superview.frame;
    bkgndRect.size.height += kbSize.height;
    [activeField.superview setFrame:bkgndRect];
    [scrollView setContentOffset:CGPointMake(0.0, activeField.frame.origin.y)
animated:YES];

    keyboardShown = YES;
}
```

Drawing Text

In addition to the UIKit classes for displaying and editing text, iPhone OS also includes several ways to draw text directly on the screen. The easiest and most efficient way to draw simple strings is using the UIKit additions to the `NSString` class. These extensions include methods for drawing strings using a variety of attributes wherever you want them on the screen. There are also methods for computing the size of a rendered string before you actually draw it, which can help you lay out your application content more precisely.

Important: Because of the performance implications, you should avoid drawing text directly whenever possible. Static text can be drawn much more efficiently using one or more `UILabel` objects than it can be using a custom drawing routine. Similarly, the `UITextField` class includes different styles that make it easier to integrate editable text areas into your content.

When you need to draw custom text strings in your interface, use the methods of `NSString` to do so. UIKit includes extensions to the basic `NSString` class that allow you to draw strings in your views. These methods allow you to adjust the position of the rendered text precisely and blend it with the rest of your view's content. The methods of this class also let you compute the bounding rectangle for your text in advance based on the desired font and style attributes. For information, see *NSString UIKit Additions Reference*.

If you need more control over the fonts you plan to use during drawing, you can also use the functions of the Core Graphics framework to do your drawing. The Core Graphics frameworks provides methods for the precise drawing and placement of glyphs and text. For more information about these functions and their use, see *Quartz 2D Programming Guide* and *Core Graphics Framework Reference*.

Displaying Content in a Web View

If your user interface includes a `UIWebView` object, you can display local content or content that is loaded from the network. When loading local content, you can either create the content dynamically or load it from a file and display it using the `loadData:MIMEType:textEncodingName:baseURL:` or `loadHTMLString:baseURL:` method. The method in Listing 5-4 uses the `loadData:MIMEType:textEncodingName:baseURL:` method to load the contents of a PDF file into a web view.

Listing 5-4 Loading a local PDF file into the web view

```
- (void)viewDidLoad {
```

```
    [super viewDidLoad];

    NSString *thePath = [[NSBundle mainBundle]
pathForResource:@"iPhone_User_Guide" ofType:@"pdf"];
    if (thePath) {
        NSData *pdfData = [NSData dataWithContentsOfFile:thePath];
        [(UIWebView *)self.view loadData:pdfData MIMEType:@"application/pdf"
            textEncodingName:@"utf-8" baseURL:nil];
    }
}
```

To load content from the network, you create an `NSURLRequest` object and pass it to the `loadRequest:` method of your web view.

```
[self.myWebView loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.apple.com/"]]];
```

Because loading a web resource might take some time, you might displaying an activity indicator to indicate that the load is underway. You can do this by assigning a delegate to the web view and implementing the `UIWebViewDelegate` methods, as in Listing 5-5. The delegate displays an activity indicator when the load starts and hides it when the load ends. If there is a problem with the load, it creates an HTML error message and, using the `loadHTMLString:baseURL:` method, loads it into the web view for display.

Listing 5-5 The web-view delegate managing network loading

```
- (void)webViewDidStartLoad:(UIWebView *)webView
{
    // starting the load, show the activity indicator in the status bar
    [UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
}

- (void)webViewDidFinishLoad:(UIWebView *)webView
{
    // finished loading, hide the activity indicator in the status bar
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}

- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error
{
    // load error, hide the activity indicator in the status bar
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;

    // report the error inside the webview
    NSString* errorString = [NSString stringWithFormat:
                             @"<html><center><font size=+5 color='red'>An error
 occurred:<br>%@</font></center></html>",
                              error.localizedDescription];
    [self.myWebView loadHTMLString:errorString baseURL:nil];
}
```

If, after initiating a network-based load request, you must release your web view for any reason, you must cancel the pending request before releasing the web view. You can cancel a load request using the web view's `stopLoading` method. A typical place to include this code would be in the `viewWillDisappear:` method of the owning view controller. To determine if a request is still pending, you can check the value in the web view's `loading` property. Listing 5-6 illustrates how you might do this.

Listing 5-6 Stopping a load request when the web view is to disappear

```
- (void)viewWillDisappear:(BOOL)animated
{
    if ( [self.myWebView loading] ) {
        [self.myWebView stopLoading];
    }
    self.myWebView.delegate = nil;    // disconnect the delegate as the webview
 is hidden
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}
```

The `loadRequest:` example is taken from the *UICatalog* sample code project.

Files and Networking

An application running in iPhone OS has access to the local file system and network, which you can access using assorted Core OS and Core Services frameworks. Being able to read and write files in the local file system lets you save user data and application state for subsequent uses. Accessing the network gives you the ability to communicate with network servers to perform remote operations and send and retrieve data.

File and Data Management

Files in iPhone OS share space with the user's media and personal files on the flash-based memory. For security purposes, your application is placed in its own directory and is limited to reading and writing files in that directory only. The following sections describe the structure of an application's local file system and several techniques for reading and writing files.

Commonly Used Directories

For security purposes, an application has only a few locations in which it can write its data and preferences. When an application is installed on a device, a home directory is created for the application. Table 6-1 lists some of the important subdirectories inside the home directory that you might need to access. This table describes the intended usage and access restrictions for each directory and whether the directory's contents are backed up by iTunes. For more information about the backup and restore process, see "Backup and Restore" (page 134). For more information about the application home directory itself, see "The Application Sandbox" (page 23).

Table 6-1 Directories of an iPhone application

Directory	Description
<*Application_Home*>/*AppName*.app	This is the bundle directory containing the application itself. Because an application must be signed, you must not make changes to the contents of this directory at runtime. Doing so may prevent your application from launching later. In iPhone OS 2.1 and later, the contents of this directory are not backed up by iTunes. However, iTunes does perform an initial sync of any applications purchased from the App Store.

Directory	Description
<*Application_Home*>/Documents/	This is the directory you should use to write any application-specific data files. Use this directory to store user data or other information that should be backed up regularly. For information about how to get the path of this directory, see "Getting Paths to Application Directories" (page 136). The contents of this directory are backed up by iTunes.
<*Application_Home*>/Library/Preferences	This directory contains application-specific preference files. You should not create preferences files directly but should instead use the `NSUserDefaults` class or *CFPreferences* API to get and set application preferences; see "Adding the Settings Bundle" (page 200). The contents of this directory are backed up by iTunes.
<*Application_Home*>/Library/Caches	Use this directory to write any application-specific support files that you want to persist between launches of the application. Your application is generally responsible for adding and removing these files. However, iTunes removes these files during a full restore of the device so you should be able to recreate them as needed. To access this directory, use the interfaces described in "Getting Paths to Application Directories" (page 136) to get the path to the directory. In iPhone OS 2.2 and later, the contents of this directory are not backed up by iTunes.
<*Application_Home*>/tmp/	Use this directory to write temporary files that you do not need to persist between launches of your application. Your application should remove files from this directory when it determines they are no longer needed. (The system may also purge lingering files from this directory when your application is not running.) For information about how to get the path of this directory, see "Getting Paths to Application Directories" (page 136). In iPhone OS 2.1 and later, the contents of this directory are not backed up by iTunes.

Backup and Restore

You do not have to prepare your application in any way for backup and restore operations. In iPhone OS 2.2 and later, when a device is plugged into a computer and synced, iTunes performs an incremental backup of all files, except for those in the following directories:

- <*Application_Home*>/*AppName*.app
- <*Application_Home*>/Library/Caches
- <*Application_Home*>/tmp

Although iTunes does back up the application bundle itself, it does not do this during every sync operation. Applications purchased from the App Store on the device are backed up when that device is next synced with iTunes. Applications are not backed up during subsequent sync operations though unless the application bundle itself has changed (because the application was updated, for example).

To prevent the syncing process from taking a long time, you should be selective about where you place files inside your application's home directory. The *<Application_Home>*`/Documents` directory should be used to store user data files or files that cannot be easily recreated by your application. Files used to store temporary data should be placed inside the *Application Home*`/tmp` directory and deleted by your application when they are no longer needed. If your application creates data files that can be used during subsequent launches, it should place those files in the *Application Home*`/Library/Caches` directory.

Note: If your application creates large data files, or files that change frequently, you should consider storing them in the *Application Home*`/Library/Caches` directory and not in the *<Application_Home>*`/Documents` directory. Backing up large data files can slow down the backup process significantly. The same is true for files that change regularly (and therefore must be backed up frequently). Placing these files in the `Caches` directory prevents them from being backed up (in iPhone OS 2.2 and later) during every sync operation.

For additional guidance about how you should use the directories in your application, see Table 6-1 (page 133).

Files Saved During Application Updates

Updating an application replaces the previous application with the new one downloaded by the user. During this process, iTunes installs the updated application in a new application directory. It then moves the user's data files from the old installation over to the new application directory before deleting the old installation. Files in the following directories are guaranteed to be preserved during the update process:

- *<Application_Home>*`/Documents`
- *<Application_Home>*`/Library/Preferences`

Although files in other user directories may also be moved over, you should not rely on them being present after an update.

Keychain Data

A keychain is a secure, encrypted container for passwords and other secrets. The keychain data for an application is stored outside of the application sandbox. If an application is uninstalled, that data is automatically removed. When the user backs up application data using iTunes, the keychain data is also backed up. However, it can only be restored to the device from which the backup was made. An upgrade of an application does not affect its keychain data.

For more on the iPhone OS keychain, see "Keychain Services Concepts" in *Keychain Services Programming Guide*.

Getting Paths to Application Directories

At various levels of the system, there are programmatic ways to obtain file-system paths to the directory locations of an application's sandbox. However, the preferred way to retrieve these paths is with the Cocoa programming interfaces.

You can use the `NSSearchPathForDirectoriesInDomains` function to get exact paths to your `Documents` and `Caches` directories. To get a path to the `tmp` directory, us the `NSTemporaryDirectory` function. When you have one of these "base" paths, you can use the path-related methods of `NSString` to modify the path information or create new path strings. For example, upon retrieving the temporary directory path, you could append a file name and use the resulting string to create a file with the given name in the temporary directory.

Note: If you are using frameworks with ANSI C programmatic interfaces—including those that take paths—recall that `NSString` objects are "toll-free bridged" with their Core Foundation counterparts. This means that you can cast a `NSString` object (such as the return by one of the above functions) to a `CFStringRef` type, as shown in the following example:

```
CFStringRef tmpDirectory = (CFStringRef)NSTemporaryDirectory();
```

For more information on toll-free bridging, see *Carbon-Cocoa Integration Guide*.

The `NSSearchPathForDirectoriesInDomains` function of the Foundation framework lets you obtain the full path to several application-related directories. To use this function in iPhone OS, specify an appropriate search path constant for the first parameter and `NSUserDomainMask` for the second parameter. Table 6-2 lists several of the most commonly used constants and the directories they return.

Table 6-2 Commonly used search path constants

Constant	Directory
`NSDocumentDirectory`	*<Application_Home>*`/Documents`
`NSCachesDirectory`	*<Application_Home>*`/Library/Caches`
`NSApplicationSupportDirectory`	*<Application_Home>*`/Library/Application Support`

Because the `NSSearchPathForDirectoriesInDomains` function was designed originally for Mac OS X, where multiple such directories could exist, it returns an array of paths rather than a single path. In iPhone OS, the resulting array should contain the single path to the desired directory. Listing 6-1 shows a typical use of this function.

Listing 6-1 Getting a file-system path to the application's `Documents/` directory

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

You can call `NSSearchPathForDirectoriesInDomains` using a domain-mask parameter other than `NSUserDomainMask` or a directory constant other than those in Table 6-2 (page 136), but the application will be unable to write to any of the returned directory locations. For example, if you specify `NSApplicationDirectory` as the directory parameter and `NSSystemDomainMask` as the domain-mask parameter, you get the path `/Applications` returned (on the device), but your application cannot write any files to this location.

Another consideration to keep in mind is the difference in directory locations between platforms. The paths returned by the `NSSearchPathForDirectoriesInDomains` function differ depending on whether you're running your application on the device or on the Simulator. For example, take the function call shown in Listing 6-1 (page 136). On the device, the returned path (`documentsDirectory`) is similar to the following:

```
/var/mobile/Applications/30B51836-D2DD-43AA-BCB4-9D4DADFED6A2/Documents
```

However, on the Simulator, the returned path takes the following form:

```
/Volumes/Stuff/Users/johnDoe/Library/Application Support/iPhone
Simulator/User/Applications/118086A0-FAAF-4CD4-9A0F-CD5E8D287270/Documents
```

To read and write user preferences, use the `NSUserDefaults` class or the `CFPreferences` API. These interfaces eliminate the need for you to construct a path to the `Library/Preferences/` directory and read and write preference files directly. For more information on using these interfaces, see "Adding the Settings Bundle" (page 200).

If your application contains sound, image, or other resources in the application bundle, you should use the `NSBundle` class or `CFBundleRef` opaque type to load those resources. Bundles have an inherent knowledge of where resources live inside the application. In addition, bundles are aware of the user's language preferences and are able to choose localized resources over default resources automatically. For more information on bundles, see "The Application Bundle" (page 24).

Reading and Writing File Data

The iPhone OS provides several ways to read, write, and manage files.

- Foundation framework:
  - If you can represent your application's data as a property list, convert the property list to an `NSData` object using the `NSPropertyListSerialization` API. You can then write the data object to disk using the methods of the `NSData` class.
  - If your application's model objects adopt the `NSCoding` protocol, you can archive a graph of these model objects using the `NSKeyedArchiver` class, and especially its `archivedDataWithRootObject:` method.
  - The `NSFileHandle` class in Foundation framework provides random access to the contents of a file.
  - The `NSFileManager` class in Foundation framework provides methods to create and manipulate files in the file system.
- Core OS calls:
  - Calls such as `fopen`, `fread`, and `fwrite` also let you read and write file data either sequentially or via random access.
  - The `mmap` and `munmap` calls provide an efficient way to load large files into memory and access their contents.

Note: The preceding list of Core OS calls is just a sample of the more commonly used calls. For a more complete list of the available functions, see the list of functions in section 3 of *iPhone OS Manual Pages*.

The following sections show examples of how to use some of the higher-level techniques for reading and writing files. For additional information about the file-related classes of the Foundation framework, see *Foundation Framework Reference*.

Reading and Writing Property List Data

A property list is a form of data representation that encapsulates several Foundation (and Core Foundation) data types, including dictionaries, arrays, strings, dates, binary data, and numerical and Boolean values. Property lists are commonly used to store structured configuration data. For example, the `Info.plist` file found in every Cocoa and iPhone applications is a property list that stores configuration information about the application itself. You can use property lists yourself to store additional information, such as the state of your application when it quits.

In code, you typically construct a property list starting with either a dictionary or array as a container object. You then add other property-list objects, including (possibly) other dictionaries and arrays. The keys of dictionaries must be string objects. The values for those keys are instances of `NSDictionary`, `NSArray`, `NSString`, `NSDate`, `NSData`, and `NSNumber`.

For an application whose data can be represented by a property-list object (such as an `NSDictionary` object), you could write that property list to disk using a method such as the one shown in Listing 6-2. This method serializes the property list object into an `NSData` object, then calls the `writeApplicationData:toFile:` method (the implementation for which is shown in Listing 6-4 (page 140)) to write that data to disk.

Listing 6-2 Converting a property-list object to an `NSData` object and writing it to storage

```
- (BOOL)writeApplicationPlist:(id)plist toFile:(NSString *)fileName {
    NSString *error;
    NSData *pData = [NSPropertyListSerialization dataFromPropertyList:plist
format:NSPropertyListBinaryFormat_v1_0 errorDescription:&error];
    if (!pData) {
        NSLog(@"%@", error);
        return NO;
    }
    return ([self writeApplicationData:pData toFile:(NSString *)fileName]);
}
```

When writing property list files in iPhone OS, it is important to store your files in binary format. You do this by specifying the `NSPropertyListBinaryFormat_v1_0` key in the *`format`* parameter of the `dataFromPropertyList:format:errorDescription:` method. The binary property-list format is much more compact than the other format options, which are text based. This compactness not only minimizes the amount of space taken up on the user's device, it also improves the time it takes to read and write the property list.

Listing 6-3 shows the corresponding code for loading a property-list file from disk and reconstituting the objects in that property list.

Listing 6-3 Reading a property-list object from the application's `Documents` directory

```
- (id)applicationPlistFromFile:(NSString *)fileName {
```

```
    NSData *retData;
    NSString *error;
    id retPlist;
    NSPropertyListFormat format;

    retData = [self applicationDataFromFile:fileName];
    if (!retData) {
        NSLog(@"Data file not returned.");
        return nil;
    }
    retPlist = [NSPropertyListSerialization propertyListFromData:retData
mutabilityOption:NSPropertyListImmutable format:&format errorDescription:&error];
    if (!retPlist){
        NSLog(@"Plist not returned, error: %@", error);
    }
    return retPlist;
}
```

For more on property lists and the `NSPropertyListSerialization` class, see *Property List Programming Guide*.

Using Archivers to Read and Write Data

An archiver converts an arbitrary collection of objects into a stream of bytes. Although this may sound similar to the process employed by the `NSPropertyListSerialization` class, there is an important difference. A property-list serializer can convert only a limited set of (mostly scalar) data types. Archivers can convert arbitrary Objective-C objects, scalar types, arrays, structures, strings, and more.

The key to the archiving process is in the target objects themselves. The objects manipulated by an archiver must conform to the `NSCoding` protocol, which defines the interface for reading and writing the object's state. When an archiver encodes a set of objects, it sends an `encodeWithCoder:` message to each one, which the object then uses to write out its critical state information to the corresponding archive. The unarchiving process reverses the flow of information. During unarchiving, each object receives an `initWithCoder:` message, which it uses to initialize itself with the state information currently in the archive. Upon completion of the unarchiving process, the stream of bytes is reconstituted into a new set of objects that have the same state as the ones written to the archive previously.

The Foundation framework supports two kinds of archivers—sequential and keyed. Keyed archivers are more flexible and are recommended for use in your application. The following example shows how to archive a graph of objects using a keyed archiver. The `representation` method of the `_myDataSource` object returns a single object (possibly an array or dictionary) that points to all of the objects to be included in the archive. The data object is then written to a file whose path is specified by the `myFilePath` variable.

```
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:[_myDataSource
representation]];
[data writeToFile:myFilePath atomically:YES];
```

Note: You could also send a `archiveRootObject:toFile:` message to the `NSKeyedArchiver` object to create the archive and write it to storage in one step.

To load the contents of an archive from disk, you simply reverse the process. After loading the data from disk, you use the `NSKeyedUnarchiver` class and its `unarchiveObjectWithData:` class method to get back the model-object graph. For example, to unarchive the data from the previous example, you could use the following code:

```
NSData* data = [NSData dataWithContentsOfFile:myFilePath];
id rootObject = [NSKeyedUnarchiver unarchiveObjectWithData:data];
```

For more information on how to use archivers and how to make your objects support the `NSCoding` protocol, see *Archives and Serializations Programming Guide for Cocoa*.

Writing Data to Your Documents Directory

After you have an `NSData` object encapsulating the application data (either as an archive or a serialized property list), you can call the method shown in Listing 6-4 to write that data to the application `Documents` directory.

Listing 6-4 Writing data to the application's `Documents` directory

```
- (BOOL)writeApplicationData:(NSData *)data toFile:(NSString *)fileName {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    if (!documentsDirectory) {
        NSLog(@"Documents directory not found!");
        return NO;
    }
    NSString *appFile = [documentsDirectory
stringByAppendingPathComponent:fileName];
    return ([data writeToFile:appFile atomically:YES]);
}
```

Reading Data from the Documents Directory

To read a file from your application's Documents directory, construct the path for the file name and use the desired method to read the file contents into memory. For relatively small files—that is, files less than a few memory pages in size—you could use the code in Listing 6-5 to obtain a data object for the file contents. This example constructs a full path to the file in the `Documents` directory, creates a data object from it, and then returns that object.

Listing 6-5 Reading data from the application's `Documents` directory

```
- (NSData *)applicationDataFromFile:(NSString *)fileName {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    NSString *appFile = [documentsDirectory
stringByAppendingPathComponent:fileName];
    NSData *myData = [[[NSData alloc] initWithContentsOfFile:appFile]
autorelease];
    return myData;
}
```

For files that would require multiple memory pages to hold in memory, you should avoid loading the entire file all at once. This is especially important if you plan to use only part of the file. For larger files, you should consider mapping the file into memory using either the `mmap` function or the `initWithContentsOfMappedFile:` method of `NSData`.

Choosing when to map files versus load them directly is up to you. It is relatively safe to load a file entirely into memory if it requires only a few (3-4) memory pages. If your file requires several dozen or a hundred pages, however, you would probably find it more efficient to map the file into memory. As with any such determination, though, you should measure your application's performance and determine how long it takes to load the file and allocate the necessary memory.

File Access Guidelines

When creating files or writing out file data, keep the following guidelines in mind:

- Minimize the amount of data you write to the disk. File operations are relatively slow and involve writing to the Flash disk, which has a limited lifespan. Some specific tips to help you minimize file-related operations include:
  - Write only the portions of the file that changed, but aggregate changes when you can. Avoid writing out the entire file just to change a few bytes.
  - When defining your file format, group frequently modified content together so as to minimize the overall number of blocks that need to be written to disk each time.
  - If your data consists of structured content that is randomly accessed, store it in a Core Data persistent store or a SQLite database. This is especially important if the amount of data you are manipulating could grow to be more than a few megabytes in size.
- Avoid writing cache files to disk. The only exception to this rule is when your application quits and you need to write state information that can be used to put your application back into the same state when it is next launched.

Saving State Information

When the user presses the Home button, iPhone OS quits your application and returns to the Home screen. Similarly, if your application opens a URI whose scheme is handled by a different application, iPhone OS quits your application and opens the URI in the other application. In other words, any action that would cause your application to suspend or go to the background in Mac OS X causes your application to quit in iPhone OS. Because these actions happen regularly on mobile devices, your application must change the way it manages volatile data and application state.

Unlike most desktop applications, where the user manually chooses when to save files to disk, your application should save changes automatically at key points in your workflow. Exactly when you save data is up to you, but there are two potential options. Either you can save each change immediately as the user makes it, or you can batch changes on the same page together and save them when the page is dismissed, a new page is displayed, or the application quits. Under no circumstances should you let the user navigate to a new page of content without saving the data on the previous page.

When your application is asked to quit, you should save the current state of your application to a temporary cache file or to the preferences database. The next time the user launches your application, use that information to restore your application to its previous state. The state information you save should be as minimal as possible but still let you accurately restore your application to an appropriate point. You do not have to display the exact same screen the user was manipulating previously if doing so would not make sense. For example, if a user edits a contact and then leaves the Phone application, upon returning, the Phone application displays the top-level list of contacts, rather than the editing screen for the contact.

Case Sensitivity

The file system for iPhone OS–based devices is case sensitive. Whenever you work with filenames, you should be sure that the case matches exactly or your code may be unable to open or access the file.

Networking

The networking stack in iPhone OS includes several interfaces over the radio hardware of iPhone and iPod touch devices. The main programming interface is the CFNetwork framework, which builds on top of BSD sockets and opaque types in the Core Foundation framework to communicate with network entities. You can also use the `NSStream` classes in the Foundation framework and the low-level BSD sockets found in the Core OS layer of the system.

The following sections provide iPhone-specific tips for developers who need to incorporate networking features into their applications. For information about how to use the CFNetwork framework for network communication, see *CFNetwork Programming Guide* and *CFNetwork Framework Reference*. For information about using the `NSStream` class, see *Foundation Framework Reference*.

Tips for Efficient Networking

When implementing code to receive or transmit across the network, remember that doing so is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving helps improve battery life. To that end, you should consider the following tips when writing your network-related code:

- For protocols you control, define your data formats to be as compact as possible.
- Avoid communicating using chatty protocols.
- Transmit data packets in bursts whenever you can.

The cellular and Wi-Fi radios are designed to power down when there is no activity. Doing so can take several seconds though, depending on the radio. If your application transmits small bursts of data every few seconds, the radios may stay powered up and continue to consume power, even when they are not actually doing anything. Rather than transmit small amounts of data more often, it is better to transmit a larger amount of data once or at relatively large intervals.

When communicating over the network, it is also important to remember that packets can be lost at any time. When writing your networking code, you should be sure to make it as robust as possible when it comes to failure handling. It is perfectly reasonable to implement handlers that respond to changes in network conditions, but do not be surprised if those handlers are not called consistently. For example, the Bonjour networking callbacks may not always be called immediately in response to the disappearance of a network service. The Bonjour system service does immediately invoke browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This might occur if the device providing the network service unexpectedly loses network connectivity or the notification is lost in transit.

Using Wi-Fi

If your application accesses the network using the Wi-Fi radios, you must notify the system of that fact by including the `UIRequiresPersistentWiFi` key in the application's `Info.plist` file. The inclusion of this key lets the system know that it should display the network selection panel if it detects any active Wi-Fi hot spots. It also lets the system know that it should not attempt to shut down the Wi-Fi hardware while your application is running.

To prevent the Wi-Fi hardware from using too much power, iPhone OS has a built-in timer that turns off the hardware completely after 30 minutes if no application has requested its use through the `UIRequiresPersistentWiFi` key. If the user launches an application that includes the key, iPhone OS effectively disables the timer for the duration of the application's life cycle. As soon as that application quits, however, the system reenables the timer.

Note: Note that even when `UIRequiresPersistentWiFi` has a value of `true`, it has no effect when the device is idle (that is, screen-locked). The application is considered inactive and, although it may function on some levels, it has no Wi-Fi connection.

For more information on the `UIRequiresPersistentWiFi` key and the keys of the `Info.plist` file, see "The Information Property List" (page 26).

The Airplane Mode Alert

If the device is in airplane mode when your application launches, the system may display a panel to notify the user of that fact. The system displays this panel only when all of the following conditions are met:

- Your application's information property list (`Info.plist`) file contains the `UIRequiresPersistentWiFi` key and the value of that key is set to true.
- Your application launches while the device is currently in airplane mode.
- Wi-Fi on the device has not been manually reenabled after the switch to airplane mode.

Multimedia Support

Whether multimedia features are central or incidental to your application, iPhone users expect high quality. When presenting video content, take advantage of the device's high-resolution screen and high frame rates. When designing the audio portion of your application, keep in mind that compelling sound adds immeasurably to a user's overall experience.

You can take advantage of the iPhone OS multimedia frameworks for adding features like:

- High-quality audio recording, playback, and streaming
- Immersive game sounds
- Live voice chat
- Playback of content from a user's iPod library
- Video playback and recording on supported devices

Using Sound in iPhone OS

iPhone OS offers a rich set of tools for working with sound in your application. These tools are arranged into frameworks according to the features they provide, as follows:

- Use the **Media Player framework** to play songs, audio books, or audio podcasts from a user's iPod library. For details, see *Media Player Framework Reference*, *iPod Library Access Programming Guide*, and the *AddMusic* sample code project.
- Use the **AV Foundation framework** to play and record audio using a simple Objective-C interface. For details, see *AV Foundation Framework Reference* and the *avTouch* sample code project.
- Use the **Audio Toolbox framework** to play audio with synchronization capabilities, access packets of incoming audio, parse audio streams, convert audio formats, and record audio with access to individual packets. For details, see *Audio Toolbox Framework Reference* and the *SpeakHere* sample code project.
- Use the **Audio Unit framework** to connect to and use audio processing plug-ins. For details, see *Audio Unit Framework Reference*, *System Audio Unit Access Guide*, and the *aurioTouch* and *iPhoneMultichannelMixerTest* sample code projects.
- Use the **OpenAL framework** to provide positional audio playback in games and other applications. iPhone OS supports OpenAL 1.1. For information on OpenAL, see the OpenAL website, *OpenAL FAQ for iPhone OS*, and the *oalTouch* sample code project.

To allow your code to use the features of an audio framework, add that framework to your Xcode project, link against it in any relevant targets, and add an appropriate `#import` statement near the top of relevant source files. For example, to provide access to the AV Foundation framework in a source file, add a `#import <AVFoundation/AVFoundation.h>` statement near the top of the file. For detailed information on how to add frameworks to your project, see Files in Projects in *Xcode Project Management Guide*.

Important: To use the features of the Audio Unit framework, add the *Audio Toolbox* framework to your Xcode project and link against it in any relevant targets. Then add a `#import <AudioUnit/AudioUnit.h>` statement near the top of relevant source files.

This section on sound provides a quick introduction to implementing iPhone OS audio features, as listed here:

- To play songs, audio podcasts, and audio books from a user's iPod library, see "Playing Media Items with iPod Library Access" (page 150).
- To play and record audio in the fewest lines of code, use the AV Foundation framework. See "Playing Sounds Easily with the AVAudioPlayer Class" (page 153) and "Recording with the AVAudioRecorder Class" (page 158).
- To provide full-featured audio playback including stereo positioning, level control, and simultaneous sounds, use OpenAL. See "Playing Sounds with Positioning Using OpenAL" (page 157).
- To provide lowest latency audio, especially when doing simultaneous input and output (such as for a VoIP application), use the I/O unit or the Voice Processing I/O unit. See "Audio Unit Support in iPhone OS" (page 161).
- To play sounds with the highest degree of control, including support for synchronization, use Audio Queue Services. See "Playing Sounds with Control Using Audio Queue Services" (page 155). Audio Queue Services also supports recording and provides access to incoming audio packets, as described in "Recording with Audio Queue Services" (page 160).
- To parse audio streamed from a network connection, use Audio File Stream Services. See "Parsing Streamed Audio" (page 160).
- To play user-interface sound effects, or to invoke vibration on devices that provide that feature, use System Sound Services. See "Playing UI Sound Effects or Invoking Vibration Using System Sound Services" (page 151).

Be sure to read the next section, "The Basics: Audio Codecs, Supported Audio Formats, and Audio Sessions" (page 146), for critical information on how audio works on iPhone. Also read "Best Practices for iPhone Audio" (page 161), which offers guidelines and lists the audio and file formats to use for best performance and best user experience.

When you're ready to dig deeper, the iPhone Dev Center contains guides, reference books, sample code, and more. For tips on how to perform common audio tasks, see *Audio & Video Coding How-To's*. For in-depth explanations of audio development in iPhone OS, see *Core Audio Overview*, *Audio Session Programming Guide*, *Audio Queue Services Programming Guide*, *System Audio Unit Access Guide*, and *iPod Library Access Programming Guide*.

The Basics: Audio Codecs, Supported Audio Formats, and Audio Sessions

To get oriented toward iPhone audio development, it's important to understand a few critical things about the hardware and software architecture of iPhone OS devices—described in this section.

iPhone Hardware and Software Audio Codecs

To ensure optimum performance and quality, you need to pick the right audio format and audio codec type. Starting in iPhone OS 3.0, most audio formats can use software-based encoding (for recording) and decoding (for playback). Software codecs support simultaneous playback of multiple sounds, but may entail significant CPU overhead.

Hardware-assisted decoding provides excellent performance—but does not support simultaneous playback of multiple sounds. If you need to maximize video frame rate in your application, minimize the CPU impact of your audio playback by using uncompressed audio or the IMA4 format, or use hardware-assisted decoding of your compressed audio assets.

For best-practice advice on picking an audio format, see "Preferred Audio Formats in iPhone OS" (page 162).

Table 7-1 describes the playback audio codecs available on iPhone OS devices.

Table 7-1 Audio playback formats and codecs

Audio decoder/playback format	Hardware-assisted decoding	Software-based decoding
AAC (MPEG-4 Advanced Audio Coding)	Yes	Yes, starting in iPhone OS 3.0
ALAC (Apple Lossless)	Yes	Yes, starting in iPhone OS 3.0
AMR (Adaptive Multi-Rate, a format for speech)	-	Yes
HE-AAC (MPEG-4 High Efficiency AAC)	Yes	-
iLBC (internet Low Bitrate Codec, another format for speech)	-	Yes
IMA4 (IMA/ADPCM)	-	Yes
Linear PCM (uncompressed, linear pulse-code modulation)	-	Yes
MP3 (MPEG-1 audio layer 3)	Yes	Yes, starting in iPhone OS 3.0
µ-law and a-law	-	Yes

When using hardware-assisted decoding, the device can play *only a single instance* of one of the supported formats at a time. For example, if you are playing a stereo MP3 sound using the hardware codec, a second simultaneous MP3 sound will use software decoding. Similarly, you cannot simultaneously play an AAC and an ALAC sound using hardware. If the iPod application is playing an AAC or MP3 sound in the background, it has claimed the hardware codec; your application then plays AAC, ALAC, and MP3 audio using software decoding.

To play multiple sounds with best performance, or to efficiently play sounds while the iPod is playing in the background, use linear PCM (uncompressed) or IMA4 (compressed) audio.

To learn how to check at runtime which hardware and software codecs are available on a device, read the discussion for the `kAudioFormatProperty_HardwareCodecCapabilities` constant in *Audio Format Services Reference* and read Technical Q&A QA1663, "Determining the availability of the AAC hardware encoder at runtime."

To summarize how iPhone OS supports audio formats for single or multiple playback:

- **Linear PCM and IMA4 (IMA/ADPCM)** You can play multiple linear PCM or IMA4 sounds simultaneously in iPhone OS without incurring CPU resource problems. The same is true for the AMR and iLBC speech-quality formats, and for the µ-law and a-law compressed formats. When using compressed formats, check the sound quality to ensure it meets your needs.
- **AAC, HE-AAC, MP3, and ALAC (Apple Lossless)** Playback for AAC, HE-AAC, MP3, and ALAC sounds can use efficient hardware-assisted decoding on iPhone OS devices, but these codecs all share a single hardware path. The device can play only a single instance of one of these formats at a time using hardware-assisted decoding.

The single hardware path for AAC, HE-AAC, MP3, and ALAC playback has implications for "play along" style applications, such as a virtual piano. If the user is playing a song in one of these three formats in the iPod application, then your application—to play along over that audio—will employ software decoding.

Table 7-2 describes the recording audio codecs available on iPhone OS devices.

Table 7-2 Audio recording formats and codecs

Audio encoder/recording format	Hardware-assisted encoding	Software-based encoding
AAC (MPEG-4 Advanced Audio Coding)	Yes, starting with iPhone 3GS and with iPod touch (2nd generation)	Yes
ALAC (Apple Lossless)	-	Yes
iLBC (internet Low Bitrate Codec, for speech)	-	Yes
IMA4 (IMA/ADPCM)	-	Yes
Linear PCM (uncompressed, linear pulse-code modulation)	-	Yes
µ-law and a-law	-	Yes

Audio Sessions

The iPhone OS audio session APIs let you define your application's general audio behavior and design it to work well within the larger audio context of the device it's running on. These APIs are described in *Audio Session Services Reference* and *AVAudioSession Class Reference*. Using these APIs, you can specify such behaviors as:

- Whether or not your audio should be silenced by the Ring/Silent switch
- Whether or not your audio should stop upon screen lock
- Whether other audio, such as from the iPod, should continue playing or be silenced when your audio starts

The audio session APIs also let you respond to user actions, such as the plugging in or unplugging of headsets, and to events that use the device's sound hardware, such as Clock and Calendar alarms and incoming phone calls.

The audio session APIs provide three programmatic features, described in Table 7-3.

Table 7-3 Features provided by the audio session APIs

Audio session feature	Description
Setting categories	A category is a key that identifies a set of audio behaviors for your application. By setting a category, you indicate your audio intentions to iPhone OS, such as whether your audio should continue when the screen locks. There are six categories, described in Audio Session Categories. You can fine-tune the behavior of some categories, as explained in "Fine-Tuning the Category" in *Audio Session Programming Guide*.
Handling interruptions and route changes	Your audio session posts messages when your audio is interrupted, when an interruption ends, and when the hardware audio route changes. These messages let you respond gracefully to changes in the larger audio environment—such as an interruption due to in an incoming phone call. For details, see Handling Audio Hardware Route Changes and Handling Audio Interruptions.
Optimizing for hardware characteristics	You can query the audio session to discover characteristics of the device your application is running on, such as hardware sample rate, number of hardware channels, and whether audio input is available. For details, see Optimizing for Device Hardware.

There are two interfaces for working with the audio session:

- A streamlined, objective-C interface that gives you access to the core audio session features and is described in *AVAudioSession Class Reference* and *AVAudioSessionDelegate Protocol Reference*.
- A C-based interface that provides comprehensive access to all basic and advanced audio session features and is described in *Audio Session Services Reference*.

You can mix and match audio session code from AV Foundation and Audio Session Services—the interfaces are completely compatible.

An audio session comes with some default behavior that you can use to get started in development. However, except for certain special cases, the default behavior is unsuitable for a shipping application that uses audio.

For example, when using the default audio session, audio in your application stops when the Auto-Lock period times out and the screen locks. If you want to ensure that playback continues with the screen locked, include the following lines in your application's initialization code:

```
NSError *setCategoryErr = nil;
NSError *activationErr  = nil;
[[AVAudioSession sharedInstance]
              setCategory: AVAudioSessionCategoryPlayback
                    error: &setCategoryErr];
[[AVAudioSession sharedInstance]
                setActive: YES
                    error: &activationErr];
```

The `AVAudioSessionCategoryPlayback` category ensures that playback continues when the screen locks. Activating the audio session puts the specified category into effect.

How you handle the interruption caused by an incoming phone call or Clock or Calendar alarm depends on the audio technology you are using, as shown in Table 7-4.

Table 7-4 Handling audio interruptions

Audio technology	How interruptions work
AV Foundation framework	The `AVAudioPlayer` and `AVAudioRecorder` classes provide delegate methods for interruption start and end. Implement these methods to update your user interface and optionally, after interruption ends, to resume paused playback. The system automatically pauses playback or recording upon interruption, and reactivates your audio session when you resume playback or recording. If you want to save and restore playback position between application launches, save playback position on interruption as well as on application quit.
Audio Queue Services, I/O audio unit	These technologies put your application in control of handling interruptions. You are responsible for saving playback or recording position and reactivating your audio session after interruption ends. Implement the `AVAudioSession` interruption delegate methods or write an interruption listener callback function.
OpenAL	When using OpenAL for playback, implement the `AVAudioSession` interruption delegate methods or write an interruption listener callback function—as when using Audio Queue Services. However, the delegate or callback must additionally manage the OpenAL context.
System Sound Services	Sounds played using System Sound Services go silent when an interruption starts. They can automatically be used again if the interruption ends. Applications cannot influence the interruption behavior for sounds that use this playback technology.

Every iPhone OS application—with rare exception—should actively manage its audio session. For a complete explanation of how to do this, read *Audio Session Programming Guide*. To ensure that your application conforms to Apple recommendations for audio session behavior, read "Using Sound" in *iPhone Human Interface Guidelines*.

Playing Audio

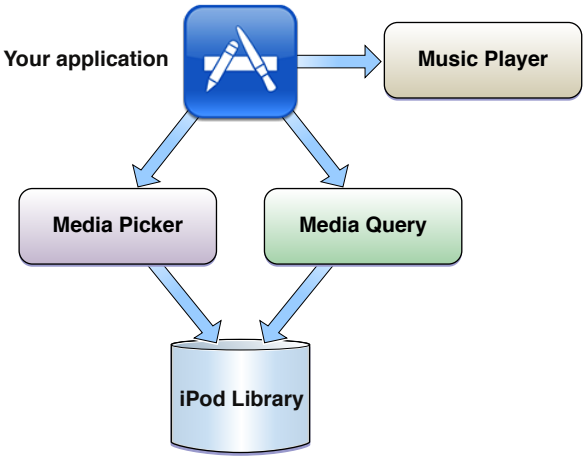
This section introduces you to playing sounds in iPhone OS using iPod library access, System Sound Services, Audio Queue Services, the AV Foundation framework, and OpenAL.

Playing Media Items with iPod Library Access

Starting in iPhone OS 3.0, iPod library access lets your application play a user's songs, audio books, and audio podcasts. The API design makes basic playback very simple while also supporting advanced searching and playback control.

As shown in Figure 7-1, your application has two ways to retrieve media items. The media item picker, shown on the left, is an easy-to-use, pre-packaged view controller that behaves like the built-in iPod application's music selection interface. For many applications, this is sufficient. If the picker doesn't provide the specialized access control you want, the media query interface will. It supports predicate-based specification of items from the iPod library.

Figure 7-1 Using iPod library access



As depicted in the figure to the right of your application, you then play the retrieved media items using the music player provided by this API.

For a complete explanation of how to add media item playback to your application, see *iPod Library Access Programming Guide*. For a code example, see the *AddMusic* sample code project.

Playing UI Sound Effects or Invoking Vibration Using System Sound Services

To play user-interface sound effects (such as button clicks), or to invoke vibration on devices that support it, use System Sound Services. This compact interface is described in *System Sound Services Reference*. You can find sample code in the *SysSound* sample in the iPhone Dev Center.

Note: Sounds played with System Sound Services are not subject to configuration using your audio session. As a result, you cannot keep the behavior of System Sound Services audio in line with other audio behavior in your application. This is the most important reason to avoid using System Sound Services for any audio apart from its intended uses.

The `AudioServicesPlaySystemSound` function lets you very simply play short sound files. The simplicity carries with it a few restrictions. Your sound files must be:

- No longer than 30 seconds in duration
- In linear PCM or IMA4 (IMA/ADPCM) format
- Packaged in a `.caf`, `.aif`, or `.wav` file

In addition, when you use the `AudioServicesPlaySystemSound` function:

- Sounds play at the current system audio volume, with no programmatic volume control available
- Sounds play immediately
- Looping and stereo positioning are unavailable

The similar `AudioServicesPlayAlertSound` function plays a short sound as an alert. If a user has configured their device to vibrate in Ring Settings, calling this function invokes vibration in addition to playing the sound file.

Note: System-supplied alert sounds and system-supplied user-interface sound effects are not available to your application. For example, using the `kSystemSoundID_UserPreferredAlert` constant as a parameter to the `AudioServicesPlayAlertSound` function will not play anything.

To play a sound with the `AudioServicesPlaySystemSound` or `AudioServicesPlayAlertSound` function, first create a sound ID object, as shown in Listing 7-1.

Listing 7-1 Creating a sound ID object

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

// Get the URL to the sound file to play. The file in this case
// is "tap.aiff"
soundFileURLRef  = CFBundleCopyResourceURL (
                        mainBundle,
                        CFSTR ("tap"),
                        CFSTR ("aif"),
                        NULL
                    );

// Create a system sound object representing the sound file
AudioServicesCreateSystemSoundID (
    soundFileURLRef,
    &soundFileObject
);
```

Then play the sound, as shown in Listing 7-2.

Listing 7-2 Playing a system sound

```
- (IBAction) playSystemSound {
    AudioServicesPlaySystemSound (self.soundFileObject);
}
```

In typical use, which includes playing a sound occasionally or repeatedly, retain the sound ID object until your application quits. If you know that you will use a sound only once—for example, in the case of a startup sound—you can destroy the sound ID object immediately after playing the sound, freeing memory.

Applications running on iPhone OS devices that support vibration can trigger that feature using System Sound Services. You specify the vibrate option with the `kSystemSoundID_Vibrate` identifier. To trigger it, use the `AudioServicesPlaySystemSound` function, as shown in Listing 7-3.

Listing 7-3 Triggering vibration

```
#import <AudioToolbox/AudioToolbox.h>
#import <UIKit/UIKit.h>
- (void) vibratePhone {
    AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
}
```

If your application is running on an iPod touch, this code does nothing.

Playing Sounds Easily with the AVAudioPlayer Class

The `AVAudioPlayer` class provides a simple Objective-C interface for playing sounds. If your application does not require stereo positioning or precise synchronization, and if you are not playing audio captured from a network stream, Apple recommends that you use this class for playback.

Using an audio player you can:

- Play sounds of any duration
- Play sounds from files or memory buffers
- Loop sounds
- Play multiple sounds simultaneously (although not with precise synchronization)
- Control relative playback level for each sound you are playing
- Seek to a particular point in a sound file, which supports application features such as fast forward and rewind
- Obtain audio power data that you can use for audio level metering

The `AVAudioPlayer` class lets you play sound in any audio format available in iPhone OS, as described in Table 7-1 (page 147). For a complete description of this class's interface, see *AVAudioPlayer Class Reference*.

To configure an audio player:

1. Assign a sound file to the audio player.
2. Prepare the audio player for playback, which acquires the hardware resources it needs.
3. Designate an audio player delegate object, which handles interruptions as well as the playback-completed event.

The code in Listing 7-4 illustrates these steps. It would typically go into an initialization method of the controller class for your application. (In production code, you'd include appropriate error handling.)

Listing 7-4 Configuring an AVAudioPlayer object

```
// in the corresponding .h file:
// @property (nonatomic, retain) AVAudioPlayer *player;

// in the .m file:
@synthesize player; // the player object

NSString *soundFilePath =
```

```
            [[NSBundle mainBundle] pathForResource: @"sound"
                                            ofType: @"wav"];

NSURL *fileURL = [[NSURL alloc] initFileURLWithPath: soundFilePath];

AVAudioPlayer *newPlayer =
            [[AVAudioPlayer alloc] initWithContentsOfURL: fileURL
                                                   error: nil];
[fileURL release];

self.player = newPlayer;
[newPlayer release];

[player prepareToPlay];
[player setDelegate: self];
```

The delegate (which can be your controller object) handle interruptions and updates the user interface when a sound has finished playing. The delegate methods for the `AVAudioPlayer` class are described in *AVAudioPlayerDelegate Protocol Reference*. Listing 7-5 shows a simple implementation of one delegate method. This code updates the title of a Play/Pause toggle button when a sound has finished playing.

Listing 7-5 Implementing an AVAudioPlayer delegate method

```
- (void) audioPlayerDidFinishPlaying: (AVAudioPlayer *) player
                        successfully: (BOOL) completed {
    if (completed == YES) {
        [self.button setTitle: @"Play" forState: UIControlStateNormal];
    }
}
```

To play, pause, or stop an `AVAudioPlayer` object, call one of its playback control methods. You can test whether or not playback is in progress by using the `playing` property. Listing 7-6 shows a basic play/pause toggle method that controls playback and updates the title of a `UIButton` object.

Listing 7-6 Controlling an AVAudioPlayer object

```
- (IBAction) playOrPause: (id) sender {

    // if already playing, then pause
    if (self.player.playing) {
        [self.button setTitle: @"Play" forState: UIControlStateHighlighted];
        [self.button setTitle: @"Play" forState: UIControlStateNormal];
        [self.player pause];

    // if stopped or paused, start playing
    } else {
        [self.button setTitle: @"Pause" forState: UIControlStateHighlighted];
        [self.button setTitle: @"Pause" forState: UIControlStateNormal];
        [self.player play];
    }
}
```

The `AVAudioPlayer` class uses the Objective-C declared properties feature for managing information about a sound—such as the playback point within the sound's timeline, and for accessing playback options—such as volume and looping. For example, you can set the playback volume for an audio player as shown here:

```
[self.player setVolume: 1.0];    // available range is 0.0 through 1.0
```

For more information on the `AVAudioPlayer` class, see *AVAudioPlayer Class Reference*.

Playing Sounds with Control Using Audio Queue Services

Audio Queue Services adds playback capabilities beyond those available with the `AVAudioPlayer` class. Using Audio Queue Services for playback lets you:

- Precisely schedule when a sound plays, allowing synchronization
- Precisely control volume on a buffer-by-buffer basis
- Play audio that you have captured from a stream using Audio File Stream Services

Audio Queue Services lets you play sound in any audio format available in iPhone OS, as described in Table 7-1 (page 147). You can also use this technology for recording, as explained in "Recording Audio" (page 158).

For detailed information on using this technology, see *Audio Queue Services Programming Guide* and *Audio Queue Services Reference*. For sample code, see the *SpeakHere* sample.

Creating an Audio Queue Object

To create an audio queue object for playback, perform these three steps:

1. Create a data structure to manage information needed by the audio queue, such as the audio format for the data you want to play.
2. Define a callback function for managing audio queue buffers. The callback uses Audio File Services to read the file you want to play. (In iPhone OS 2.1 and later, you can also use Extended Audio File Services to read the file.)
3. Instantiate the playback audio queue using the `AudioQueueNewOutput` function.

Listing 7-7 illustrates these steps using ANSI C. (In production code, you'd include appropriate error handling.) The *SpeakHere* sample project shows these same steps in the context of a C++ program.

Listing 7-7 Creating an audio queue object

```
static const int kNumberBuffers = 3;
// Create a data structure to manage information needed by the audio queue
struct myAQStruct {
    AudioFileID                   mAudioFile;
    CAStreamBasicDescription      mDataFormat;
    AudioQueueRef                 mQueue;
    AudioQueueBufferRef           mBuffers[kNumberBuffers];
    SInt64                        mCurrentPacket;
    UInt32                        mNumPacketsToRead;
    AudioStreamPacketDescription  *mPacketDescs;
    bool                          mDone;
};
// Define a playback audio queue callback function
static void AQTestBufferCallback(
    void                  *inUserData,
    AudioQueueRef         inAQ,
    AudioQueueBufferRef   inCompleteAQBuffer
) {
```

```
    myAQStruct *myInfo = (myAQStruct *)inUserData;
    if (myInfo->mDone) return;
    UInt32 numBytes;
    UInt32 nPackets = myInfo->mNumPacketsToRead;

    AudioFileReadPackets (
        myInfo->mAudioFile,
        false,
        &numBytes,
        myInfo->mPacketDescs,
        myInfo->mCurrentPacket,
        &nPackets,
        inCompleteAQBuffer->mAudioData
    );
    if (nPackets > 0) {
        inCompleteAQBuffer->mAudioDataByteSize = numBytes;
        AudioQueueEnqueueBuffer (
            inAQ,
            inCompleteAQBuffer,
            (myInfo->mPacketDescs ? nPackets : 0),
            myInfo->mPacketDescs
        );
        myInfo->mCurrentPacket += nPackets;
    } else {
        AudioQueueStop (
            myInfo->mQueue,
            false
        );
        myInfo->mDone = true;
    }
}
// Instantiate an audio queue object
AudioQueueNewOutput (
    &myInfo.mDataFormat,
    AQTestBufferCallback,
    &myInfo,
    CFRunLoopGetCurrent(),
    kCFRunLoopCommonModes,
    0,
    &myInfo.mQueue
);
```

Controlling the Playback Level

Audio queue objects give you two ways to control playback level.

To set playback level directly, use the `AudioQueueSetParameter` function with the `kAudioQueueParam_Volume` parameter, as shown in Listing 7-8. Level change takes effect immediately.

Listing 7-8 Setting the playback level directly

```
Float32 volume = 1;    // linear scale, range from 0.0 through 1.0
AudioQueueSetParameter (
    myAQstruct.audioQueueObject,
    kAudioQueueParam_Volume,
    volume
);
```

You can also set playback level for an audio queue buffer by using the `AudioQueueEnqueueBufferWithParameters` function. This lets you assign audio queue settings that are, in effect, carried by an audio queue buffer as you enqueue it. Such changes take effect when the buffer begins playing.

In both cases, level changes for an audio queue remain in effect until you change them again.

Indicating Playback Level

You can obtain the current playback level from an audio queue object by:

1. Enabling metering for the audio queue object by setting its `kAudioQueueProperty_EnableLevelMetering` property to `true`
2. Querying the audio queue object's `kAudioQueueProperty_CurrentLevelMeter` property

The value of this property is an array of `AudioQueueLevelMeterState` structures, one per channel. Listing 7-9 shows this structure:

Listing 7-9 The `AudioQueueLevelMeterState` structure

```
typedef struct AudioQueueLevelMeterState {
    Float32     mAveragePower;
    Float32     mPeakPower;
};  AudioQueueLevelMeterState;
```

Playing Multiple Sounds Simultaneously

To play multiple sounds simultaneously, create one playback audio queue object for each sound. For each audio queue, schedule the first buffer of audio to start at the same time using the `AudioQueueEnqueueBufferWithParameters` function.

Starting in iPhone OS 3.0, nearly all supported audio formats can be used for simultaneous playback—namely, all those that can be played using software decoding, as described in Table 7-1 (page 147). For the most processor-efficient multiple playback, use linear PCM (uncompressed) or IMA4 (compressed) audio.

Playing Sounds with Positioning Using OpenAL

The open-sourced OpenAL audio API, available in iPhone OS in the OpenAL framework, provides an interface optimized for positioning sounds in a stereo field during playback. Playing, positioning, and moving sounds works just as it does on other platforms. OpenAL also lets you mix sounds. OpenAL uses the I/O unit for playback, resulting in the lowest latency.

For all of these reasons, OpenAL is your best choice for playing sounds in game applications on iPhone OS–based devices. However, OpenAL is also a good choice for general iPhone OS application audio playback needs.

OpenAL 1.1 support in iPhone OS is built on top of Core Audio. For more information, see *OpenAL FAQ for iPhone OS*. For OpenAL documentation, see the OpenAL website at http://openal.org. For sample code, see *oalTouch*.

Recording Audio

iPhone OS supports audio recording using the `AVAudioRecorder` class and Audio Queue Services. These interfaces do the work of connecting to the audio hardware, managing memory, and employing codecs as needed. You can record audio in any of the formats listed in Table 7-2 (page 148).

Recording takes place at a system-defined input level in iPhone OS. The system takes input from the audio source that the user has chosen—the built-in microphone or, if connected, the headset microphone or other input source.

Recording with the AVAudioRecorder Class

The easiest way to record sound in iPhone OS is with the `AVAudioRecorder` class, described in *AVAudioRecorder Class Reference*. This class provides a highly-streamlined, Objective-C interface that makes it easy to provide sophisticated features like pausing/resuming recording and handling audio interruptions. At the same time, you retain complete control over recording format.

To prepare for recording using an audio recorder:

1. Specify a sound file URL.
2. Set up the audio session.
3. Configure the audio recorder's initial state.

Application launch is a good time to do this part of the setup, as shown in Listing 7-10. Variables such as `soundFileURL` and `recording` in this example are declared in the class interface. (In production code, you would include appropriate error handling.)

Listing 7-10 Setting up the audio session and the sound file URL

```
- (void) viewDidLoad {

    [super viewDidLoad];

    NSString *tempDir = NSTemporaryDirectory ();
    NSString *soundFilePath =
                [tempDir stringByAppendingString: @"sound.caf"];

    NSURL *newURL = [[NSURL alloc] initFileURLWithPath: soundFilePath];
    self.soundFileURL = newURL;
    [newURL release];

    AVAudioSession *audioSession = [AVAudioSession sharedInstance];
    audioSession.delegate = self;
    [audioSession setActive: YES error: nil];

    recording = NO;
    playing = NO;
}
```

To handle interruptions and the completion of recording, add the `AVAudioSessionDelegate` and `AVAudioRecorderDelegate` protocol names to the interface declaration for your implementation. If your application also does playback, also adopt the `AVAudioPlayerDelegate Protocol Reference` protocol.

To implement a record method, you can use code such as that shown in Listing 7-11. (In production code, you would include appropriate error handling.)

Listing 7-11 A record/stop method using the AVAudioRecorder class

```
- (IBAction) recordOrStop: (id) sender {

   if (recording) {

      [soundRecorder stop];
      recording = NO;
      self.soundRecorder = nil;

      [recordOrStopButton setTitle: @"Record" forState:
                                       UIControlStateNormal];
      [recordOrStopButton setTitle: @"Record" forState:
                                       UIControlStateHighlighted];
      [[AVAudioSession sharedInstance] setActive: NO error: nil];

   } else {

      [[AVAudioSession sharedInstance]
                           setCategory: AVAudioSessionCategoryRecord
                                 error: nil];

      NSDictionary *recordSettings =
         [[NSDictionary alloc] initWithObjectsAndKeys:
           [NSNumber numberWithFloat: 44100.0], AVSampleRateKey,
            [NSNumber numberWithInt: kAudioFormatAppleLossless], AVFormatIDKey,
             [NSNumber numberWithInt: 1], AVNumberOfChannelsKey,
             [NSNumber numberWithInt: AVAudioQualityMax],
                                       AVEncoderAudioQualityKey,
             nil];

      AVAudioRecorder *newRecorder =
            [[AVAudioRecorder alloc] initWithURL: soundFileURL
                                       settings: recordSettings
                                          error: nil];
      [recordSettings release];
      self.soundRecorder = newRecorder;
      [newRecorder release];

      soundRecorder.delegate = self;
      [soundRecorder prepareToRecord];
      [soundRecorder record];
      [recordOrStopButton setTitle: @"Stop" forState: UIControlStateNormal];
     [recordOrStopButton setTitle: @"Stop" forState: UIControlStateHighlighted];

      recording = YES;
   }
}
```

For more information on the `AVAudioRecorder` class, see *AVAudioRecorder Class Reference*.

Recording with Audio Queue Services

To set up for recording with audio with Audio Queue Services, your application instantiates a recording audio queue object and provides a callback function. The callback stores incoming audio data in memory for immediate use or writes it to a file for long-term storage.

Just as with playback, you can obtain the current recording audio level from an audio queue object by querying its `kAudioQueueProperty_CurrentLevelMeter` property, as described in "Indicating Playback Level" (page 157).

For detailed examples of how to use Audio Queue Services to record audio, see Recording Audio in *Audio Queue Services Programming Guide*. For sample code, see the *SpeakHere* sample.

Parsing Streamed Audio

To play streamed audio content, such as from a network connection, use Audio File Stream Services in concert with Audio Queue Services. Audio File Stream Services parses audio packets and metadata from common audio file container formats in a network bitstream. You can also use it to parse packets and metadata from on-disk files.

In iPhone OS, you can parse the same audio file and bitstream formats that you can in Mac OS X, as follows:

- MPEG-1 Audio Layer 3, used for .mp3 files
- MPEG-2 ADTS, used for the .aac audio data format
- AIFC
- AIFF
- CAF
- MPEG-4, used for .m4a, .mp4, and .3gp files
- NeXT
- WAVE

Having retrieved audio packets, you can play back the recovered sound in any of the formats supported in iPhone OS, as listed in Table 7-1 (page 147).

For best performance, network streaming applications should use data from Wi-Fi connections. iPhone OS lets you determine which networks are reachable and available through its System Configuration framework and its `SCNetworkReachabilityRef` opaque type, described in *SCNetworkReachability Reference*. For sample code, see the *Reachability* sample in the iPhone Dev Center.

To connect to a network stream, use interfaces from Core Foundation, such as the one described in *CFHTTPMessage Reference*. Parse the network packets to recover audio packets using Audio File Stream Services. Then buffer the audio packets and send them to a playback audio queue object.

Audio File Stream Services relies on interfaces from Audio File Services, such as the `AudioFramePacketTranslation` structure and the `AudioFilePacketTableInfo` structure. These are described in *Audio File Services Reference*.

For more information on using streams, refer to *Audio File Stream Services Reference*.

Audio Unit Support in iPhone OS

iPhone OS provides a set of audio processing plug-ins, known as *audio units*, that you can use in any application. The interfaces in the Audio Unit framework let you open, connect, and use these audio units.

To use the features of the Audio Unit framework, add the *Audio Toolbox* framework to your Xcode project and link against it in any relevant targets. Then add a `#import <AudioUnit/AudioUnit.h>` statement near the top of relevant source files. For detailed information on how to add frameworks to your project, see Files in Projects in *Xcode Project Management Guide*.

Table 7-5 lists the audio units provided in iPhone OS.

Table 7-5 System-supplied audio units

Audio unit	Description
Converter unit	The Converter unit, of type `kAudioUnitSubType_AUConverter`, lets you convert audio data from one format to another.
iPod Equalizer unit	The iPod EQ unit, of type `kAudioUnitSubType_AUiPodEQ`, provides a simple, preset-based equalizer you can use in your application.
3D Mixer unit	The 3D Mixer unit, of type `kAudioUnitSubType_AU3DMixerEmbedded`, lets you mix multiple audio streams, specify stereo output panning, manipulate sample rate, and more.
Multichannel Mixer unit	The Multichannel Mixer unit, of type `kAudioUnitSubType_MultiChannelMixer`, lets you mix multiple audio streams to a single stream.
Generic Output unit	The Generic Output unit, of type `kAudioUnitSubType_GenericOutput`, supports converting to and from linear PCM format; can be used to start and stop a graph.
I/O unit	The I/O unit, of type `kAudioUnitSubType_RemoteIO`, lets you connect to audio input and output hardware and supports realtime I/O. For sample code showing you how to use this audio unit, see *aurioTouch*.
Voice Processing I/O unit	The Voice Processing I/O unit, of type `kAudioUnitSubType_VoiceProcessingIO`, has the characteristics of the I/O unit and adds echo suppression for two-way communication.

For more information on using system audio units, see *System Audio Unit Access Guide*. For reference documentation, see *Audio Unit Framework Reference*. The iPhone Dev Center provides two sample-code projects that demonstrate use of system audio units: *aurioTouch* and *iPhoneMultichannelMixerTest*.

Best Practices for iPhone Audio

This section lists some important tips for using audio in iPhone OS and describes the best audio data formats for various uses.

Tips for Using Audio

Table 7-6 lists some important tips to keep in mind when using audio in iPhone OS.

Table 7-6 Audio tips

Tip	Action
Use compressed audio appropriately	For AAC, MP3, and ALAC (Apple Lossless) audio, decoding can take place using hardware-assisted codecs. While efficient, this is limited to one audio stream at a time. If you need to play multiple sounds simultaneously, store those sounds using the IMA4 (compressed) or linear PCM (uncompressed) format.
Convert to the data format and file format you need	The `afconvert` tool in Mac OS X lets you convert to a wide range of audio data formats and file types. See "Preferred Audio Formats in iPhone OS" (page 162) and the `afconvert` man page.
Evaluate audio memory issues	When playing sound with Audio Queue Services, you write a callback that sends short segments of audio data to audio queue buffers. In some cases, loading an entire sound file to memory for playback, which minimizes disk access, is best. In other cases, loading just enough data at a time to keep the buffers full is best. Test and evaluate which strategy works best for your application.
Reduce audio file sizes by limiting sample rates, bit depths, and channels	Sample rate and the number of bits per sample have a direct impact on the size of your audio files. If you need to play many such sounds, or long-duration sounds, consider reducing these values to reduce the memory footprint of the audio data. For example, rather than use 44.1 kHz sampling rate for sound effects, you could use a 32 kHz (or possibly lower) sample rate and still provide reasonable quality. Using monophonic (single-channel) audio instead of stereo (two channel) reduces file size. For each sound asset, consider whether mono could suit your needs.
Pick the appropriate technology	Use OpenAL when you want a convenient, high-level interface for positioning sounds in a stereo field or when you need low latency playback. To parse audio packets from a file or a network stream, use Audio File Stream Services. For simple playback of single or multiple sounds, use the `AVAudioPlayer` class. For recording to a file, use the `AVAudioRecorder` class. For audio chat, use the Voice Processing I/O unit. To play audio resources synced from a user's iTunes library, use iPod Library Access. When your sole audio need is to play alerts and user-interface sound effects, use Core Audio's System Sound Services. For other audio applications, including playback of streamed audio, precise synchronization, and access to packets of incoming audio, use Audio Queue Services.
Code for low latency	For the lowest possible playback latency, use OpenAL or use the I/O unit directly.

Preferred Audio Formats in iPhone OS

For uncompressed (highest quality) audio, use 16-bit, little endian, linear PCM audio data packaged in a CAF file. You can convert an audio file to this format in Mac OS X using the `afconvert` command-line tool, as shown here:

```
/usr/bin/afconvert -f caff -d LEI16 {INPUT} {OUTPUT}
```

The `afconvert` tool lets you convert to a wide range of audio data formats and file types. See the `afconvert` man page, and enter `afconvert -h` at a shell prompt, for more information.

For compressed audio when playing one sound at a time, and when you don't need to play audio simultaneously with the iPod application, use the AAC format packaged in a CAF or m4a file.

For less memory usage when you need to play multiple sounds simultaneously, use IMA4 (IMA/ADPCM) compression. This reduces file size but entails minimal CPU impact during decompression. As with linear PCM data, package IMA4 data in a CAF file.

Using Video in iPhone OS

Recording and Editing Video

Starting in iPhone OS 3.0, you can record video, with included audio, on supported devices. To display the video recording interface, create and push a `UIImagePickerController` object, just as for displaying the still-camera interface.

To record video, you must first check that the camera source type (`UIImagePickerControllerSourceTypeCamera`) is available and that the movie media type (`kUTTypeMovie`) is available for the camera. Depending on the media types you assign to the `mediaTypes` property, the picker can directly display the still camera or the video camera, or a selection interface that lets the user choose.

Using the `UIImagePickerControllerDelegate` protocol, register as a delegate of the image picker. Your delegate object receives a completed video recording by way of the `imagePickerController:didFinishPickingMediaWithInfo:` method.

On supported devices, you can also pick previously-recorded videos from a user's photo library.

For more information on using the image picker class, see *UIImagePickerController Class Reference*. For information on trimming recorded videos, see *UIVideoEditorController Class Reference* and *UIVideoEditorControllerDelegate Protocol Reference*.

Playing Video Files

iPhone OS supports the ability to play back video files directly from your application using the Media Player framework, described in *Media Player Framework Reference*. Video playback is supported in full screen mode only and can be used by game developers who want to play short animations or by any developers who want to play media files. When you start a video from your application, the media player interface takes over, fading the screen to black and then fading in the video content. You can play a video with or without user controls for adjusting playback. Enabling some or all of these controls (shown in Figure 7-2) gives the user the ability to change the volume, change the playback point, or start and stop the video. If you disable all of these controls, the video plays until completion.

Figure 7-2 Media player interface with transport controls

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

Listing 7-12 shows a sample method that plays back the video at a specified URL. The play method is an asynchronous call that returns control to the caller while the movie plays. The movie controller loads the movie in a full-screen view, and animates the movie into place on top of the application's existing content. When playback is finished, the movie controller sends a notification received by the application controller object, which releases the movie controller now that it is no longer needed.

Listing 7-12 Playing full-screen movies

```
-(void) playMovieAtURL: (NSURL*) theURL {

    MPMoviePlayerController* theMovie =
                [[MPMoviePlayerController alloc] initWithContentURL: theURL];

    theMovie.scalingMode = MPMovieScalingModeAspectFill;
    theMovie.movieControlMode = MPMovieControlModeHidden;

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

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

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

    [[NSNotificationCenter defaultCenter]
                    removeObserver: self
                              name: MPMoviePlayerPlaybackDidFinishNotification
                            object: theMovie];
```

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

For a list of supported video formats, see *iPhone OS Technology Overview*.

Device Support

iPhone OS supports a variety of features that make the mobile computing experience compelling for users. Through iPhone OS, your applications can access hardware features, such as the accelerometers and camera, and software features, such as the user's photo library. The following sections describe these features and show you how to integrate them into your own applications.

Setting Required Hardware Capabilities

If your application requires device-related features in order to run, you can add a list of required capabilities to your application. At runtime, iPhone OS will not launch your application unless those capabilities are present on the device. Further, the App Store will also restrict your application to users whose devices are capable of running your application.

You add the list of required capabilities by adding the `UIRequiredDeviceCapabilities` key to your application's `info.plist` file. This key is supported in iPhone OS 3.0 and later. The value of this key is either an array or a dictionary. If you use an array, the presence of a given key indicates the corresponding feature is required. If you use a dictionary, you must specify a Boolean value for each key indicating whether the feature is required. In both cases, not including a key indicates that the feature is not required.

Table 8-1 lists the keys that you can include in the array or dictionary associated with the `UIRequiredDeviceCapabilities` key. You should include keys only for the features that your application absolutely requires. If your application can accommodate missing features by not executing the appropriate code paths, you do not need to include the corresponding key.

Table 8-1 Dictionary keys for the `UIRequiredDeviceCapabilities` key

Key	Description
`telephony`	Include this key if your application requires the presence of the Phone application. You might require this feature if your application opens URLs with the `tel` scheme.
`sms`	Include this key if your application requires the presence of the Messages application. You might require this feature if your application opens URLs with the `sms` scheme.
`still-camera`	Include this key if your application uses the `UIImagePickerController` interface to capture images from the device's still camera.
`auto-focus-camera`	Include this key if your application requires auto-focus capabilities in the device's still camera. Although most developers should not need to include this key, you might include it if your application supports macro photography or requires sharper images in order to do some sort of image processing.
`video-camera`	Include this key if your application uses the `UIImagePickerController` interface to capture video from the device's camera.

Key	Description
`wifi`	Include this key if your application requires access to the networking features of the device.
`accelerometer`	Include this key if your application uses the `UIAccelerometer` interface to receive accelerometer events. You do not need to include this key if your application detects only device orientation changes.
`location-services`	Include this key if your application uses the Core Location framework to access the device's current location. (This key refers to the general location services feature. If you specifically need GPS-level accuracy, you should also include the `gps` key.)
`gps`	Include this key if your application requires the presence of GPS (or AGPS) hardware for greater accuracy when tracking locations. If you include this key, you should also include the `location-services` key. You should require GPS only if your application needs more accurate location data than the cell or Wi-fi radios might otherwise allow.
`magnetometer`	Include this key if your application uses Core Location framework to receive heading-related events.
`microphone`	Include this key if your application uses the built-in microphone or supports accessories that provide a microphone.
`opengles-1`	Include this key if your application requires the presence of the OpenGL ES 1.1 interfaces.
`opengles-2`	Include this key if your application requires the presence of the OpenGL ES 2.0 interfaces.
`armv6`	Include this key if your application is compiled only for the armv6 instruction set. (iPhone OS v3.1 and later.)
`armv7`	Include this key if your application is compiled only for the armv7 instruction set. (iPhone OS v3.1 and later.)
`peer-peer`	Include this key if your application requires peer-to-peer connectivity over Bluetooth. (iPhone OS v3.1 and later.)

Determining the Available Hardware Support

Applications designed for iPhone OS must be able to run on devices with different hardware features. Although features such as the accelerometers and Wi-Fi networking are available on all devices, some devices may not include a camera or GPS hardware. If your application requires such features, you should always notify the user of that fact before they purchase the application. For features that are not required, but which you might want to support when present, you must check to see if the feature is available before trying to use it.

Important: If a feature absolutely must be present in order for your application to run, configure the `UIRequiredDeviceCapabilities` key in your application's `Info.plist` file accordingly. This key prevents users from installing applications that require features that are not present on a device. You should not include a key, however, if your application can function with or without the given feature. For more information about configuring this key, see "The Information Property List" (page 26).

Table 8-2 lists the techniques for determining if certain types of hardware are available. You should use these techniques in cases where your application can function without the feature but still uses it when present.

Table 8-2 Identifying available hardware features

Feature	Options
To determine if the network is available…	Use the reachability interfaces of the Software Configuration framework to determine the current network connectivity. For an example of how to use the Software Configuration framework, see *Reachability*.
To determine if the still camera is available…	Use the `isSourceTypeAvailable:` method of the `UIImagePickerController` class to determine if the camera is available. For more information, see "Taking Pictures with the Camera" (page 188).
To determine if audio input (a microphone) is available…	In iPhone OS 3.0 and later, use the `AVAudioSession` class to determine if audio input is available. This class accounts for many different sources of audio input on iPhone OS–based devices, including built-in microphones, headset jacks, and connected accessories. For more information, see *AVAudioSession Class Reference*.
To determine if GPS hardware is present…	Specify a high accuracy level when configuring the `CLLocationManager` object to deliver location updates to your application. The Core Location framework does not provide direct information about the availability of specific hardware but instead uses accuracy values to provide you with the data you need. If the accuracy reported in subsequent location events is insufficient, you can let the user know. For more information, see "Getting the User's Current Location" (page 177)
To determine if a specific accessory is available…	Use the classes of the External Accessory framework to find the appropriate accessory object and connect to it. For more information, see "Communicating with External Accessories" (page 169).

Communicating with External Accessories

In iPhone OS 3.0 and later, the External Accessory framework (`ExternalAccessory.framework`) provides a conduit for communicating with accessories attached to an iPhone or iPod touch device. Application developers can use this conduit to integrate accessory-level features into their applications.

Note: The following sections show you how to connect to accessories from an iPhone application. If you are interested in becoming a developer of accessories for iPhone or iPod touch, you can find information about how to do so on http://developer.apple.com.

To use the features of the External Accessory framework, you must add `ExternalAccessory.framework` to your Xcode project and link against it in any relevant targets. To access the classes and headers of the framework, include an `#import <ExternalAccessory/ExternalAccessory.h>` statement at the top of any relevant source files. For more information on how to add frameworks to your project, see Files in Projects in *Xcode Project Management Guide*. For general information about the classes of the External Accessory framework, see *External Accessory Framework Reference*.

Accessory Basics

Communicating with an external accessory requires you to work closely with the accessory manufacturer to understand the services provided by that accessory. Manufacturers must build explicit support into their accessory hardware for communicating with iPhone OS. As part of this support, an accessory must support at least one command **protocol**, which is a custom scheme for sending data back and forth between the accessory and an attached application. Apple does not maintain a registry of protocols; it is up to the manufacturer to decide which protocols to support and whether to use custom protocols or standard protocols supported by other manufacturers.

As part of your communication with the accessory manufacturer, you must find out what protocols a given accessory supports. To prevent namespace conflicts, protocol names are specified as reverse-DNS strings of the form `com.apple.myProtocol`. This allows each manufacturer to define as many protocols as needed to support their line of accessories.

Applications communicate with an accessory by opening a session to that accessory using a specific protocol. You open a session by creating an instance of the `EASession` class, which contains `NSInputStream` and `NSOutputStream` objects for communicating with the accessory. Your application uses these stream objects to send raw data packets to the accessory and to receive similar packets in return. Therefore, you must understand the expected format of each data packet based on the protocol you want to support.

Declaring the Protocols Your Application Supports

Applications that are able to communicate with an external accessory should declare the protocols they support in their `Info.plist` file. Declaring support for specific protocols lets the system know that your application can be launched when that accessory is connected. If no application supports the connected accessory, the system may choose to launch the App Store and point out applications that do.

To declare the protocols your application supports, you must include the `UISupportedExternalAccessoryProtocols` key in your application's `Info.plist` file. This key contains an array of strings that identify the communications protocols that your application supports. Your application can include any number of protocols in this list and the protocols can be in any order. The system does not use this list to determine which protocol your application should choose; it uses it only to determine if your application is capable of communicating with the accessory. It is up to your code to choose an appropriate communications protocol when it begins talking to the accessory.

Connecting to an Accessory at Runtime

Accessory are not visible through the External Accessory framework until they have been connected by the system and made ready for use. When an accessory does become visible, your application can get the appropriate accessory object and open a session using one or more of the protocols supported by the accessory.

The shared `EAAccessoryManager` object provides the main entry point for applications looking to communicate with accessories. This class contains an array of already connected accessory objects that you can enumerate to see if there is one your application supports. Most of the information in an `EAAccessory` object (such as the name, manufacturer, and model information) is intended for display purposes only. To determine whether your application can connect to an accessory, you must look at the accessory's protocols and see if there is one your application supports.

Note: It is possible for more than one accessory object to support the same protocol. If that happens, your code is responsible for choosing which accessory object to use.

For a given accessory object, only one session at a time is allowed for a specific protocol. The `protocolStrings` property of each `EAAccessory` object contains a dictionary whose keys are the supported protocols. If you attempt to create a session using a protocol that is already in use, the External Accessory framework generates an error.

Listing 8-1 shows a method that checks the list of connected accessories and grabs the first one that the application supports. It creates a session for the designated protocol and configures the input and output streams of the session. By the time this method returns the session object, it is connected to the accessory and ready to begin sending and receiving data.

Listing 8-1 Creating a communications session for an accessory

```
- (EASession *)openSessionForProtocol:(NSString *)protocolString
{
    NSArray *accessories = [[EAAccessoryManager sharedAccessoryManager]
                                connectedAccessories];
    EAAccessory *accessory = nil;
    EASession *session = nil;

    for (EAAccessory *obj in accessories)
    {
        if ([[obj protocolStrings] containsObject:protocolString])
        {
            accessory = obj;
            break;
        }
    }

    if (accessory)
    {
        session = [[EASession alloc] initWithAccessory:accessory
                                forProtocol:protocolString];
        if (session)
        {
            [[session inputStream] setDelegate:self];
           [[session inputStream] scheduleInRunLoop:[NSRunLoop currentRunLoop]
                                    forMode:NSDefaultRunLoopMode];
```

```
            [[session inputStream] open];
            [[session outputStream] setDelegate:self];
           [[session outputStream] scheduleInRunLoop:[NSRunLoop currentRunLoop]
                                             forMode:NSDefaultRunLoopMode];
            [[session outputStream] open];
            [session autorelease];
        }
    }

    return session;
}
```

After the input and output streams are configured, the final step is to process the stream-related data. Listing 8-2 shows the fundamental structure of a delegate's stream processing code. This method responds to events from both input and output streams of the accessory. As the accessory sends data to your application an event arrives indicating there are bytes available to be read. Similarly, when the accessory is ready to receive data from your application, events arrive indicating that fact. (Of course, your application does not always have to wait for an event to arrive before it can write bytes to the stream. It can also call the stream's `hasBytesAvailable` method to see if the accessory is still able to receive data.) For more information on streams and handling stream-related events, see *Stream Programming Guide for Cocoa*.

Listing 8-2 Processing stream events

```
// Handle communications from the streams.
- (void)stream:(NSStream*)theStream handleEvent:(NSStreamEvent)streamEvent
{
    switch (streamEvent)
    {
        case NSStreamHasBytesAvailable:
            // Process the incoming stream data.
            break;

        case NSStreamEventHasSpaceAvailable:
            // Send the next queued command.
            break;

        default:
            break;
    }

}
```

Monitoring Accessory-Related Events

The External Accessory framework is capable of sending notifications whenever a hardware accessory is connected or disconnected. Although it is capable, it does not do so automatically. Your application must specifically request that notifications be generated by calling the `registerForLocalNotifications` method of the `EAAccessoryManager` class. When an accessory is connected, authenticated, and ready to interact with your application, the framework sends an `EAAccessoryDidConnectNotification` notification. When an accessory is disconnected, it sends an `EAAccessoryDidDisconnectNotification` notification. You can register to receive these notifications using the default `NSNotificationCenter`, and both notifications include information about which accessory was affected.

In addition to receiving notifications through the default notification center, an application that is currently interacting with an accessory can assign a delegate to the corresponding `EAAccessory` object and be notified of changes. Delegate objects must conform to the `EAAccessoryDelegate` protocol, which currently contains the optional `accessoryDidDisconnect:` method. You can use this method to receive disconnection notices without first setting up a notification observer.

For more information about how to register to receive notifications, see *Notification Programming Topics for Cocoa*.

Accessing Accelerometer Events

An accelerometer measures changes in velocity over time along a given linear path. The iPhone and iPod touch each contain three accelerometers, one along each of the primary axes of the device. This combination of accelerometers lets you detect movement of the device in any direction. You can use this data to track both sudden movements in the device and the device's current orientation relative to gravity.

Note: In iPhone OS 3.0 and later, if you are trying to detect specific types of motion, such as shaking motions, you should consider tracking that motion using motion events instead of the accelerometer interfaces. Motion events offer a consistent way to detect specific types of accelerometer movement and are described in more detail in "Motion Events" (page 95).

Every application has a single `UIAccelerometer` object that can be used to receive acceleration data. You get the instance of this class using the `sharedAccelerometer` class method of `UIAccelerometer`. Using this object, you set the desired reporting interval and a custom delegate to receive acceleration events. You can set the reporting interval to be as small as 10 milliseconds, which corresponds to a 100 Hz update rate, although most applications can operate sufficiently with a larger interval. As soon as you assign your delegate object, the accelerometer starts sending it data. Thereafter, your delegate receives data at the requested update interval.

Listing 8-3 shows the basic steps for configuring the accelerometer. In this example, the update frequency is 50 Hz, which corresponds to an update interval of 20 milliseconds. The `myDelegateObject` is a custom object that you define; it must support the `UIAccelerometerDelegate` protocol, which defines the method used to receive acceleration data.

Listing 8-3 Configuring the accelerometer

```
#define kAccelerometerFrequency        50 //Hz
-(void)configureAccelerometer
{
    UIAccelerometer*  theAccelerometer = [UIAccelerometer sharedAccelerometer];
    theAccelerometer.updateInterval = 1 / kAccelerometerFrequency;

    theAccelerometer.delegate = self;
    // Delegate events begin immediately.
}
```

The shared accelerometer delivers event data at regular intervals to your delegate's `accelerometer:didAccelerate:` method, shown in Listing 8-4. You can use this method to process the accelerometer data however you want. In general it is recommended that you use some sort of filter to isolate the component of the data in which you are interested.

Listing 8-4 Receiving an accelerometer event

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
{
    UIAccelerationValue x, y, z;
    x = acceleration.x;
    y = acceleration.y;
    z = acceleration.z;

    // Do something with the values.
}
```

To stop the delivery of acceleration events, set the delegate of the shared `UIAccelerometer` object to `nil`. Setting the delegate object to `nil` lets the system know that it can turn off the accelerometer hardware as needed, and thus save battery life.

The acceleration data you receive in your delegate method represents the instantaneous values reported by the accelerometer hardware. Even when a device is completely at rest, the values reported by this hardware can fluctuate slightly. When using these values, you should be sure to account for these fluctuations by averaging out the values over time or by calibrating the data you receive. For example, the Bubble Level sample application provides controls for calibrating the current angle against a known surface. Subsequent readings are then reported relative to the calibrated angle. If your own code requires a similar level of accuracy, you should also include some sort of calibration option in your user interface.

Choosing an Appropriate Update Interval

When configuring the update interval for acceleration events, it is best to choose an interval that minimizes the number of delivered events while still meeting the needs of your application. Few applications need acceleration events delivered 100 times a second. Using a lower frequency prevents your application from running as often and can therefore improve battery life. Table 8-3 lists some typical update frequencies and what you can do with the acceleration data generated at that frequency.

Table 8-3 Common update intervals for acceleration events

Event frequency (Hz)	Usage
10–20	Suitable for use in determining the vector representing the current orientation of the device.
30–60	Suitable for games and other applications that use the accelerometers for real-time user input.
70–100	Suitable for applications that need to detect high-frequency motion. For example, you might use this interval to detect the user hitting the device or shaking it very quickly.

Isolating the Gravity Component from Acceleration Data

If you are using the accelerometer data to detect the current orientation of a device, you need to be able to filter out the portion of the acceleration data that is caused by gravity from the portion that is caused by motion of the device. To do this, you can use a low-pass filter to reduce the influence of sudden changes on the accelerometer data. The resulting filtered values would then reflect the more constant effects of gravity.

Listing 8-5 shows a simplified version of a low-pass filter. This example uses a low-value filtering factor to generate a value that uses 10 percent of the unfiltered acceleration data and 90 percent of the previously filtered value. The previous values are stored in the `accelX`, `accelY`, and `accelZ` member variables of the class. Because acceleration data comes in regularly, these values settle out quickly and respond slowly to sudden but short-lived changes in motion.

Listing 8-5 Isolating the effects of gravity from accelerometer data

```
#define kFilteringFactor 0.1

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    // Use a basic low-pass filter to keep only the gravity component of each
axis.
    accelX = (acceleration.x * kFilteringFactor) + (accelX * (1.0 -
kFilteringFactor));
    accelY = (acceleration.y * kFilteringFactor) + (accelY * (1.0 -
kFilteringFactor));
    accelZ = (acceleration.z * kFilteringFactor) + (accelZ * (1.0 -
kFilteringFactor));

   // Use the acceleration data.
}
```

Isolating Instantaneous Motion from Acceleration Data

If you are using accelerometer data to detect just the instant motion of a device, you need to be able to isolate sudden changes in movement from the constant effect of gravity. You can do that with a high-pass filter.

Listing 8-6 shows a simplified high-pass filter computation. The acceleration values from the previous event are stored in the `accelX`, `accelY`, and `accelZ` member variables of the class. This example computes the low-pass filter value and then subtracts it from the current value to obtain just the instantaneous component of motion.

Listing 8-6 Getting the instantaneous portion of movement from accelerometer data

```
#define kFilteringFactor 0.1

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    // Subtract the low-pass value from the current value to get a simplified
high-pass filter
    accelX = acceleration.x - ( (acceleration.x * kFilteringFactor) + (accelX
* (1.0 - kFilteringFactor)) );
    accelY = acceleration.y - ( (acceleration.y * kFilteringFactor) + (accelY
* (1.0 - kFilteringFactor)) );
```

```
    accelZ = acceleration.z - ( (acceleration.z * kFilteringFactor) + (accelZ
* (1.0 - kFilteringFactor)) );

   // Use the acceleration data.
}
```

Getting the Current Device Orientation

If you need to know only the general orientation of the device, and not the exact vector of orientation, you should use the methods of the `UIDevice` class to retrieve that information. Using the `UIDevice` interface is simpler and does not require you to calculate the orientation vector yourself.

Before getting the current orientation, you must tell the `UIDevice` class to begin generating device orientation notifications by calling the `beginGeneratingDeviceOrientationNotifications` method. Doing so turns on the accelerometer hardware (which may otherwise be off to conserve power).

Shortly after enabling orientation notifications, you can get the current orientation from the `orientation` property of the shared `UIDevice` object. You can also register to receive `UIDeviceOrientationDidChangeNotification` notifications, which are posted whenever the general orientation changes. The device orientation is reported using the `UIDeviceOrientation` constants, which indicate whether the device is in landscape or portrait mode or whether the device is face up or face down. These constants indicate the physical orientation of the device and need not correspond to the orientation of your application's user interface.

When you no longer need to know the orientation of the device, you should always disable orientation notifications by calling the `endGeneratingDeviceOrientationNotifications` method of `UIDevice`. Doing so gives the system the opportunity to disable the accelerometer hardware if it is not in use elsewhere.

Using Location and Heading Services

The Core Location framework provides support for locating the user's current location and heading. This framework gathers information from the available onboard hardware and reports it to your application asynchronously. The availability of data is dependent on the type of device and whether or not the needed hardware is currently enabled, which it may not be if the device is in airplane mode.

To use the features of the Core Location framework, you must add `CoreLocation.framework` to your Xcode project and link against it in any relevant targets. To access the classes and headers of the framework, include an `#import <CoreLocation/CoreLocation.h>` statement at the top of any relevant source files. For more information on how to add frameworks to your project, see Files in Projects in *Xcode Project Management Guide*.

For general information about the classes of the Core Location framework, see *Core Location Framework Reference*.

Getting the User's Current Location

The Core Location framework lets you locate the current position of the device and use that information in your application. The framework takes advantage of the device's built-in hardware, triangulating a position fix from available signal information. It then reports the location to your code and occasionally updates that position information as it receives new or improved signals.

If you do use the Core Location framework, be sure to do so sparingly and to configure the location service appropriately. Gathering location data involves powering up the onboard radios and querying the available cell towers, Wi-Fi hotspots, or GPS satellites, which can take several seconds. In addition, requesting more accurate location data may require the radios to remain on for a longer period of time. Leaving this hardware on for extended periods of time can drain the device's battery. Given that position information does not change too often, it is usually sufficient to establish an initial position fix and then acquire updates periodically after that. If you are sure you need regular position updates, you can also configure the service with a minimum threshold distance to minimize the number of position updates your code must process.

To retrieve the user's current location, create an instance of the `CLLocationManager` class and configure it with the desired accuracy and threshold parameters. To begin receiving location notifications, assign a delegate to the object and call the `startUpdatingLocation` method to start the determination of the user's current location. When new location data is available, the location manager notifies its assigned delegate object. If a location update has already been delivered, you can also get the most recent location data directly from the `CLLocationManager` object without waiting for a new event to be delivered.

Listing 8-7 shows implementations of a custom `startUpdates` method and the `locationManager:didUpdateToLocation:fromLocation:` delegate method. The `startUpdates` method creates a new location manager object (if one does not already exist) and uses it to start generating location updates. (In this case, the `locationManager` variable is a member variable declared by the `MyLocationGetter` class, which also conforms to the `CLLocationManagerDelegate` protocol.) The handler method uses the timestamp of the event to determine how recent it is. If it is an old event, the handler ignores it and waits for a more recent one, at which point it disables the location service.

Listing 8-7 Initiating and processing location updates

```
#import <CoreLocation/CoreLocation.h>

@implementation MyLocationGetter
- (void)startUpdates
{
    // Create the location manager if this object does not
    // already have one.
    if (nil == locationManager)
        locationManager = [[CLLocationManager alloc] init];

    locationManager.delegate = self;
    locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;

    // Set a movement threshold for new events
    locationManager.distanceFilter = 500;

    [locationManager startUpdatingLocation];
}


// Delegate method from the CLLocationManagerDelegate protocol.
- (void)locationManager:(CLLocationManager *)manager
```

```
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    // If it's a relatively recent event, turn off updates to save power
    NSDate* eventDate = newLocation.timestamp;
    NSTimeInterval howRecent = [eventDate timeIntervalSinceNow];
    if (abs(howRecent) < 5.0)
    {
        [manager stopUpdatingLocation];

        printf("latitude %+.6f, longitude %+.6f\n",
                newLocation.coordinate.latitude,
                newLocation.coordinate.longitude);
    }
    // else skip the event and process the next one.
}
@end
```

Checking the timestamp of an event is recommended because the location service often returns the last cached location event immediately. It can take several seconds to obtain a rough location fix so the old data simply serves as a way to reflect the last known location. You can also use the accuracy as a means of determining whether you want to accept an event. As it receives more accurate data, the location service may return additional events, with the accuracy values reflecting the improvements accordingly.

Note: The Core Location framework records timestamp values at the beginning of each location query, not when that query returns. Because Core Location uses several different techniques to get a location fix, queries can sometimes come back in a different order than their timestamps might otherwise indicate. As a result, it is normal for new events to sometimes have timestamps that are slightly older than those from previous events. The framework concentrates on improving the accuracy of the location data with each new event it delivers, regardless of the timestamp values.

Getting Heading-Related Events

The Core Location framework supports two different ways to get heading-related information. Devices that contain GPS hardware can provide rough information about the current direction of travel using the same location events used to determine the user's latitude and longitude. Devices that contain a magnetometer can provide more precise heading information through heading objects, which are instances of the `CLHeading` class.

The process for retrieving rough location events using GPS hardware is the same as the one described in "Getting the User's Current Location" (page 177). The `CLLocation` object reported to your application delegate contains `course` and `speed` properties with the relevant information. This interface is appropriate for most applications that track the user's movement over time, such as those that implement car-based navigation systems. For compass-based applications, or other applications that might involve knowing the user's current heading when not in motion, you can ask the location manager to provide heading objects.

In order to receive heading objects, your application must be running on a device that contains a magnetometer. A magnetometer measures nearby magnetic fields emanating from the Earth and uses them to determine the precise orientation of the device. Although a magnetometer can be affected by local magnetic fields, such as those emanating from fixed magnets found in audio speakers, motors, and many other types of electronic devices, the Core Location framework is smart enough to filter out many local fields to ensure that heading objects contain useful data.

Note: If your application requires either course or heading information, you should include the `UIRequiredDeviceCapabilities` key in your application's `Info.plist` file appropriately. This key lets you specify exactly which features your application requires in order to operate and can be used to require the presence of the GPS or magnetometer hardware. For more information about setting the value of this key, see "The Information Property List" (page 26).

To receive heading events, you create a `CLLocationManager` object, assign a delegate to it, and call the `startUpdatingHeading` method, as shown in Listing 8-8. Before asking for heading events, however, you should always check the `headingAvailable` property of the location manager to make sure that the appropriate hardware is present. If it is not, you should fall back to using location-based events to retrieve course information.

Listing 8-8 Initiating the delivery of heading events

```
CLLocationManager* locManager = [[CLLocationManager alloc] init];
if (locManager.headingAvailable)
{
   locManager.delegate = myDelegateObject; // Assign your custom delegate object
   locManager.headingFilter = 5;
   [locManager startUpdatingHeading];
}
else
   // Use location events instead
```

The object you assign to the delegate property must conform to the `CLLocationManagerDelegate` protocol. When a new heading event arrives, the location manager object calls the `locationManager:didUpdateHeading:` method to deliver that event to your application. Upon receiving a new event, you should check the `headingAccuracy` property to ensure that the data you just received is valid, as shown in Listing 8-9.

Listing 8-9 Processing heading events

```
- (void)locationManager:(CLLocationManager*)manager
didUpdateHeading:(CLHeading*)newHeading
{
   // If the accuracy is valid, go ahead and process the event.
   if (newHeading.headingAccuracy > 0)
   {
      CLLocationDirection theHeading = newHeading.magneticHeading;

      // Do something with the event data.
   }
}
```

The `magneticHeading` property of a `CLHeading` object contains the main heading data and is always present. This property gives you a heading measurement relative to magnetic North, which is not at the same location as the north pole. If you want a heading relative to the north pole (also known as geographical North), you must call the `startUpdatingLocation` method to start the delivery of location updates before you call `startUpdatingHeading`. You can then use the `trueHeading` property of the `CLHeading` object to obtain the heading to geographical North.

Displaying Maps and Annotations

Introduced in iPhone OS 3.0, the Map Kit framework lets you embed a fully functional map interface into your application window. The map support provided by this framework includes many of the features normally found in the Maps application. You can display standard street-level map information, satellite imagery, or a combination of the two. You can zoom and pan the map programmatically, and the framework provides automatic support for the touch events that let users zoom and pan the map. You can annotate the map with custom information. And you can use the reverse geocoder feature of the framework to find the address associated with map coordinates.

To use the features of the Map Kit framework, you must add `MapKit.framework` to your Xcode project and link against it in any relevant targets. To access the classes and headers of the framework, include an `#import <MapKit/MapKit.h>` statement at the top of any relevant source files. For more information on how to add frameworks to your project, see Files in Projects in *Xcode Project Management Guide*. For general information about the classes of the Map Kit framework, see *MapKit Framework Reference*.

Important: The Map Kit framework uses Google services to provide map data. Use of the framework and its associated interfaces binds you to the Google Maps/Google Earth API terms of service. You can find these terms of service at http://code.google.com/apis/maps/iphone/terms.html.

Adding a Map View to Your User Interface

To add maps to your application, you embed an instance of the `MKMapView` class into your application's view hierarchy. This class provides support both for displaying the map information and for managing the user interactions with that information. You can create an instance of this class programmatically (initializing it with the `initWithFrame:` method) or use Interface Builder to add it to one of your nib files.

Because it is a view, you can use the `frame` property of a map view to move and resize it however you like within your view hierarchy. Although the map view does not have any controls of its own, you can layer toolbars and other views on top of it to add controls for interacting with the map contents. Any subviews you add to the map view remain fixed in place and do not scroll with the map contents. If you want to add custom content to the map itself, and have that content scroll with the map, you must create annotations as described in "Displaying Annotations" (page 182).

The `MKMapView` class has a handful of properties that you can configure before displaying it. The most important property to set is the `region` property. This property defines the portion of the map that should be displayed initially and is how you zoom and pan the contents of the map.

Zooming and Panning the Map Content

The `region` property of the `MKMapView` class controls the portion of the map that is displayed at any given time. When you want to zoom or pan the map, all you have to do is change the values in this property appropriately. The property consists of an `MKCoordinateRegion` structure, which has the following definition:

```
typedef struct {
   CLLocationCoordinate2D center;
   MKCoordinateSpan span;
} MKCoordinateRegion;
```

To pan the map to a new location, you change the values in the *center* field. To zoom in and out, you change the values in the *span* field. You specify the values for these fields using map coordinates, which are measured in degrees, minutes, and seconds. For the span field, the values you specify represent latitudinal and longitudinal distances. Although latitudinal distances are relatively fixed at approximately 111 kilometers per degree, longitudinal distances vary with the latitude. At the equator, longitudinal values are approximately 111 kilometers per degree but at the poles they approach zero. Of course, you can always use the `MKCoordinateRegionMakeWithDistance` function to create a region using kilometer values instead of degrees.

To update the map without animating your changes, you can modify the `region` or `centerCoordinate` property directly. To animate your changes, you must use either the `setRegion:animated:` or `setCenterCoordinate:animated:` method. The `setCenterCoordinate:animated:` method lets you pan the map without zooming in or out inadvertently. The `setRegion:animated:` method lets you pan and zoom at the same time. For example, to pan the map to the left by half the current map width, you could use the following code to find the coordinate at the left edge of the map and use that as the new center point, as shown here:

```
CLLocationCoordinate2D mapCenter = myMapView.centerCoordinate;
mapCenter = [myMapView convertPoint:
               CGPointMake(1, (myMapView.frame.size.height/2.0))
               toCoordinateFromView:myMapView];
[myMapView setCenterCoordinate:mapCenter animated:YES];
```

To zoom in and out, instead of modifying the center coordinate, you modify the span. To zoom in, you decrease the span. To zoom out, you increase it. In other words if the current span is one degree, specifying a span of two degrees zooms out by a factor of two:

```
MKCoordinateRegion theRegion = myMapView.region;

// Zoom out
theRegion.span.longitudeDelta *= 2.0;
theRegion.span.latitudeDelta *= 2.0;
[myMapView setRegion:theRegion animated:YES];
```

Displaying the User's Current Location

The Map Kit framework includes built-in support for displaying the user's current location on the map. To show this location, set the `showsUserLocation` property of your map view object to `YES`. Doing so causes the map view to use the Core Location framework to find the user's location and add an annotation of type `MKUserLocation` to the map.

The addition of the `MKUserLocation` annotation object to the map is reported by the delegate the same way custom annotations are. If you want to associate a custom annotation view with the user's location, you should return that view from your delegate object's `mapView:viewForAnnotation:` method. If you want to use the default annotation view, you should return `nil` from that method instead.

Converting Between Coordinates and Pixels

Although you normally specify points on the map using latitude and longitude values, there may be times when you need to convert to and from pixels in the map view object itself. For example, if you allow the user to drag annotations around the map surface, the event handlers for your custom annotation views would

need to convert frame coordinates to map coordinates so that they could update the associated annotation object. The `MKMapView` class includes several routines for converting back and forth between map coordinates and the local coordinate system of the map view object itself, including:

```
convertCoordinate:toPointToView:
convertPoint:toCoordinateFromView:
convertRegion:toRectToView:
convertRect:toRegionFromView:
```

For more information about handling events in your custom annotations, see "Handling Events in an Annotation View" (page 185).

Displaying Annotations

Annotations are pieces of map content that you define and layer on top of the map itself. The Map Kit framework implements annotations in two parts: an annotation object and a view to display that annotation. For the most part, you are responsible for providing these custom objects. However, the framework does provide standard annotations and views that you can use as well.

Displaying annotations in your map view is a two-step process:

1. Create your annotation object and add it to your map view.
2. Implement the `mapView:viewForAnnotation:` method of your delegate object and create the corresponding annotation view there.

An annotation object is any object that conforms to the `MKAnnotation` protocol. Typically annotation objects are relatively small data objects that store the coordinate of the annotation and the relevant information about the annotation, such as its name. Because annotations are defined using a protocol, any object in your application can become one. In practice though, you should make your annotation objects lightweight, because the map view keeps references to them until you remove them explicitly; however, the same is not necessarily true of annotation views.

When an annotation needs to be displayed on screen, the map view is responsible for making sure the annotation object has an associated annotation view. It does this by calling the the `mapView:viewForAnnotation:` method of its delegate object when the annotation's coordinate is about to become visible on the screen. Because annotation views tend to be more heavyweight objects than their corresponding annotation objects, though, the map view tries not to keep too many annotation views around in memory at the same time. To do this, it implements a recycling scheme for annotation views that is similar to how table views recycle table cells during scrolling. When an annotation view moves off screen, the map view can disassociate that view from its annotation object and put it on a reuse queue. Before creating a new annotation view object, your delegate's `mapView:viewForAnnotation:` method should always check this reuse queue to see if an existing view is available by calling the map view's `dequeueReusableAnnotationViewWithIdentifier:` method. If that method returns a valid view object, you can reinitialize the view and return it; otherwise, you should create and return a new view object.

Adding and Removing Annotation Objects

You do not add annotation views to the map directly. Instead, you add annotation objects, which are typically not views. An annotation object is any object in your application that conforms to the `MKAnnotation` protocol. The most important part of any annotation object is its `coordinate` property, which is a required property of the `MKAnnotation` protocol and provides the anchor point for the annotation on the map.

To add an annotation to the map view, all you have to do is call the `addAnnotation:` or `addAnnotations:` method of the map view object. It is up to you to decide when it is appropriate to add annotations to the map view and to provide the user interface for doing so. You can provide a toolbar with commands that allow the user to create new annotations or you can create the annotations programmatically yourself, perhaps using a local or remote database of information.

If your application needs to dispose of an old annotation, you should always remove it from the map using the `removeAnnotation:` or `removeAnnotations:` method before deleting it. Because the map view shows all annotations that it knows about, you need to remove annotations explicitly if you do not want them displayed on the map. For example, if your application allows the user to filter a list of restaurants or local sights, you would need to remove any annotations that did not match the filter criteria.

Defining Annotation Views

The Map Kit framework provides two annotation view classes: `MKAnnotationView` and `MKPinAnnotationView`. The `MKAnnotationView` class is a concrete view that defines the basic behavior for all annotation views. The `MKPinAnnotationView` class is a subclass of `MKAnnotationView` that displays one of the standard system pin images at the associated annotation's coordinate point.

You can use the `MKAnnotationView` class as is to display simple annotations or you can subclass it to provide more interactive behavior. When using the class as is, you provide a custom image to represent the content you want to display on the map and assign it to the `image` property of the annotation view. Using the class this way is perfect for situations where your you do not display dynamically changing content and do not support any user interactivity. However, if you do want dynamic content or user interactivity, you must define a custom subclass.

In a custom subclass, you can draw dynamic content in one of two ways. You can continue to use the image property to display images for the annotation, perhaps setting up a timer so that you can change the current image at regular intervals. You can also override the view's `drawRect:` method and draw your content explicitly, again setting up a timer so that you can call the view's `setNeedsDisplay` method periodically.

When drawing content using the `drawRect:` method, you must always remember to specify the size of your annotation view shortly after initialization. The default initialization method for annotation views does not take a frame rectangle as a parameter. Instead, it uses the image you specify in the `image` property to set that frame size later. If you do not set an image, though, you must set the frame size explicitly in order for your rendered content to be visible.

For information on how to support user interactivity in your annotation views, see "Handling Events in an Annotation View" (page 185). For information on how to set up timers, see *Timer Programming Topics for Cocoa*.

Creating Annotation Views

You always create annotation views in the `mapView:viewForAnnotation:` method of your delegate object. Before creating a new view, you should always check to see if there is an existing view waiting to be used by calling the `dequeueReusableAnnotationViewWithIdentifier:` method. If this method returns a

non-`nil` value, you should assign the annotation provided by the map view to the view's `annotation` property, perform any additional configuration needed to put the view in a known state, and return it. If the method returns `nil`, you should proceed with creating and returning a new annotation view object.

Listing 8-10 shows a sample implementation of the `mapView:viewForAnnotation:` method. This method provides pin annotation views for custom annotation objects. If an existing pin annotation view already exists, this method associates the annotation object to that view. If no view is in the reuse queue, this method creates a new one, setting up the basic properties of the view and configuring an accessory view for the annotation's callout.

Listing 8-10 Creating annotation views

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
                    viewForAnnotation:(id <MKAnnotation>)annotation
{
    // If it's the user location, just return nil.
    if ([annotation isKindOfClass:[MKUserLocation class]])
        return nil;

    // Handle any custom annotations.
    if ([annotation isKindOfClass:[CustomPinAnnotation class]])
    {
        // Try to dequeue an existing pin view first.
        MKPinAnnotationView*    pinView = (MKPinAnnotationView*)[mapView
        dequeueReusableAnnotationViewWithIdentifier:@"CustomPinAnnotation"];

        if (!pinView)
        {
            // If an existing pin view was not available, create one
          pinView = [[[MKPinAnnotationView alloc] initWithAnnotation:annotation
                      reuseIdentifier:@"CustomPinAnnotation"]
                             autorelease];
            pinView.pinColor = MKPinAnnotationColorRed;
            pinView.animatesDrop = YES;
            pinView.canShowCallout = YES;

            // Add a detail disclosure button to the callout.
            UIButton* rightButton = [UIButton buttonWithType:
                                UIButtonTypeDetailDisclosure];
            [rightButton addTarget:self action:@selector(myShowDetailsMethod:)
                                forControlEvents:UIControlEventTouchUpInside];
            pinView.rightCalloutAccessoryView = rightButton;
        }
        else
            pinView.annotation = annotation;

        return pinView;
    }

    return nil;
}
```

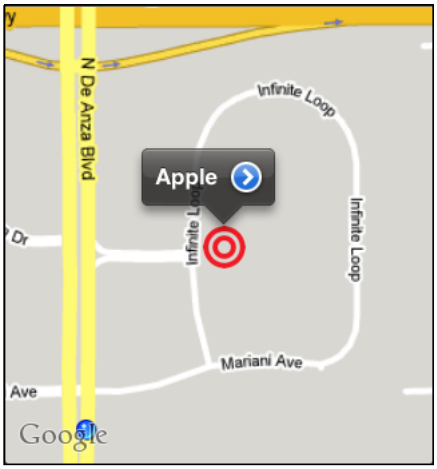
Handling Events in an Annotation View

Although they live in a special layer above the map content, annotation views are full-fledged views capable of receiving touch events. You can use these events to provide more user interactivity for your annotations. For example, you could use touch events in the view to drag the view around the surface of the map.

Note: Because maps are displayed in a scrolling interface, there is typically a short delay between the time the user touches your custom view and the corresponding events are delivered. This delay gives the underlying scroll view a chance to determine if the touch event is part of a scrolling gesture.

The following sequence of code listings show you how to implement a user-movable annotation view. The annotation view displays a bullseye image directly over the annotation's coordinate point and includes a custom accessory view for displaying details about the target. Figure 8-1 shows an instance of this annotation view, along with its callout bubble.

Figure 8-1 The bullseye annotation view



Listing 8-11 shows the definition of the `BullseyeAnnotationView` class. The class includes some additional member variables that it uses during tracking to move the view correctly. It also stores a pointer to the map view itself, the value for which is set by the code in the `mapView:viewForAnnotation:` method when it creates or reinitializes the annotation view. The map view object is needed when event tracking is finished to adjust the map coordinate of the annotation object.

Listing 8-11 The BullseyeAnnotationView class

```
@interface BullseyeAnnotationView : MKAnnotationView
{
    BOOL isMoving;
    CGPoint startLocation;
    CGPoint originalCenter;

    MKMapView* map;
}

@property (assign,nonatomic) MKMapView* map;

- (id)initWithAnnotation:(id <MKAnnotation>)annotation;
```

```
@end

@implementation BullseyeAnnotationView
@synthesize map;
- (id)initWithAnnotation:(id <MKAnnotation>)annotation
{
    self = [super initWithAnnotation:annotation
                reuseIdentifier:@"BullseyeAnnotation"];
    if (self)
    {
        UIImage*    theImage = [UIImage imageNamed:@"bullseye32.png"];
        if (!theImage)
            return nil;

        self.image = theImage;
        self.canShowCallout = YES;
        self.multipleTouchEnabled = NO;
        map = nil;

        UIButton*    rightButton = [UIButton buttonWithType:
                      UIButtonTypeDetailDisclosure];
        [rightButton addTarget:self action:@selector(myShowAnnotationAddress:)
                      forControlEvents:UIControlEventTouchUpInside];
        self.rightCalloutAccessoryView = rightButton;
    }
    return self;
}
@end
```

When a touch event first arrives in a bullseye view, the `touchesBegan:withEvent:` method of that class records information about the initial touch event, as shown in Listing 8-12. It uses this information later in its `touchesMoved:withEvent:` method to adjust the position of the view. All location information is stored in the coordinate space of the superview.

Listing 8-12 Tracking the view's location

```
@implementation BullseyeAnnotationView (TouchBeginMethods)
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // The view is configured for single touches only.
    UITouch* aTouch = [touches anyObject];
    startLocation = [aTouch locationInView:[self superview]];
    originalCenter = self.center;

    [super touchesBegan:touches withEvent:event];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch* aTouch = [touches anyObject];
    CGPoint newLocation = [aTouch locationInView:[self superview]];
    CGPoint newCenter;

    // If the user's finger moved more than 5 pixels, begin the drag.
    if ( (abs(newLocation.x - startLocation.x) > 5.0) ||
         (abs(newLocation.y - startLocation.y) > 5.0) )
         isMoving = YES;
```

```
    // If dragging has begun, adjust the position of the view.
    if (isMoving)
    {
        newCenter.x = originalCenter.x + (newLocation.x - startLocation.x);
        newCenter.y = originalCenter.y + (newLocation.y - startLocation.y);
        self.center = newCenter;
    }
    else    // Let the parent class handle it.
        [super touchesMoved:touches withEvent:event];
}
@end
```

When the user stops dragging an annotation view, you need to adjust the coordinate of the original annotation to ensure the view remains in the new position. Listing 8-13 shows the `touchesEnded:withEvent:` method for the `BullseyeAnnotationView` class. This method uses the map member variable to convert the pixel-based point into a map coordinate value. Because the `coordinate` property of an annotation is normally read-only, the annotation object in this case implements a custom `changeCoordinate` method to update the value it stores locally and reports using the `coordinate` property. If the touch event was cancelled for some reason, the `touchesCancelled:withEvent:` method returns the annotation view to its original position.

Listing 8-13 Handling the final touch events

```
@implementation BullseyeAnnotationView (TouchEndMethods)
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (isMoving)
    {
        // Update the map coordinate to reflect the new position.
        CGPoint newCenter = self.center;
        BullseyeAnnotation* theAnnotation = self.annotation;
        CLLocationCoordinate2D newCoordinate = [map convertPoint:newCenter
                            toCoordinateFromView:self.superview];

        [theAnnotation changeCoordinate:newCoordinate];

        // Clean up the state information.
        startLocation = CGPointZero;
        originalCenter = CGPointZero;
        isMoving = NO;
    }
    else
        [super touchesEnded:touches withEvent:event];
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (isMoving)
    {
        // Move the view back to its starting point.
        self.center = originalCenter;

        // Clean up the state information.
        startLocation = CGPointZero;
        originalCenter = CGPointZero;
        isMoving = NO;
```

```
    }
    else
        [super touchesCancelled:touches withEvent:event];
}
@end
```

Getting Placemark Information from the Reverse Geocoder

The Map Kit framework deals primarily with map coordinate values, which consist of a latitude and longitude pair. Although map coordinates are good for code, they are not always the easiest things for users to understand. To make it easier for users, you can use the `MKReverseGeocoder` class to obtain placemark information such as the street address, city, state, and country associated with a given map coordinate.

The `MKReverseGeocoder` class queries the underlying map service for information about the map coordinate you specify. Because it requires access to the network, a reverse geocoder object always performs its query asynchronously and reports its results back using the associated delegate object. The delegate for a reverse geocoder must conform to the `MKReverseGeocoderDelegate` protocol.

To start a reverse geocoder query, create an instance of the `MKReverseGeocoder`, class, assign an appropriate object to the `delegate` property, and call the `start` method. If the query completes successfully, your delegate receives an `MKPlacemark` object with the results. Placemark objects are themselves annotation objects—that is, they adopt the `MKAnnotation` protocol—so you can add them to your map view's list of annotations if you want.

The Google terms of service require that the `MKReverseGeocoder` class be used in conjunction with a Google map. In addition, each application that uses the Map Kit framework has a limited amount of reverse geocoding capacity, so it is to your advantage to use reverse geocode requests sparingly. Here are some rules of thumb to apply when creating your requests:

- Send at most one reverse-geocoding request for any one user action.
- If the user performs multiple actions that involve reverse-geocoding the same location, reuse the results from the initial reverse-geocoding request instead of starting individual requests for each action.
- When you want to update the location automatically (such as when the user is moving), reissue the reverse-geocoding request only when the user's location has moved a significant distance and after a reasonable amount of time has passed. For example, in a typical situation, you should not send more than one reverse-geocode request per minute.
- Do not start a reverse-geocoding request at a time when the user will not see the results immediately. For example, do not start a request if your application recently resigned the active state (possibly because of an interruption such as a phone call) and is waiting to become active again.

Taking Pictures with the Camera

UIKit provides access to a device's camera through the `UIImagePickerController` class. This class displays the standard system interface for taking pictures using the available camera. It also supports optional controls for resizing and cropping the image after the user takes it. This class can also be used to select photos from the user's photo library.

The view representing the camera interface is a modal view that is managed by the `UIImagePickerController` class. You should never access this view directly from your code. To display it, you must call the `presentModalViewController:animated:` method of the currently active view controller, passing a `UIImagePickerController` object as the new view controller. Upon being installed, the picker controller automatically slides the camera interface into position, where it remains active until the user approves the picture or cancels the operation. At that time, the picker controller notifies its delegate of the user's choice.

Interfaces managed by the `UIImagePickerController` class may not be available on all devices. Before displaying the camera interface, you should always make sure that the interface is available by calling the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class. You should always respect the return value of this method. If this method returns `NO`, it means that the current device does not have a camera or that the camera is currently unavailable for some reason. If the method returns `YES`, you display the camera interface by doing the following:

1. Create a new `UIImagePickerController` object.
2. Assign a delegate object to the picker controller.

   In most cases, the current view controller acts as the delegate for the picker, but you can use an entirely different object if you prefer. The delegate object must conform to the `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` protocols.

   Note: If your delegate does not conform to the `UINavigationControllerDelegate` protocol, you may see a warning during compilation. However, because the methods of this protocol are optional, the warning has no impact on your code. To eliminate the warning, you can include the `UINavigationControllerDelegate` protocol in the list of supported protocols for your delegate's class. You do not need to implement the corresponding methods.

3. Set the picker type to `UIImagePickerControllerSourceTypeCamera`.
4. Optionally, enable or disable the picture editing controls by assigning an appropriate value to the `allowsImageEditing` property.
5. Call the `presentModalViewController:animated:` method of the current view controller to display the picker.

Listing 8-14 shows the code representing the preceding set of steps. As soon as you call the `presentModalViewController:animated` method, the picker controller takes over, displaying the camera interface and responding to all user interactions until the interface is dismissed. To choose an existing photo from the user's photo library, all you have to do is change the value in the `sourceType` property of the picker to `UIImagePickerControllerSourceTypePhotoLibrary`.

Listing 8-14 Displaying the interface for taking pictures

```
-(BOOL)startCameraPickerFromViewController:(UIViewController*)controller
usingDelegate:(id<UIImagePickerControllerDelegate>)delegateObject
{
    if ( (![UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
            || (delegateObject == nil) || (controller == nil))
        return NO;
```

```
    UIImagePickerController* picker = [[UIImagePickerController alloc] init];
    picker.sourceType = UIImagePickerControllerSourceTypeCamera;
    picker.delegate = delegateObject;
    picker.allowsImageEditing = YES;

    // Picker is displayed asynchronously.
    [controller presentModalViewController:picker animated:YES];
    return YES;
}
```

When the user taps the appropriate button to dismiss the camera interface, the `UIImagePickerController` notifies the delegate of the user's choice but does not dismiss the interface. The delegate is responsible for dismissing the picker interface. (Your application is also responsible for releasing the picker when done with it, which you can do in the delegate methods.) It is for this reason that the delegate is actually the view controller object that presented the picker in the first place. Upon receiving the delegate message, the view controller would call its `dismissModalViewControllerAnimated:` method to dismiss the camera interface.

Listing 8-15 shows the delegate methods for dismissing the camera interface displayed in Listing 8-14 (page 189). These methods are implemented by a custom `MyViewController` class, which is a subclass of `UIViewController` and, for this example, is considered to be the same object that displayed the picker in the first place. The `useImage:` method is an empty placeholder for the work you would do in your own version of this class and should be replaced by your own custom code.

Listing 8-15 Delegate methods for the image picker

```
@implementation MyViewController (ImagePickerDelegateMethods)

- (void)imagePickerController:(UIImagePickerController *)picker
                    didFinishPickingImage:(UIImage *)image
                    editingInfo:(NSDictionary *)editingInfo
{
    [self useImage:image];

    // Remove the picker interface and release the picker object.
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
    [picker release];
}

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
    [picker release];
}

// Implement this method in your code to do something with the image.
- (void)useImage:(UIImage*)theImage
{
}
@end
```

If image editing is enabled and the user successfully picks an image, the `image` parameter of the `imagePickerController:didFinishPickingImage:editingInfo:` method contains the edited image. You should treat this image as the selected image, but if you want to store the original image, you can get it (along with the crop rectangle) from the dictionary in the `editingInfo` parameter.

Picking a Photo from the Photo Library

UIKit provides access to the user's photo library through the `UIImagePickerController` class. This controller displays a photo picker interface, which provides controls for navigating the user's photo library and selecting an image to return to your application. You also have the option of enabling user editing controls, which let the user the pan and crop the returned image. This class can also be used to present a camera interface.

Because the `UIImagePickerController` class is used to display the interface for both the camera and the user's photo library, the steps for using the class are almost identical for both. The only difference is that you assign the `UIImagePickerControllerSourceTypePhotoLibrary` value to the `sourceType` property of the picker object. The steps for displaying the camera picker are discussed in "Taking Pictures with the Camera" (page 188).

Note: As you do for the camera picker, you should always call the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class and respect the return value of the method. You should never assume that a given device has a photo library. Even if the device has a library, this method could still return `NO` if the library is currently unavailable.

Using the Mail Composition Interface

In iPhone OS 3.0 and later, you can use the `MFMailComposeViewController` class to present a standard mail composition interface inside your own applications. Prior to displaying the interface, you use the methods of the class to configure the email recipients, the subject, body, and any attachments you want to include. Upon posting the interface (using the standard view controller techniques), the user has the option of editing the email contents before submitting the email to the Mail application for delivery. The user also has the right to cancel the email altogether.

Note: In all versions of iPhone OS, you can also compose email messages by creating and opening a URL that uses the `mailto` scheme. URLs of this type are automatically handled by the Mail application. For more information on how to open a URL of this type, see "Communicating with Other Applications" (page 34).

To use the mail composition interface, you must add `MessageUI.framework` to your Xcode project and link against it in any relevant targets. To access the classes and headers of the framework, include an `#import <MessageUI/MessageUI.h>` statement at the top of any relevant source files. For information on how to add frameworks to your project, see Files in Projects in *Xcode Project Management Guide*.

To use the `MFMailComposeViewController` class in your application, you create an instance and use its methods to set the initial email data. You must also assign an object to the `mailComposeDelegate` property of the view controller to handle the dismissal of the interface when the user accepts or cancels the email. The delegate object you specify must conform to the `MFMailComposeViewControllerDelegate` protocol.

When specifying email addresses for the mail composition interface, you specify plain string objects. If you want to use email addresses from the user's list of contacts, you can use the Address Book framework to retrieve that information. For more information on how to get email and other information using this framework, see *Address Book Programming Guide for iPhone OS*.

Listing 8-16 shows the code for creating the `MFMailComposeViewController` object and displaying the mail composition interface modally in your application. You would include the `displayComposerSheet` method in one of your custom view controllers and call the method as needed to display the interface. In this example, the parent view controller assigns itself as the delegate and implements the `mailComposeController:didFinishWithResult:error:` method. The delegate method dismisses the delegate without taking any further actions. In your own application, you could use the delegate to track whether the user sent or canceled the email by examining the value in the *`result`* parameter.

Listing 8-16 Posting the mail composition interface

```
@implementation WriteMyMailViewController (MailMethods)

-(void)displayComposerSheet
{
    MFMailComposeViewController *picker = [[MFMailComposeViewController alloc]
 init];
    picker.mailComposeDelegate = self;

    [picker setSubject:@"Hello from California!"];

    // Set up the recipients.
    NSArray *toRecipients = [NSArray arrayWithObjects:@"first@example.com",
                                    nil];
    NSArray *ccRecipients = [NSArray arrayWithObjects:@"second@example.com",
                                    @"third@example.com", nil];
    NSArray *bccRecipients = [NSArray arrayWithObjects:@"four@example.com",
                                    nil];

    [picker setToRecipients:toRecipients];
    [picker setCcRecipients:ccRecipients];
    [picker setBccRecipients:bccRecipients];

    // Attach an image to the email.
    NSString *path = [[NSBundle mainBundle] pathForResource:@"ipodnano"
                                   ofType:@"png"];
    NSData *myData = [NSData dataWithContentsOfFile:path];
    [picker addAttachmentData:myData mimeType:@"image/png"
                                   fileName:@"ipodnano"];

    // Fill out the email body text.
    NSString *emailBody = @"It is raining in sunny California!";
    [picker setMessageBody:emailBody isHTML:NO];

    // Present the mail composition interface.
    [self presentModalViewController:picker animated:YES];
    [picker release]; // Can safely release the controller now.
}

// The mail compose view controller delegate method
- (void)mailComposeController:(MFMailComposeViewController *)controller
              didFinishWithResult:(MFMailComposeResult)result
              error:(NSError *)error
{
    [self dismissModalViewControllerAnimated:YES];
}
@end
```

For more information on the standard view controller techniques for displaying interfaces, see *View Controller Programming Guide for iPhone OS*. For information about the classes of the Message UI framework, see *Message UI Framework Reference*.

Application Preferences

In traditional desktop applications, preferences are application-specific settings used to configure the behavior or appearance of an application. iPhone OS also supports application preferences, although not as an integral part of your application. Instead of each application displaying a custom user interface for its preferences, all application-level preferences are displayed using the system-supplied Settings application.

In order to integrate your custom application preferences into the Settings application, you must include a specially formatted settings bundle in the top-level directory of your application bundle. This settings bundle provides information about your application preferences to the Settings application, which is then responsible for displaying those preferences and updating the preferences database with any user-supplied values. At runtime, your application retrieves these preferences using the standard retrieval APIs. The sections that follow describe both the format of the settings bundle and the APIs you use to retrieve your preferences values.

Guidelines for Preferences

Adding your application preferences to the Settings application is most appropriate for productivity-style applications and in situations where you have preference values that are typically configured once and then rarely changed. For example, the Mail application uses these preferences to store the user's account information and message-checking settings. Because the Settings application has support for displaying preferences hierarchically, manipulating your preferences from the Settings application is also more appropriate when you have a large number of preferences. Providing the same set of preferences in your application might require too many screens and might cause confusion for the user.

When your application has only a few options or has options that the user might want to change regularly, you should think carefully about whether the Settings application is the right place for them. For instance, utility applications provide custom configuration options on the back of their main view. A special control on the view flips it over to display the options and another control flips the view back. For simple applications, this type of behavior provides immediate access to the application's options and is much more convenient for the user than going to Settings.

For games and other full-screen applications, you can use the Settings application or implement your own custom screens for preferences. Custom screens are often appropriate in games because those screens are treated as part of the game's setup. You can also use the Settings application for your preferences if you think it is more appropriate for your game flow.

Note: You should never spread your preferences across the Settings application and custom application screens. For example, a utility application with preferences on the back side of its main view should not also have configurable preferences in the Settings application. If you have preferences, pick one solution and use it exclusively.

The Preferences Interface

The Settings application implements a hierarchical set of pages for navigating application preferences. The main page of the Settings application displays the system and third-party applications whose preferences can be customized. Selecting a third-party application takes the user to the preferences for that application.

Each application has at least one page of preferences, referred to as the main page. If your application has only a few preferences, the main page may be the only one you need. If the number of preferences gets too large to fit on the main page, however, you can add more pages. These additional pages become child pages of the main page. The user accesses them by tapping on a special type of preference, which links to the new page.

Each preference you display must be of a specific type. The type of the preference defines how the Settings application displays that preference. Most preference types identify a particular type of control that is used to set the preference value. Some types provide a way to organize preferences, however. Table 9-1 lists the different element types supported by the Settings application and how you might use each type to implement your own preference pages.

Table 9-1 Preference element types

Element Type	Description
Text Field	The text field type displays an optional title and an editable text field. You can use this type for preferences that require the user to specify a custom string value. The key for this type is `PSTextFieldSpecifier`.
Title	The title type displays a read-only string value. You can use this type to display read-only preference values. (If the preference contains cryptic or nonintuitive values, this type lets you map the possible values to custom strings.) The key for this type is `PSTitleValueSpecifier`.
Toggle Switch	The toggle switch type displays an ON/OFF toggle button. You can use this type to configure a preference that can have only one of two values. Although you typically use this type to represent preferences containing Boolean values, you can also use it with preferences containing non-Boolean values. The key for this type is `PSToggleSwitchSpecifier`.
Slider	The slider type displays a slider control. You can use this type for a preference that represents a range of values. The value for this type is a real number whose minimum and maximum you specify. The key for this type is `PSSliderSpecifier`.

Element Type	Description
Multi value	The multi value type lets the user select one value from a list of values. You can use this type for a preference that supports a set of mutually exclusive values. The values can be of any type. The key for this type is `PSMultiValueSpecifier`.
Group	The group type is a way for you to organize groups of preferences on a single page. The group type does not represent a configurable preference. It simply contains a title string that is displayed immediately before one or more configurable preferences. The key for this type is `PSGroupSpecifier`.
Child Pane	The child pane type lets the user navigate to a new page of preferences. You use this type to implement hierarchical preferences. For more information on how you configure and use this preference type, see "Hierarchical Preferences" (page 199). The key for this type is `PSChildPaneSpecifier`.

For detailed information about the format of each preference type, see *Settings Application Schema Reference*. To learn now to create and edit Setting page files, see "Adding and Modifying the Settings Bundle" (page 200).

The Settings Bundle

In iPhone OS, you specify your application's preferences through a special settings bundle. This bundle has the name `Settings.bundle` and resides in the top-level directory of your application's bundle. This bundle contains one or more Settings Page files that provide detailed information about your application's preferences. It may also include other support files needed to display your preferences, such as images or localized strings. Table 9-2 lists the contents of a typical settings bundle.

Table 9-2 Contents of the `Settings.bundle` directory

Item name	Description
`Root.plist`	The Settings Page file containing the preferences for the root page. The contents of this file are described in more detail in "The Settings Page File Format" (page 198).
Additional `.plist` files.	If you build a set of hierarchical preferences using child panes, the contents for each child pane are stored in a separate Settings Page file. You are responsible for naming these files and associating them with the correct child pane.
One or more `.lproj` directories	These directories store localized string resources for your Settings Page files. Each directory contains a single strings file, whose title is specified in your Settings Page. The strings files provide the localized content to display to the user for each of your preferences.
Additional images	If you use the slider control, you can store the images for your slider in the top-level directory of the bundle.

In addition to the settings bundle, your application bundle can contain a custom icon for your application settings. If a file with the name `Icon-Settings.png` is located in the top of your application's bundle directory, that icon is used to identify your application preferences in the Settings application. If no such image file is present, the Settings application uses your application's icon file (`Icon.png` by default) instead, scaling it as necessary. Your `Icon-Settings.png` file should be a 29 x 29 pixel image.

When the Settings application launches, it checks each custom application for the presence of a settings bundle. For each custom bundle it finds, it loads that bundle and displays the corresponding application's name and icon in the Settings main page. When the user taps the row belonging to your application, Settings loads the `Root.plist` Settings Page file for your settings bundle and uses that file to display your application's main page of preferences.

In addition to loading your bundle's `Root.plist` Settings Page file, the Settings application also loads any language-specific resources for that file, as needed. Each Settings Page file can have an associated `.strings` file containing localized values for any user-visible strings. As it prepares your preferences for display, the Settings application looks for string resources in the user's preferred language and substitutes them into your preferences page prior to display.

The Settings Page File Format

Each Settings Page file in your settings bundle is stored in the iPhone Settings property-list file format, which is a structured file format. The simplest way to edit Settings Page files is using Xcode's built in editor facilities; see "Preparing the Settings Page for Editing" (page 201). You can also edit property-list files using the Property List Editor application that comes with the Xcode tools.

Note: Xcode automatically converts any XML-based property files in your project to binary format when building your application. This conversion saves space and is done for you automatically at build time.

The root element of each Settings Page file contains the keys listed in Table 9-3. Only one key is actually required, but it is recommended that you include both of them.

Table 9-3 Root-level keys of a preferences Settings Page file

Key	Type	Value
`PreferenceSpecifiers` (required)	Array	The value for this key is an array of dictionaries, with each dictionary containing the information for a single preference element. For a list of element types, see Table 9-1 (page 196). For a description of the keys associated with each element type, see *Settings Application Schema Reference*.
`StringsTable`	String	The name of the strings file associated with this file. A copy of this file (with appropriate localized strings) should be located in each of your bundle's language-specific project directories. If you do not include this key, the strings in this file are not localized. For information on how these strings are used, see "Localized Resources" (page 200).

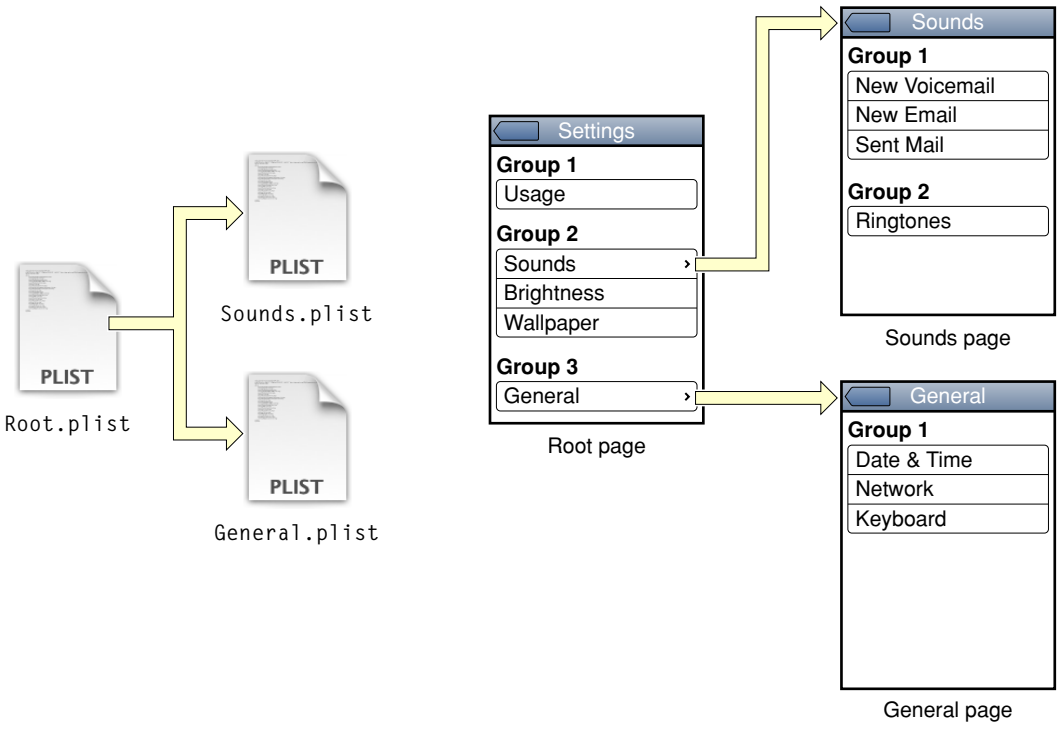
Hierarchical Preferences

If you plan to organize your preferences hierarchically, each page you define must have its own separate `.plist` file. Each `.plist` file contains the set of preferences displayed only on that page. Your application's main preferences page is always stored in the `Root.plist` file. Additional pages can be given any name you like.

To specify a link between a parent page and a child page, you include a child pane element in the parent page. A child pane element creates a row that, when tapped, displays a new page of settings. The `File` key of the child pane element identifies the name of the `.plist` file that defines the contents of the child page. The `Title` key identifies the title of the child page; this title is also used as the text of the row that the user taps to display the child page. The Settings application automatically provides navigation controls on the child page to allow the user to navigate back to the parent page.

Figure 9-1 shows how this hierarchical set of pages works. The left side of the figure shows the `.plist` files, and the right side shows the relationships between the corresponding pages.

Figure 9-1 Organizing preferences using child panes



For more information about child pane elements and their associated keys, see *Settings Application Schema Reference*.

Localized Resources

Because preferences contain user-visible strings, you should provide localized versions of those strings with your settings bundle. Each page of preferences can have an associated `.strings` file for each localization supported by your bundle. When the Settings application encounters a key that supports localization, it checks the appropriately localized `.strings` file for a matching key. If it finds one, it displays the value associated with that key.

When looking for localized resources such as `.strings` files, the Settings application follows the same rules that Mac OS X applications do. It first tries to find a localized version of the resource that matches the user's preferred language setting. If a resource does not exist for the user's preferred language, an appropriate fallback language is selected.

For information about the format of strings files, language-specific project directories, and how language-specific resources are retrieved from bundles, see *Internationalization Programming Topics*.

Adding and Modifying the Settings Bundle

Xcode provides a template for adding a Settings bundle to your current project. The default settings bundle contains a `Root.plist` file and a default language directory for storing any localized resources. You can then expand this bundle to include additional property list files and resources needed by your Settings bundle.

Adding the Settings Bundle

To add a settings bundle to your Xcode project:

1. Choose File > New File.
2. Choose the iPhone OS > Settings > Settings Bundle template.
3. Name the file `Settings.bundle`.

In addition to adding a new Settings bundle to your project, Xcode automatically adds that bundle to the Copy Bundle Resources build phase of your application target. Thus, all you have to do is modify the property list files of your Settings bundle and add any needed resources.

The newly added `Settings.bundle` bundle has the following structure:

```
Settings.bundle/
    Root.plist
    en.lproj/
        Root.strings
```

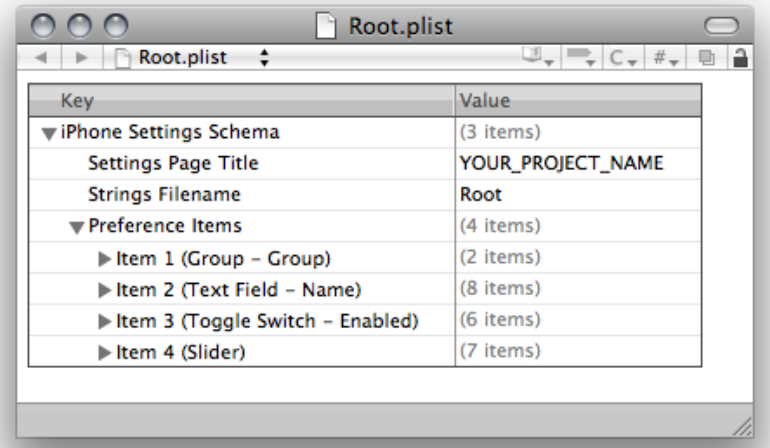
Preparing the Settings Page for Editing

After creating your Settings bundle using the Settings Bundle template, you can format the contents of your schema files to make them easier to edit. The following steps show you how to do this for the `Root.plist` file of your Settings bundle but the steps are the same for any other schema files you create.

1. Display the contents of the `Root.plist` file of your Settings bundle.
    a. In the Groups & Files list, disclose `Settings.bundle` to view its contents.
    b. Select the `Root.plist` file. Its contents appear in the Detail view.
2. In the Detail view, select the Root key of the `Root.plist` file.
3. Choose View > Property List Type > iPhone Settings plist.

    This command formats the contents of the property list inside the Detail view. Instead of showing the property list key names and values, Xcode substitutes human-readable strings (as shown in Figure 9-2) to make it easier to understand and edit the file's contents.

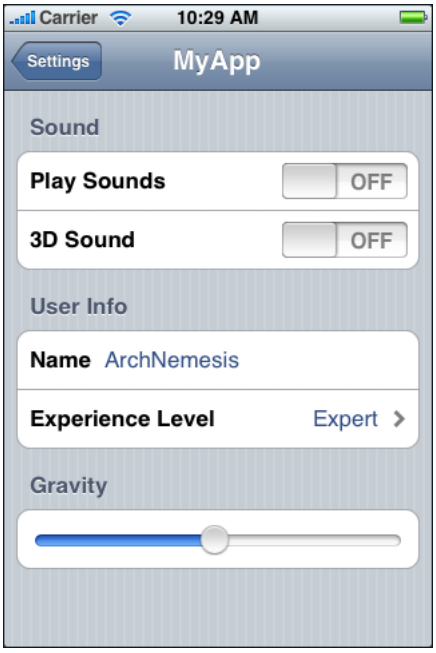
Figure 9-2 Formatted contents of the `Root.plist` file



Configuring a Settings Page: A Tutorial

This section contains a tutorial that shows you how to configure a Settings page to display the controls you want. The goal of the tutorial is to create a page like the one in Figure 9-2. If you have not yet created a Settings bundle for your project, you should do so as described in "Preparing the Settings Page for Editing" (page 201) before proceeding with these steps.

Figure 9-3 A root Settings page



1. Change the value of the Settings Page Title key to the name of your application.

   Double-click the `YOUR_PROJECT_NAME` text and change the text to `MyApp`.

2. Disclose the Preference Items key to display the default items that come with the template.

3. Change the title of `Item 1` to `Sound`:

   - Disclose `Item 1` of `Preference Items`.
   - Change the value of the `Title` key from `Group` to `Sound`.
   - Leave the `Type` key set to `Group`.

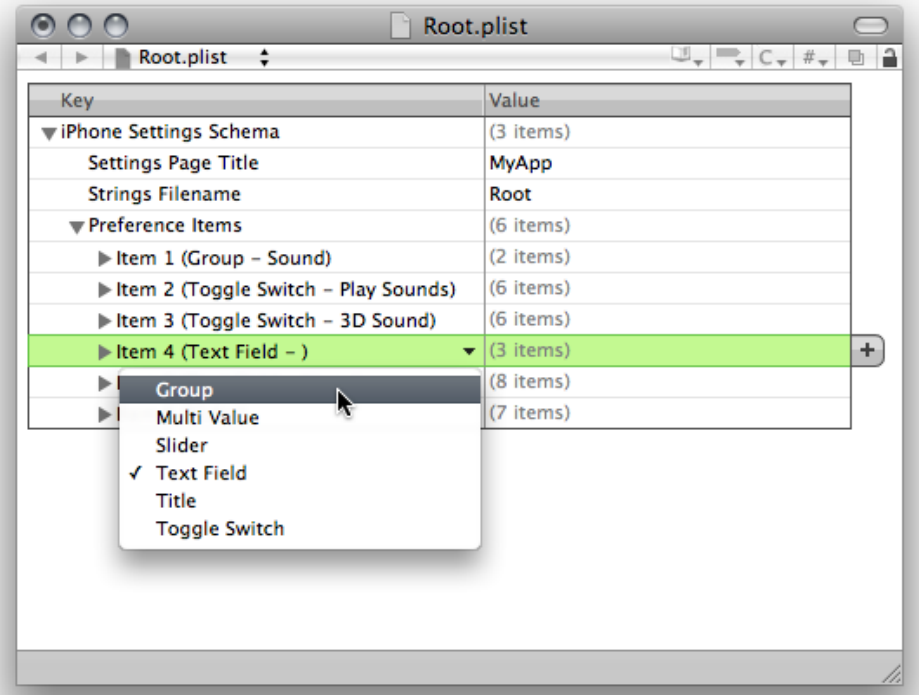
4. Create the first toggle switch for the newly renamed Sound group.

   - Select `Item 3` of `Preference Items` and choose Edit > Cut.
   - Select `Item 1` and choose Edit > Paste. (This moves the toggle switch item in front of the text field item.)
   - Disclose the toggle switch item to reveal its configuration keys.
   - Change the value of the `Title` key to `Play Sounds`.
   - Change the value of the `Identifier` key to `play_sounds_preference`. The item should now be configured as shown in the following figure.

Item 2 (Toggle Switch – Play Sounds)	(6 items)
Type	Toggle Switch
Title	Play Sounds
Identifier	play_sounds_preference
Default Value	☑
Value for ON	YES
Value for OFF	NO

5. Create a second toggle switch for the Sound group.
    - Select `Item 2` (the Play Sounds toggle switch).
    - Select Edit > Copy.
    - Select Edit >Paste. This places a copy of the toggle switch right after the first one.
    - Disclose the new toggle switch item to reveal its configuration keys.
    - Change the value of its `Title` key to `3D Sound`.
    - Change the value of its `Identifier` key to `3D_sound_preference`.

    At this point, you have finished the first group of settings and are ready to create the User Info group.
6. Change `Item 4` into a Group element and name it `User Info`.
    - Click `Item 4` in the `Preferences Items`. This displays a drop-down menu with a list of item types.
    - From the drop-down menu, select `Group` to change the type of the element.



- Disclose the contents of `Item 4`.
- Set the value of the `Title` key to `User Info`.

7. Create the Name field.

   - Select `Item 5` in the Preferences Item.
   - Using the drop-down menu, change its type to `Text Field`.
   - Set the value of the `Title` key to `User Info`.
   - Set value of the `Identifier` key to `user_name`.
   - Toggle the disclosure button of the item to hide its contents.

8. Create the Experience Level settings.

   - Select `Item 5` and click the plus (+) button (or press Return) to create a new item.
   - Click the new item and set its type to `Multi Value`.
   - Disclose the items contents and set its title to `Experience Level`, its identifier to `experience_preference`, and its default value to `0`.
   - With the Default Value key selected, click the plus button to add a `Titles` array.
   - Open the disclosure button for the `Titles` array and click the items button along the right edge of the table. Clicking this button adds a new subitem to `Titles`.

- Select the new subitem and click the plus button 2 more times to create 3 total subitems.
- Set the values of the subitems to `Beginner`, `Expert`, and `Master`.
- Select the `Titles` key again and click its disclosure button to hide its subitems.
- Click the plus button to create the `Values` array.
- Add 3 subitems to the `Values` array and set their values to `0`, `1`, and `2`.
- Click the disclosure button of `Item 6` to hide its contents.

9. Add the final group to your settings page.
    - Create a new item and set its type to `Group` and its title to `Gravity`.
    - Create another new item and set its type to `Slider`, its identifier to `gravity_preference`, its default value to `1`, and its maximum value to `2`.

Creating Additional Settings Page Files

The Settings Bundle template includes the `Root.plist` file, which defines your application's top Settings page. To define additional Settings pages, you must add additional property list files to your Settings bundle. You can do this either from the Finder or from Xcode.

To add a property list file to your Settings bundle in Xcode, do the following:

1. In the Groups and Files pane, open your Settings bundle and select the `Root.plist` file.
2. Choose File > New.
3. Choose Other > Property List.
4. Select the new file and choose View > Property List Type > iPhone Settings plist to configure it as a settings file.

After adding a new Settings page to your Settings bundle, you can edit the page's contents as described in "Configuring a Settings Page: A Tutorial" (page 201). To display the settings for your page, you must reference it from a Child Pane element as described in "Hierarchical Preferences" (page 199).

Accessing Your Preferences

iPhone applications get and set preferences values using either the Foundation and Core Foundation frameworks. In the Foundation framework, you use the `NSUserDefaults` class to get and set preference values. In the Core Foundation framework, you use several preferences-related functions to get and set values.

Listing 9-1 shows a simple example of how to read a preference value from your application. This example uses the `NSUserDefaults` class to read a value from the preferences created in "Configuring a Settings Page: A Tutorial" (page 201) and assign it to an application-specific instance variable.

Listing 9-1 Accessing preference values in an application

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [self setShouldPlaySounds:[defaults boolForKey:@"play_sounds_preference"]];

    // Finish app initialization...
}
```

For information about the `NSUserDefaults` methods used to read and write preferences, see *NSUserDefaults Class Reference*. For information about the Core Foundation functions used to read and write preferences, see *Preferences Utilities Reference*.

Debugging Preferences for Simulated Applications

When running your application, the iPhone Simulator stores any preferences values for your application in `~/Library/Application Support/iPhone Simulator/User/Applications/`<*APP_ID*>`/Library/Preferences`, where <*APP_ID*> is a programmatically generated directory name that iPhone OS uses to identify your application.

Each time you reinstall your application, iPhone OS performs a clean install, which deletes any previous preferences. In other words, building or running your application from Xcode always installs a new version, replacing any old contents. To test preference changes between successive executions, you must run your application directly from the simulator interface and not from Xcode.

Document Revision History

This table describes the changes to *iPhone Application Programming Guide*.

Date	Notes
2010-03-24	Added warnings about thread safety for UIKit classes.
2010-02-24	Made minor corrections.
2010-01-20	Updated the "Multimedia Support" (page 145) chapter with improved descriptions of audio formats and codecs.
2009-10-19	Moved the iPhone specific `Info.plist` keys to *Information Property List Key Reference*.
	Updated the "Multimedia Support" (page 145) chapter for iPhone OS 3.1.
2009-06-17	Added information about using the compass interfaces.
	Moved information about OpenGL support to *OpenGL ES Programming Guide for iPhone OS*.
	Updated the list of supported `Info.plist` keys.
2009-05-14	Updated for iPhone OS 3.0.
	Added code examples to "Copy and Paste Operations" in the Event Handling chapter.
	Added a section on keychain data to the Files and Networking chapter.
	Added information about how to display map and email interfaces.
	Made various small corrections.
2009-01-06	Fixed several typos and clarified the creation process for child pages in the Settings application.
2008-11-12	Added guidance about floating-point math considerations
	Updated information related to what is backed up by iTunes.
2008-10-15	Reorganized the contents of the book.
	Moved the high-level iPhone OS information to *iPhone OS Technology Overview*.
	Moved information about the standard system URL schemes to *Apple URL Scheme Reference*.

Date	Notes
	Moved information about the development tools and how to configure devices to *iPhone Development Guide*.
	Created the Core Application chapter, which now introduces the application architecture and covers much of the guidance for creating iPhone applications.
	Added a Text and Web chapter to cover the use of text and web classes and the manipulation of the onscreen keyboard.
	Created a separate chapter for Files and Networking and moved existing information into it.
	Changed the title from *iPhone OS Programming Guide*.
2008-07-08	New document that describes iPhone OS and the development process for iPhone applications.