iOS App Programming Guide

Developer

Contents

Figures, Tables, and Listings

About iOS App Programming

This document is the starting point for creating iOS apps. It describes the fundamental architecture of iOS apps, including how the code you write fits together with the code provided by iOS. This document also offers practical guidance to help you make better choices during your design and planning phase and guides you to the other documents in the iOS developer library that contain more detailed information about how to address a specific task.

The contents of this document apply to all iOS apps running on all types of iOS devices, including iPad, iPhone, and iPod touch.

> **Note:** Development of iOS apps requires an Intel-based Macintosh computer with the iOS SDK installed. For information about how to get the iOS SDK, go to the iOS Dev Center.

At a Glance

The starting point for any new app is identifying the design choices you need to make and understanding how those choices map to an appropriate implementation.

Translate Your Initial Idea into an Implementation Plan

Every great iOS app starts with a great idea, but translating that idea into actions requires some planning. Every iOS app relies heavily on design patterns, and those design patterns influence much of the code you need to write. So before you write any code, take the time to explore the possible techniques and technologies available for writing that code. Doing so can save you a lot of time and frustration.

> **Relevant Chapter:** "App Design Basics" (page 12)

UIKit Provides the Core of Your App

The core infrastructure of an iOS app is built from objects in the UIKit framework. The objects in this framework provide all of the support for handling events, displaying content on the screen, and interacting with the rest of the system. Understanding the role these objects play, and how you modify them to customize the default app behavior, is therefore very important for writing apps quickly and correctly.

Relevant Chapter: "Core App Objects" (page 17)

Apps Must Behave Differently in the Foreground and Background

An iOS device runs multiple apps simultaneously but only one app—the foreground app—has the user's attention at any given time. The current foreground app is the only app allowed to present a user interface and respond to touch events. Other apps remain in the background, usually asleep but sometimes running additional code. Transitioning between the foreground and background states involves changing several aspects of your app's behavior.

Relevant Chapter: "App States and Multitasking" (page 33)

iCloud Affects the Design of Your Data Model and UI Layers

iCloud allows you to share the user's data among multiple instances of your app running on different iOS and Mac OS X devices. Incorporating support for iCloud into your app involves changing many aspects of how you manage your files. Because files in iCloud are accessible by more than just your app, all file operations must be synchronized to prevent data corruption. And depending on your app and how it presents its data, iCloud can also require changes to portions of your user interface.

Relevant Chapter: "Integrating iCloud Support Into Your App" (page 26)

Apps Require Some Specific Resources

There are some resources that must be present in all iOS apps. Most apps include images, sounds, and other types of resources for presenting the app's content but the App Store also requires some specific resources be present. The reason is that iOS uses several specific resources when presenting your app to the user and when coordinating interactions with other parts of the system. So these resources are there to improve the overall user experience.

Relevant Chapter: "App-Related Resources" (page 93)

Apps Should Restore Their Previous UI State at Launch Time

At launch time, your app should restore its user interface to the state it was in when it was last used. During normal use, the system controls when apps are terminated. Normally when this happens, the app displays its default user interface when it is relaunched. With state restoration, UIKit helps your app restore your app's interface to its previous state, which promotes a consistent user experience.

Relevant Chapter: "State Preservation and Restoration" (page 67)

Many App Behaviors Can Be Customized

The core architecture of all apps may be the same, but there are still ways for you to tweak the high-level design of your app. Some of these tweaks are how you add specific high-level features, such as data protection and URL handling. Others affect the design of specific types of apps, such as VoIP apps.

Relevant Chapter: "Advanced App Tricks" (page 108)

Apps Must Be Tuned for Performance

Great apps are always tuned for the best possible performance. For iOS apps, performance means more than just writing fast code. It often means writing better code so that your user interface remains responsive to user input, your app does not degrade battery life significantly, and your app does not impact other system resources. Before you can tune your code, though, learn about the types of changes that are likely to provide the most benefit.

Relevant Chapter: "Performance Tuning" (page 127)

The iOS Environment Affects Many App Behaviors

There are aspects of iOS itself that impact how you design and write applications. Because iOS is built for mobile devices, it takes a more active role in providing security for apps. Other system behaviors also affect everything from how memory is managed to how the system responds to hardware input. All of these system behaviors affect the way you design your apps.

Relevant Appendix: "The iOS Environment" (page 137)

How to Use This Document

This document provides important information about the core objects of your app and how they work together. This document does not address the creation of any specific type of iOS app. Instead, it provides a tour of the architecture that is common to all iOS apps and highlights key places where you can modify that architecture to meet your needs. Whenever possible, the document also offers tips and guidance about ways to implement features related to the core app architecture.

Prerequisites

This document is the entry-point guide for designing an iOS app. This guide also covers many of the practical aspects involved with implementing your app. However, this book assumes that you have already installed the iOS SDK and configured your development environment. You must perform those steps before you can start writing and building iOS apps.

If you are new to iOS app development, read *Start Developing iOS Apps Today*. This document offers a step-by-step introduction to the development process to help you get up to speed quickly. It also includes a hands-on tutorial that walks you through the app-creation process from start to finish, showing you how to create a simple app and get it running quickly.

See Also

For additional information related to app design, see the following documents:

- For guidance about how to design an iOS app, read *iOS Human Interface Guidelines*. This book provides you with tips and guidance about how to create a great experience for users of your app. It also conveys the basic design philosophy surrounding iOS apps.
- If you are not sure what is possible in an iOS app, read *iOS Technology Overview*. This book provides a summary of iOS technologies and the situations where you might want to use them. This book is not required reading but is a good reference during the brainstorming phase of your project.

App Design Basics

If you are a new to developing iOS apps, you might be wondering where the app development process starts. After devising your initial idea for an app, you need to turn that idea into an action plan for implementing your app. From a design perspective, you need to make some high-level decisions about the best course of action for implementing your ideas. You also need to set up your initial Xcode project in a way that makes it easy to proceed with development.

If you are new to developing iOS apps altogether, spend some time familiarizing yourself with the basic concepts. There are tutorials to help you jump right in if you want to start writing code, but iOS is a system built from basic design patterns. Taking a little bit of time to learn those patterns will help you tremendously later.

Doing Your Initial Design

There are many ways to design an app, and many of the best approaches do not involve writing any code. A great app starts with a great idea that you then expand into a more full-featured product description. Early in the design phase, it helps to understand just what you want your app to do. Write down the set of high-level features that would be required to implement your idea. Prioritize those features based on what you think your users will need. Do a little research into iOS itself so that you understand its capabilities and how you might be able to use them to achieve your goals. And sketch out some rough interface designs on paper to visualize how your app might look.

The goal of your initial design is to answer some very important questions about your app. The set of features and the rough design of your interface help you think about what will be required later when you start writing code. At some point, you need to translate the information displayed by your app into a set of data objects. Similarly, the look of your app has an overwhelming influence on the choices you must make when implementing your user interface code. Doing your initial design on paper (as opposed to on the computer) gives you the freedom to come up with answers that are not limited by what is easy to do.

Of course, the most important thing you can do before starting your initial design is read *iOS Human Interface Guidelines*. That book describes several strategies for doing your initial design. It also offers tips and guidance about how to create apps that work well in iOS. You might also read *iOS Technology Overview* to understand how the capabilities of iOS and how you might use those capabilities to achieve your design goals.

Learning the Fundamental iOS Design Patterns and Techniques

No matter what type of app you are creating, there are a few fundamental design patterns and techniques that you must know before you start writing code. In iOS, the system frameworks provide critical infrastructure for your app and in most cases are the only way to access the underlying hardware. In turn, the frameworks use many specific design patterns and assume that you are familiar with them. Understanding these design patterns is therefore an important first step to understanding how the system can help you develop your app.

The most important design patterns you must know are:

- Model-View-Controller—This design pattern governs the overall structure of your app.
- Delegation—This design pattern facilitates the transfer information and data from one object to another.
- Target-action—This design pattern translates user interactions with buttons and controls into code that your app can execute.
- Block objects—You use blocks to implement callbacks and asynchronous code.
- Sandboxing—All iOS apps are placed in sandboxes to protect the system and other apps. The structure of the sandbox affects the placement of your app's files and has implications for data backups and some app-related features.

Accurate and efficient memory management is important for iOS apps. Because iOS apps typically have less usable memory than a comparable desktop computer, apps need to be aggressive about deleting unneeded objects and be lazy about creating objects in the first place. Apps use the compiler's **Automatic Reference Counting (ARC)** feature to manage memory efficiently. Although using ARC is not required, it is highly recommended. The alternative is to manage memory yourself by explicitly retaining and releasing objects.

There are other design patterns that you might see used occasionally or use yourself in your own code. For a complete overview of the design patterns and techniques you will use to create iOS apps, see *Start Developing iOS Apps Today*.

Translating Your Initial Design into an Action Plan

iOS assumes that all apps are built using the Model-View-Controller design pattern. Therefore, the first step you can take toward achieving this goal is to choose an approach for the data and view portions of your app.

- Choose a basic approach for your data model:
  - **Existing data model code**—If you already have data model code written in a C-based language, you can integrate that code directly into your iOS apps. Because iOS apps are written in Objective-C, they work just fine with code written in other C-based languages. Of course, there is also benefit to writing an Objective-C wrapper for any non Objective-C code.

- **Custom objects data model**—A custom object typically combines some simple data (strings, numbers, dates, URLs, and so on) with the business logic needed to manage that data and ensure its consistency. Custom objects can store a combination of scalar values and pointers to other objects. For example, the Foundation framework defines classes for many simple data types and for storing collections of other objects. These classes make it much easier to define your own custom objects.
- **Structured data model**—If your data is highly structured—that is, it lends itself to storage in a database—use Core Data (or SQLite) to store the data. Core Data provides a simple object-oriented model for managing your structured data. It also provides built-in support for some advanced features like undo and iCloud. (SQLite files cannot be used in conjunction with iCloud.)

- Decide whether you need support for documents:

  The job of a document is to manage your app's in-memory data model objects and coordinate the storage of that data in a corresponding file (or set of files) on disk. Documents normally connote files that the user created but apps can use documents to manage non user facing files too. One big advantage of using documents is that the `UIDocument` class makes interacting with iCloud and the local file system much simpler. For apps that use Core Data to store their content, the `UIManagedDocument` class provides similar support.

- Choosing an approach for your user interface:
  - **Building block approach**—The easiest way to create your user interface is to assemble it using existing view objects. Views represent visual elements such as tables, buttons, text fields, and so on. You use many views as-is but you can also customize the appearance and behavior of standard views as needed to meet your needs. You can also implement new visual elements using custom views and mix those views freely with the standard views in your interface. The advantages of views are that they provide a consistent user experience and they allow you to define complex interfaces quickly and with relatively little code.
  - **OpenGL ES-based approach**—If your app requires frequent screen updates or sophisticated rendering, you probably need to draw that content directly using OpenGL ES. The main use of OpenGL ES is for games and apps that rely heavily on sophisticated graphics, and therefore need the best performance possible.

Starting the App Creation Process

After you formulate your action plan, it is time to start coding. If you are new to writing iOS apps, it is good to take some time to explore the initial Xcode templates that are provided for development. These templates greatly simplify the work you have to do and make it possible to have an app up and running in minutes. These templates also allow you to customize your initial project to support your specific needs more precisely. To that end, when creating your Xcode project, you should already have answers to the following questions in mind:

- **What is the basic interface-style of your app?** Different types of app require different sets of initial views and view controllers. Knowing how you plan to organize your user interface lets you select an initial project template that is most suited to your needs. You can always change your user interface later, but choosing the most appropriate template first makes starting your project much easier.
- **Do you want to create a Universal app or one targeted specifically for iPad or iPhone?** Creating a universal app requires specifying different sets of views and view controllers for iPad and iPhone and dynamically selecting the appropriate set at runtime. Universal apps are preferred because they support more iOS devices but do require you to factor your code better for each platform. For information about how a universal app affects the code you write, see "Creating a Universal App" (page 108).
- **Do you want your app to use storyboards?** Storyboards simplify the design process by showing both the views and view controllers of your user interface and the transitions between them. Storyboards are supported in iOS 5 and later and are enabled by default for new projects. If your app must run on earlier versions of iOS, though, you cannot use storyboards and should continue to use nib files.
- **Do you want to use Core Data for your data model?** Some types of apps lend themselves naturally to a structured data model, which makes them ideal candidates for using Core Data. For more information about Core Data and the advantages it offers, see *Core Data Programming Guide*.

From these questions, you can use Xcode to create your initial project files and start coding.

1. If you have not yet installed Xcode, do so and configure your iOS development team. For detailed information about setting up your development teams and and preparing your Xcode environment, see *Developing for the App Store*.
2. Create your initial Xcode project.
3. Before writing any code, build and run your new Xcode project. Target your app for iOS Simulator so that you can see it run.

   Every new Xcode project starts you with a fully functional (albeit featureless) app. The app itself should run and display the default views found in the main storyboard or nib file, which are probably not very interesting. The reason that the app runs at all, though, is because of the infrastructure provided to you by UIKit. This infrastructure initializes the app, loads the initial interface file, and checks the app in with the system so that it can start handling events. For more information about this infrastructure and the capabilities it provides, see "The Core Objects of Your App" (page 17) and "The App Launch Cycle" (page 36).
4. Start writing your app's primary code.

   For new apps, you probably want to start creating the classes associated with your app's data model first. These classes usually have no dependencies on other parts of your app and should be something you can work on initially. For information about ways to build your data model, see "The Data Model" (page 20).

You might also want to start playing around with designs for your user interface by adding views to your main storyboard or nib file. From these views, you can also start identifying the places in your code where you need to respond to interface-related changes. For an overview of user interfaces and where they fit into your app's code, see "The User Interface" (page 26).

If your app supports iCloud, you should incorporate support for iCloud into your classes at an early stage. For information about adding iCloud support to your app, see "Integrating iCloud Support Into Your App" (page 26).

5. Add support for app state changes.

   In iOS, the state of an app determines what it is allowed to do and when. App states are managed by high-level objects in your app but can affect many other objects as well. Therefore, you need to consider how the current app state affects your data model and view code and update that code appropriately. For information about app states and how apps run in the foreground and background, see "App States and Multitasking" (page 33)

6. Create the resources needed to support your app.

   Apps submitted to the App Store are expected to have specific resources such as icons and launch images to make the overall user experience better. Well-factored apps also make heavy use of resource files to keep their code separate from the data that code manipulates. This factoring makes it much easier to localize your app, tweak its appearance, and perform other tasks without rewriting any code. For information about the types of resources found in a typical iOS app and how they are used, see "The App Bundle" (page 30) and "App-Related Resources" (page 93).

7. As needed, implement any app-specific behaviors that are relevant for your app.

   There are many ways to modify the way your app launches or interacts with the system. For information about the most common types of app customizations, see "Advanced App Tricks" (page 108).

8. Add the advanced features that make your app unique.

   iOS includes many other frameworks for managing multimedia, advanced rendering, game content, maps, contacts, location tracking, and many other advanced features. For an overview of the frameworks and features you can incorporate into your apps, see *iOS Technology Overview*.

9. Do some basic performance tuning for your app.

   All iOS apps should be tuned for the best possible performance. Tuned apps run faster but also use system resources, such as memory and battery life, more efficiently. For information about areas to focus on during the tuning process, see "Performance Tuning" (page 127).

10. Iterate.

    App development is an iterative process. As you add new features, you might need to revisit some or all of the preceding steps to make adjustments to your existing code.

Core App Objects

UIKit provides the infrastructure for all apps but it is your custom objects that define the specific behavior of your app. Your app consists of a handful of specific UIKit objects that manage the event loop and the primary interactions with iOS. Through a combination of subclassing, delegation, and other techniques, you modify the default behaviors defined by UIKit to implement your app.

In addition to customizing the UIKit objects, you are also responsible for providing or defining other key sets of objects. The largest set of objects is your app's data objects, the definition of which is entirely your responsibility. You must also provide a set of user interface objects, but fortunately UIKit provides numerous classes to make defining your interface easy. In addition to code, you must also provide the resources and data files you need to deliver a shippable app.

The Core Objects of Your App

From the time your app is launched by the user, to the time it exits, the UIKit framework manages much of the app's core behavior. At the heart of the app is the `UIApplication` object, which receives events from the system and dispatches them to your custom code for handling. Other UIKit classes play a part in managing your app's behavior too, and all of these classes have similar ways of calling your custom code to handle the details.

To understand how UIKit objects work with your custom code, it helps to understand a little about the objects make up an iOS app. Figure 2-1 shows the objects that are most commonly found in an iOS app, and Table 2-1 describes the roles of each object. As you can see from the diagram, iOS apps are organized around the model-view-controller design pattern. This pattern separates the data objects in the model from the views used

to present that data. This separation promotes code reuse by making it possible to swap out your views as needed and is especially useful when creating universal apps—that is, apps that can run on both iPad and iPhone.

Figure 2-1 Key objects in an iOS app

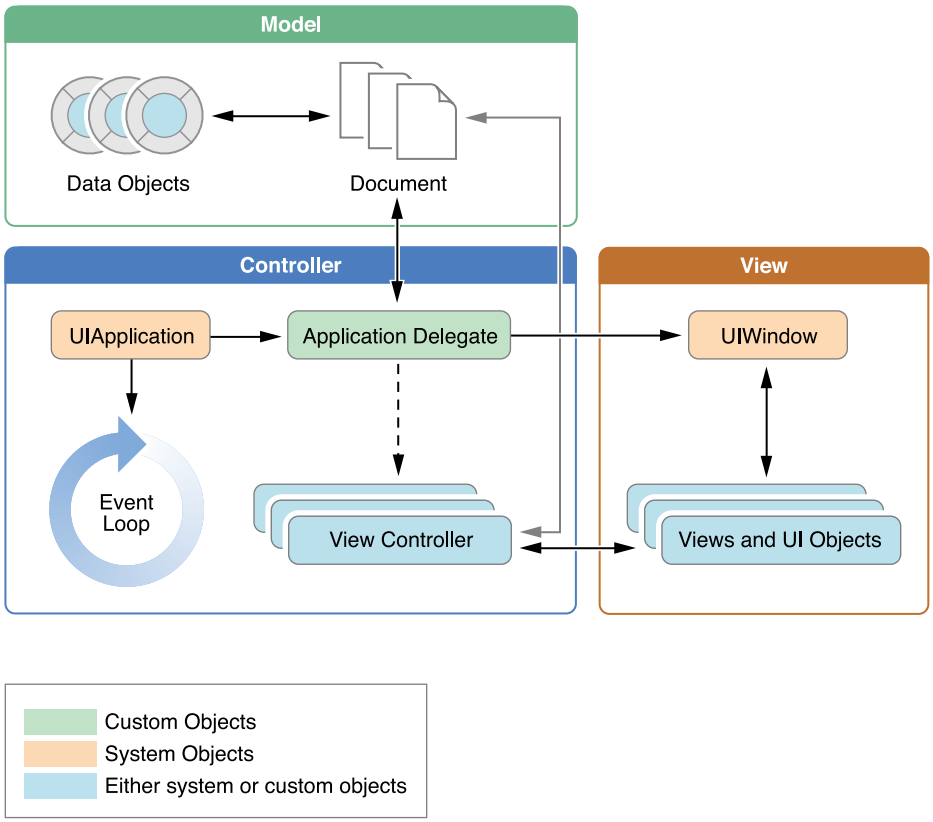


Table 2-1 The role of objects in an iOS app

Object	Description
`UIApplication` object	You use the `UIApplication` object essentially as is—that is, without subclassing. This controller object manages the app event loop and coordinates other high-level app behaviors. Your own custom app-level logic resides in your app delegate object, which works in tandem with this object.

Object	Description
App delegate object	The **app delegate** is a custom object created at app launch time, usually by the `UIApplicationMain` function. The primary job of this object is to handle state transitions within the app. For example, this object is responsible for launch-time initialization and handling transitions to and from the background. For information about how you use the app delegate to manage state transitions, see "Managing App State Changes" (page 34). In iOS 5 and later, you can use the app delegate to handle other app-related events. The Xcode project templates declare the app delegate as a subclass of `UIResponder`. If the `UIApplication` object does not handle an event, it dispatches the event to your app delegate for processing. For more information about the types of events you can handle, see *UIResponder Class Reference*.
Documents and data model objects	**Data model objects** store your app's content and are specific to your app. For example, a banking app might store a database containing financial transactions, whereas a painting app might store an image object or even the sequence of drawing commands that led to the creation of that image. (In the latter case, an image object is still a data object because it is just a container for the image data.) Apps can also use **document objects** (custom subclasses of `UIDocument`) to manage some or all of their data model objects. Document objects are not required but offer a convenient way to group data that belongs in a single file or file package. For more information about documents, see "Defining a Document-Based Data Model" (page 24).
View controller objects	**View controller objects** manage the presentation of your app's content on screen. A view controller manages a single view and its collection of subviews. When presented, the view controller makes its views visible by installing them in the app's window. The `UIViewController` class is the base class for all view controller objects. It provides default functionality for loading views, presenting them, rotating them in response to device rotations, and several other standard system behaviors. UIKit and other frameworks define additional view controller classes to implement standard system interfaces such as the image picker, tab bar interface, and navigation interface. For detailed information about how to use view controllers, see *View Controller Programming Guide for iOS*.

Object	Description
`UIWindow` object	A `UIWindow` object coordinates the presentation of one or more views on a screen. Most apps have only one window, which presents content on the main screen, but apps may have an additional window for content displayed on an external display. To change the content of your app, you use a view controller to change the views displayed in the corresponding window. You never replace the window itself. In addition to hosting views, windows work with the `UIApplication` object to deliver events to your views and view controllers.
View, control, and layer objects	Views and controls provide the visual representation of your app's content. A **view** is an object that draws content in a designated rectangular area and responds to events within that area. **Controls** are a specialized type of view responsible for implementing familiar interface objects such as buttons, text fields, and toggle switches. The UIKit framework provides standard views for presenting many different types of content. You can also define your own custom views by subclassing `UIView` (or its descendants) directly. In addition to incorporating views and controls, apps can also incorporate Core Animation layers into their view and control hierarchies. **Layer objects** are actually data objects that represent visual content. Views use layer objects intensively behind the scenes to render their content. You can also add custom layer objects to your interface to implement complex animations and other types of sophisticated visual effects.

What distinguishes one iOS app from another is the data it manages (and the corresponding business logic) and how it presents that data to the user. Most interactions with UIKit objects do not define your app but help you to refine its behavior. For example, the methods of your app delegate let you know when the app is changing states so that your custom code can respond appropriately.

For information about the specific behaviors of a given class, see the corresponding class reference. For more information about how events flow in your app and information about your app's responsibilities at various points during that flow, see "App States and Multitasking" (page 33).

The Data Model

Your app's data model comprises your data structures and the business logic needed to keep that data in a consistent state. You never want to design your data model in total isolation from your app's user interface; however, the implementation of your data model objects should be separate and not rely on the presence of

specific views or view controllers. Keeping your data separate from your user interface makes it easier to implement a universal app—one that can run on both iPad and iPhone—and also makes it easier to reuse portions of your code later.

If you have not yet defined your data model, the iOS frameworks provide help for doing so. The following sections highlight some of the technologies you can use when defining specific types of data models.

Defining a Custom Data Model

When defining a custom data model, create custom objects to represent any high-level constructs but take advantage of the system-supplied objects for simpler data types. The Foundation framework provides many objects (most of which are listed in Table 2-2) for managing strings, numbers, and other types of simple data in an object-oriented way. Using these objects is preferable to defining new objects both because it saves time and because many other system routines expect you to use the built-in objects anyway.

Table 2-2 Data classes in the Foundation framework

Data	Classes	Description
Strings and text	`NSString` (`NSMutableString`) `NSAttributedString` (`NSMutableAttributed-String`)	Strings in iOS are Unicode based. The string classes provide support for creating and manipulating strings in a variety of ways. The attributed string classes support stylized text and are used only in conjunction with Core Text.

Numbers	`NSNumber`
	`NSDecimalNumber`
	`NSIndexPath`
		When you want to store numerical values in a collection, use number objects. The `NSNumber` class can represent integer, floating-point values, Booleans, and `char` types. The `NSIndexPath` class stores a sequence of numbers and is often used to specify multi-layer selections in hierarchical lists.

Raw bytes	`NSData` (`NSMutableData`) `NSValue`	For times when you need to store raw streams of bytes, use data objects. Data objects are also commonly used to store objects in an archived form. The `NSValue` class is typically extended (using categories) and used to archive common data types such as points and rectangles.
Dates and times	`NSDate` `NSDateComponents`	Use date objects to store timestamps, calendar dates, and other time-related information.

Data	Classes	Description
URLs	`NSURL`	In addition to their traditional use for referring to network resources, URLs in iOS are the preferred way to store paths to files. The `NSURL` class even provides support for getting and setting file-related attributes.
Collections	`NSArray` (`NSMutableArray`) `NSDictionary` (`NSMutableDictionary`) `NSIndexSet` (`NSMutableIndexSet`) `NSOrderedSet` (`NSMutableOrderedSet`) `NSSet` (`NSMutableSet`)	Use collections to group related objects together in a single place. The Foundation framework provides several different types of collection classes

In addition to data-related objects, there are some other data types that are commonly used by the iOS frameworks to manage familiar types of data. You are encouraged to use these data types in your own custom objects to represent similar types of data.

- `NSInteger`/`NSUInteger`—Abstractions for scalar signed and unsigned integers that define the integer size based on the architecture.
- `NSRange`—A structure used to define a contiguous portion of a series. For example, you can use ranges to define the selected characters in a string.
- `NSTimeInterval`—The number of seconds (whole and partial) in a given time interval.
- `CGPoint`—An x and y coordinate value that defines a location.
- `CGSize`—Coordinate values that define a set of horizontal and vertical extents.
- `CGRect`—Coordinate values that define a rectangular region.

Of course, when defining custom objects, you can always incorporate scalar values directly into your class implementations. In fact, a custom data object can include a mixture of scalar and object types for its member variables. Listing 2-1 shows a sample class definition for a collection of pictures. The class in this instance contains an array of images and a list of the indexes into that array representing the selected items. The class also contains a string for the collection's title and a scalar Boolean variable indicating whether the collection is currently editable.

Listing 2-1 Definition of a custom data object

```
@interface PictureCollection : NSObject {
   NSMutableOrderedSet* pictures;
   NSMutableIndexSet* selection;

   NSString* title;
   BOOL editable;
}

@property (nonatomic, strong) NSString * title;
@property (nonatomic, readonly) NSOrderedSet* pictures;

// Method definitions...

@end
```

> **Note:** When defining data objects, it is strongly recommended that you declare properties for any member variables that you to expose to clients of the object. Synthesizing these properties in your implementation file automatically creates appropriate accessor methods with the attributes you require. This ensures that object relationships are maintained appropriately and that references to objects are removed at appropriate times.

Consider how undo operations on your custom objects might be handled. Supporting undo means being able to reverse changes made to your objects cleanly. If your objects incorporate complex business logic, you need to factor that logic in a way that can be undone easily. Here are some tips for implementing undo support in your custom objects:

- Define the methods you need to make sure that changes to your object are symmetrical. For example, if you define a method to add an item, make sure you have a method for removing an item in a similar way.
- Factor out your business logic from the code you use to change the values of member variables.
- For multistep actions, use the current `NSUndoManager` object to group the steps together.

For more information about how to implement undo support in your app, see *Undo Architecture*. For more information about the classes of the Foundation framework, see *Foundation Framework Reference*.

Defining a Structured Data Model Using Core Data

Core Data is a schema-driven object graph management and persistence framework. Fundamentally, Core Data helps you to save model objects (in the sense of the model-view-controller design pattern) to a file and get them back again. This is similar to archiving (see *Archives and Serializations Programming Guide*), but Core Data offers much more than that.

- Core Data provides an infrastructure for managing all the changes to your model objects. This gives you automatic support for undo and redo, and for maintaining reciprocal relationships between objects.
- It allows you to keep just a subset of your model objects in memory at any given time, which is very important for iOS apps.
- It uses a schema to describe the model objects. You define the principal features of your model classes—including the relationships between them—in a GUI-based editor. This provides a wealth of basic functionality "for free," including setting of default values and attribute value validation.
- It allows you to maintain disjoint sets of edits of your objects. This is useful if you want to, for example, allow the user to make edits in one view that may be discarded without affecting data displayed in another view.
- It has an infrastructure for data store versioning and migration. This lets you easily upgrade an old version of the user's file to the current version.
- It allows you to store your data in iCloud and access it from multiple devices.

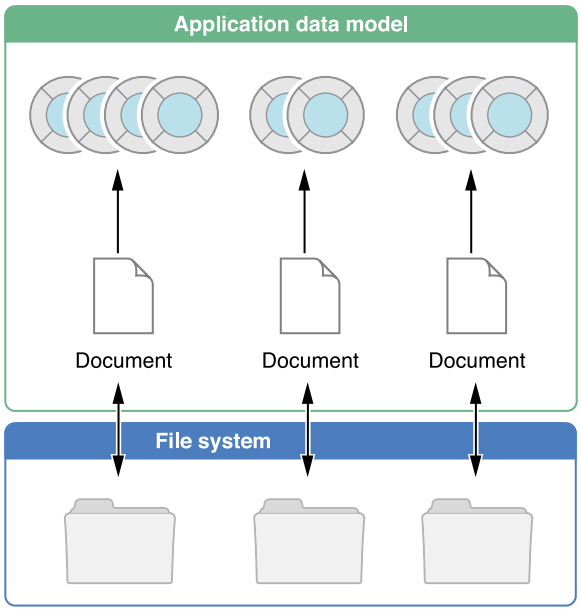
For information about how to use Core Data, see *Core Data Programming Guide*.

Defining a Document-Based Data Model

A document-based data model is a convenient way to manage the files your app writes to disk. In this type of data model, you use a document object to represent the contents of a single file (or file package) on disk. That document object is responsible for reading and writing the contents of the file and working with your app's view controllers to present the document's contents on screen. The traditional use for document objects is to manage files containing user data. For example, an app that creates and manages text files would use a separate document object to manage each text file. However, you can use document objects for private app data that is also backed by a file.

Figure 2-2 illustrates the typical relationships between documents, files, and the objects in your app's data model. With few exceptions, each document is self-contained and does not interact directly with other documents. The document manages a single file (or file package) and creates the in-memory representation of any data found in that file. Because the contents of each file are unique, the data structures associated with each document are also unique.

Figure 2-2 Using documents to manage the content of files



You use the `UIDocument` class to implement document objects in your iOS app. This class provides the basic infrastructure needed to handle the file management aspects of the document. Other benefits of `UIDocument` include:

- It provides support for autosaving the document contents at appropriate times.
- It handles the required file coordination for documents stored in iCloud. It also provides hooks for resolving version conflicts.
- It provides support for undoing actions.

You must subclass `UIDocument` in order to implement the specific behavior required by your app's documents. For detailed information about how to implement a document-based app using `UIDocument`, see *Document-Based App Programming Guide for iOS*.

Integrating iCloud Support Into Your App

No matter how you store your app's data, iCloud is a convenient way to make that data available to all of the user's devices. Supporting iCloud in your app just means changing where you store your files. Instead of storing them in your app's sandbox directory, you store them in a designated portion of the user's iCloud storage. In both cases, your app just works with files and directories. However, with iCloud, you have to do a little extra work because the data is now shared and accessible to multiple processes. Fortunately, when you use iOS frameworks to manage your data, much of the hard work needed to support iCloud is done for you.

- Document based apps get iCloud support through the `UIDocument` class. This class handles almost all of the complex interactions required to manage iCloud-based files.
- Core Data apps also get iCloud support through the Core Data framework. This framework automatically updates the data stores on all of the user's devices to account for new and changed data objects, leaving each device with a complete and up-to-date set of data.
- If you implement a custom data model and manage files yourself, you can use file presenters and file coordinators to ensure that the changes you make are done safely and in concert with the changes made on the user's other devices.
- For apps that want to share preferences or small quantities of infrequently changing data, you can use the `NSUbiquitousKeyValueStore` object to do so. This object supports the sharing of simple data types such as strings, numbers, and dates in limited quantities.

For more information about incorporating iCloud support into your apps, see *iCloud Design Guide*.

The User Interface

Every iOS app has at least one window and one view for presenting its content. The window provides the area in which to display the content and is an instance of the `UIWindow` class. Views are responsible for managing the drawing of your content (and handling touch events) and are instances of the `UIView` class. For interfaces that you build using view objects, your app's window naturally contains multiple view objects. For interfaces built using OpenGL ES, you typically have a single view and use that view to render your content.

View controllers also play a very important role in your app's user interface. A view controller is an instance of the `UIViewController` class and is responsible for managing a single set of views and the interactions between those views and other parts of your app. Because iOS apps have a limited amount of space in which to display content, view controllers also provide the infrastructure needed to swap out the views from one view controller and replace them with the views of another view controller. Thus, view controllers are you how implement transitions from one type of content to another.

You should always think of a view controller object as a self-contained unit. It handles the creation and destruction of its own views, handles their presentation on the screen, and coordinates interactions between the views and other objects in your app.

Building an Interface Using UIKit Views

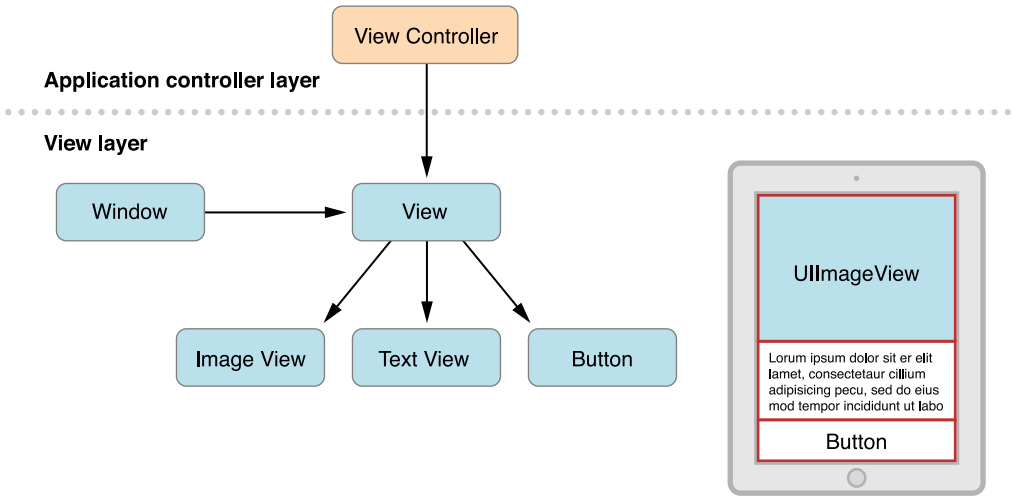
Apps that use UIKit views for drawing are easy to create because you can assemble a basic interface quickly. The UIKit framework provides many different types of views to help present and organize data. Controls—a special type of view—provide a built-in mechanism for executing custom code whenever the user performs appropriate actions. For example, clicking on a button causes the button's associated action method to be called.

The advantage of interfaces based on UIKit views is that you can assemble them graphically using Interface Builder—the visual interface editor built in to Xcode. Interface Builder provides a library of the standard views, controls, and other objects that you need to build your interface. After dragging these objects from the library, you drop them onto the work surface and arrange them in any way you want. You then use inspectors to configure those objects before saving them in a storyboard or nib file. The process of assembling your interface graphically is much faster than writing the equivalent code and allows you to see the results immediately, without the need to build and run your app.

> **Note:** You can also incorporate custom views into your UIKit view hierarchies. A custom view is a subclass of `UIView` in which you handle all of the drawing and event-handling tasks yourself. For more information about creating custom views and incorporating them into your view hierarchies, see *View Programming Guide for iOS*.

Figure 2-3 shows the basic structure of an app whose interface is constructed solely using view objects. In this instance, the main view spans the visible area of the window (minus the scroll bar) and provides a simple white background. The main view also contains three subviews: an image view, a text view, and a button. Those subviews are what the app uses to present content to the user and respond to interactions. All of the views in the hierarchy are managed by a single view controller object.

Figure 2-3 Building your interface using view objects



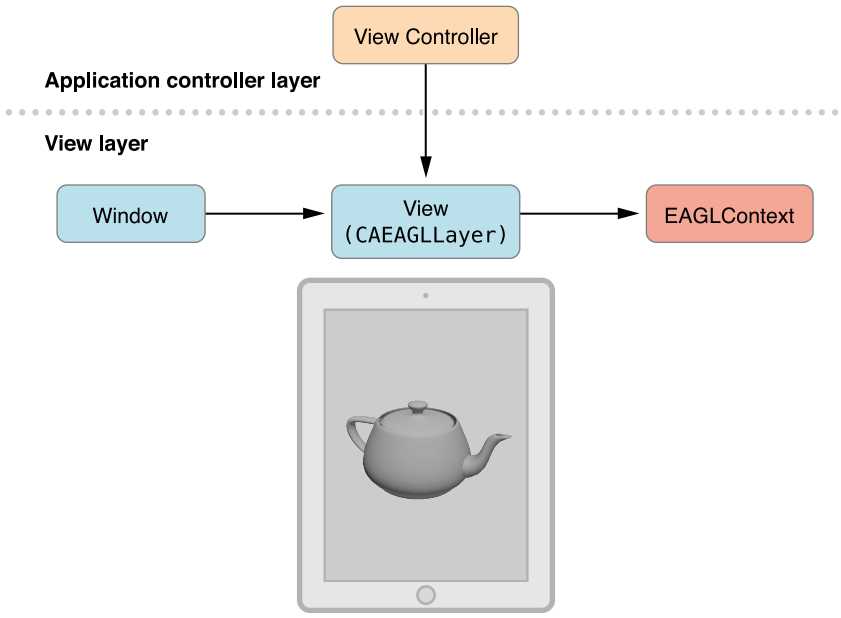
In a typical view-based app, you coordinate the onscreen views using your view controller objects. An app always has one view controller that is responsible for presenting all of the content on the screen. That view controller has a content view, which itself may contain other views. Some view controllers can also act as containers for content provided by other view controllers. For example, a split view controller displays the content from two view controllers side by side. Because view controllers play a vital role in view management, understand how they work and the benefits they provide by reading *View Controller Programming Guide for iOS*. For more information about views and the role they play in apps, see *View Programming Guide for iOS*.

Building an Interface Using Views and OpenGL ES

Games and other apps that need high frame rates or sophisticated drawing capabilities can add views specifically designed for OpenGL ES drawing to their view hierarchies. The simplest type of OpenGL ES app is one that has a window object and a single view for OpenGL ES drawing and a view controller to manage the presentation and rotation of that content. More sophisticated applications can use a mixture of both OpenGL ES views and UIKit views to implement their interfaces.

Figure 2-4 shows the configuration of an app that uses a single OpenGL ES view to draw its interface. Unlike a UIKit view, the OpenGL ES view is backed by a different type of layer object (a `CAEAGLLayer` object) instead of the standard layer used for view-based apps. The `CAEAGLLayer` object provides the drawing surface that OpenGL ES can render into. To manage the drawing environment, the app also creates an `EAGLContext` object and stores that object with the view to make it easy to retrieve.

Figure 2-4 Building your interface using OpenGL ES



For information on how to configure OpenGL ES for use in your app, see *OpenGL ES Programming Guide for iOS*.

The App Bundle

When you build your iOS app, Xcode packages it as a bundle. A **bundle** is a directory in the file system that groups related resources together in one place. An iOS app bundle contains the app executable file and supporting resource files such as app icons, image files, and localized content. Table 2-3 lists the contents of a typical iOS app bundle, which for demonstration purposes is called `MyApp`. This example is for illustrative purposes only. Some of the files listed in this table may not appear in your own app bundles.

Table 2-3 A typical app bundle

File	Example	Description
App executable	`MyApp`	The executable file contains your app's compiled code. The name of your app's executable file is the same as your app name minus the `.app` extension. This file is required.
The information property list file	`Info.plist`	The `Info.plist` file contains configuration data for the app. The system uses this data to determine how to interact with the app. This file is required and must be called `Info.plist`. For more information, see Figure 6-1 (page 120).

App icons	`Icon.png`
	`Icon@2x.png`
	`Icon-Small.png`
	`Icon-Small@2x.png`
		Your app icon is used to represent your app on the device's Home screen. Other icons are used by the system in appropriate places. Icons with `@2x` in their filename are intended for devices with Retina displays.
		An app icon is required. For information about specifying icon image files, see "App Icons" (page 98).

Launch images	`Default.png`
	`Default-Portrait.png`
	`Default-Landscape.png`
		The system uses this file as a temporary background while your app is launching. It is removed as soon as your app is ready to display its user interface.
		At least one launch image is required. For information about specifying launch images, see "App Launch (Default) Images" (page 100).

File	Example	Description
Storyboard files (or nib files)	`MainBoard.storyboard`	Storyboards contain the views and view controllers that the app presents on screen. Views in a storyboard are organized according to the view controller that presents them. Storyboards also identify the transitions (called segues) that take the user from one set of views to another. The name of the main storyboard file is set by Xcode when you create your project. You can change the name by assigning a different value to the `NSMainStoryboardFile` key in the `Info.plist` file.) Apps that use nib files instead of storyboards can replace the `NSMainStoryboardFile` key with the `NSMainNibFile` key and use that key to specify their main nib file. The use of storyboards (or nib files) is optional but recommended.
Ad hoc distribution icon	`iTunesArtwork`	If you are distributing your app ad hoc, include a 512 x 512 pixel version of your app icon. This icon is normally provided by the App Store from the materials you submit to iTunes Connect. However, because apps distributed ad hoc do not go through the App Store, your icon must be present in your app bundle instead. iTunes uses this icon to represent your app. (The file you specify should be the same one you would have submitted to the App Store, if you were distributing your app that way.) The filename of this icon must be `iTunesArtwork` and must not include a filename extension. This file is required for ad hoc distribution but is optional otherwise.

File	Example	Description
Settings bundle	`Settings.bundle`	If you want to expose custom app preferences through the Settings app, you must include a settings bundle. This bundle contains the property list data and other resource files that define your app preferences. The Settings app uses the information in this bundle to assemble the interface elements required by your app. This bundle is optional. For more information about preferences and specifying a settings bundle, see *Preferences and Settings Programming Guide*.
Nonlocalized resource files	`sun.png` `mydata.plist`	Nonlocalized resources include things like images, sound files, movies, and custom data files that your app uses. All of these files should be placed at the top level of your app bundle.

Subdirectories for localized resources	`en.lproj`
	`fr.lproj`
	`es.lproj`
		Localized resources must be placed in language-specific project directories, the names for which consist of an ISO 639-1 language abbreviation plus the `.lproj` suffix. (For example, the `en.lproj`, `fr.lproj`, and `es.lproj` directories contain resources localized for English, French, and Spanish.)
		An iOS app should be internationalized and have a *language* `.lproj` directory for each language it supports. In addition to providing localized versions of your app's custom resources, you can also localize your app icon, launch images, and Settings icon by placing files with the same name in your language-specific project directories.
		For more information, see "Localized Resource Files" (page 105).

For more information about the structure of an iOS app bundle, see *Bundle Programming Guide*. For information about how to load resource files from your bundle, see "Loading Resources Into Your App" (page 106).

App States and Multitasking

For iOS apps, it is crucial to know whether your app is running in the foreground or the background. Because system resources are more limited on iOS devices, an app must behave differently in the background than in the foreground. The operating system also limits what your app can do in the background in order to improve battery life and to improve the user's experience with the foreground app. The operating system notifies your app whenever it moves between the foreground and background. These notifications are your chance to modify your app's behavior.

While your app is in the foreground, the system sends touch events to it for processing. The UIKit infrastructure does most of the hard work of delivering events to your custom objects. All you have to do is override methods in the appropriate objects to process those events. For controls, UIKit simplifies things even further by handling the touch events for you and calling your custom code only when something interesting happens, such as when the value of a text field changes.

As you implement your app, follow these guidelines:

- (**Required**) Respond appropriately to the state transitions that occur. Not handling these transitions properly can lead to data loss and a bad user experience. For a summary of how to respond to state transitions, see "Managing App State Changes" (page 34).
- (**Required**) When moving to the background, make sure your app adjusts its behavior appropriately. For guidelines about what to do when your app moves to the background, see "Being a Responsible Background App" (page 63).
- (Recommended) Register for any notifications that report system changes your app needs. When an app is suspended, the system queues key notifications and delivers them when the app resumes execution. Apps should use these notifications to make a smooth transition back to execution. For more information, see "Processing Queued Notifications at Wakeup Time" (page 48).
- (Optional) If your app needs to do actual work while in the background, ask the system for the appropriate permissions to continue running. For more information about the types of background work you can do and how to request permission to do that work, see "Background Execution and Multitasking" (page 54).

Managing App State Changes

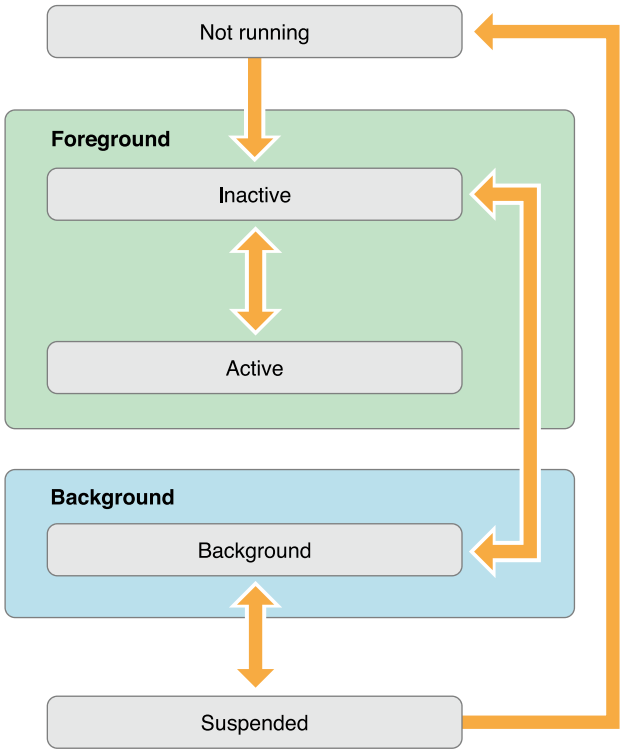
At any given moment, your app is in one of the states listed in Table 3-1. The system moves your app from state to state in response to actions happening throughout the system. For example, when the user presses the Home button, a phone call comes in, or any of several other interruptions occurs, the currently running apps change state in response. Figure 3-1 (page 35) shows the paths that an app takes when moving from state to state.

Table 3-1 App states

State	Description
Not running	The app has not been launched or was running but was terminated by the system.
Inactive	The app is running in the foreground but is currently not receiving events. (It may be executing other code though.) An app usually stays in this state only briefly as it transitions to a different state.
Active	The app is running in the foreground and is receiving events. This is the normal mode for foreground apps.
Background	The app is in the background and executing code. Most apps enter this state briefly on their way to being suspended. However, an app that requests extra execution time may remain in this state for a period of time. In addition, an app being launched directly into the background enters this state instead of the inactive state. For information about how to execute code while in the background, see "Background Execution and Multitasking" (page 54).

State	Description
Suspended	The app is in the background but is not executing code. The system moves apps to this state automatically and does not notify them before doing so. While suspended, an app remains in memory but does not execute any code. When a low-memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app.

Figure 3-1 State changes in an iOS app



Note: Apps running in iOS 3.2 and earlier do not enter the background or suspended states. In addition, some devices do not support multitasking or background execution at all, even when running iOS 4 or later. Apps running on those devices also do not enter the background or suspended states. Instead, apps are terminated upon leaving the foreground.

Most state transitions are accompanied by a corresponding call to the methods of your app delegate object. These methods are your chance to respond to state changes in an appropriate way. These methods are listed below, along with a summary of how you might use them.

- `application:willFinishLaunchingWithOptions:`—This method is your app's first chance to execute code at launch time.

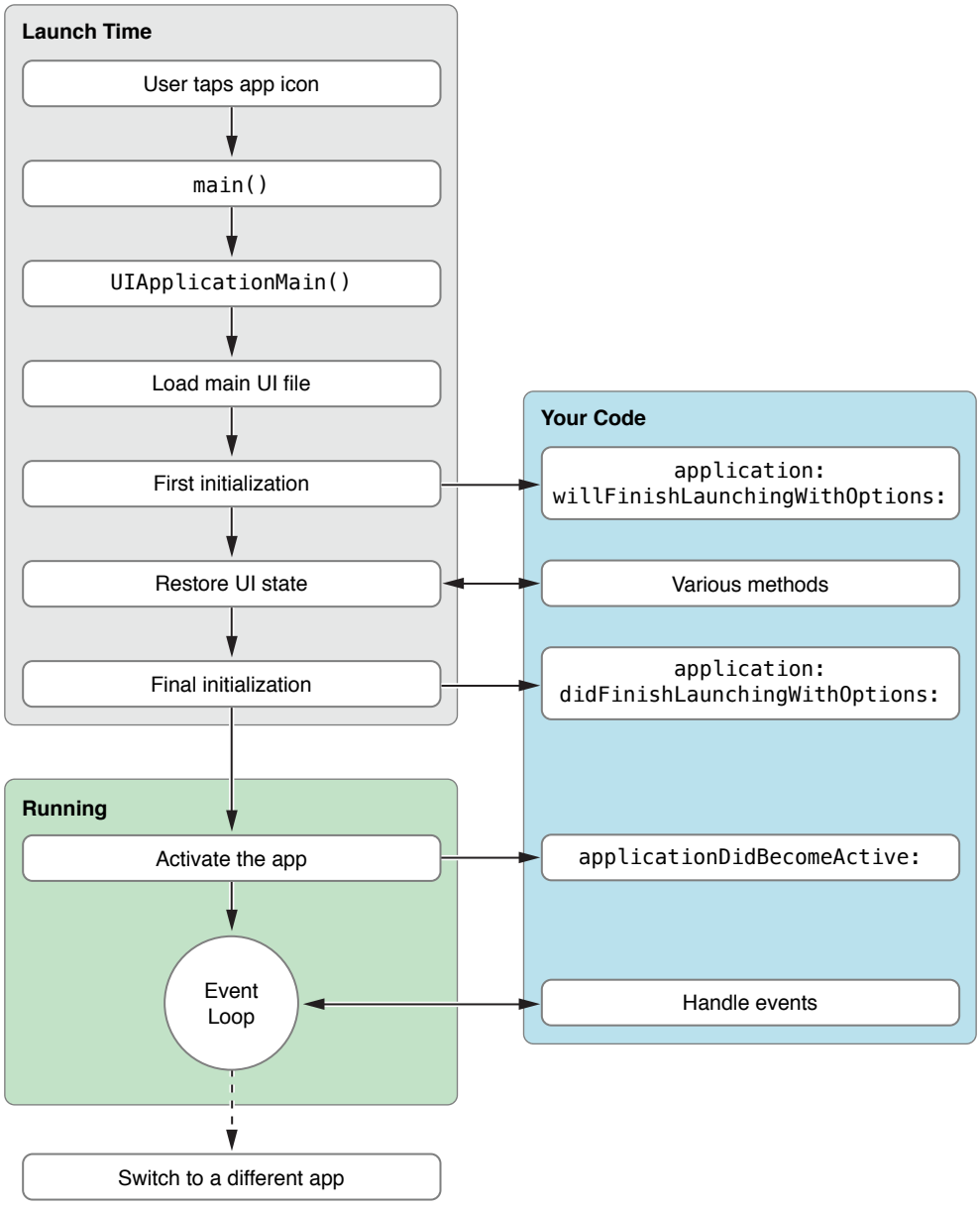
- `application:didFinishLaunchingWithOptions:`—This method allows you to perform any final initialization before your app is displayed to the user.
- `applicationDidBecomeActive:`—Lets your app know that it is about to become the foreground app. Use this method for any last minute preparation.
- `applicationWillResignActive:`—Lets you know that your app is transitioning away from being the foreground app. Use this method to put your app into a quiescent state.
- `applicationDidEnterBackground:`—Lets you know that your app is now running in the background and may be suspended at any time.
- `applicationWillEnterForeground:`—Lets you know that your app is moving out of the background and back into the foreground, but that it is not yet active.
- `applicationWillTerminate:`—Lets you know that your app is being terminated. This method is not called if your app is suspended.

The App Launch Cycle

When your app is launched, it moves from the not running state to the active or background state, transitioning briefly through the inactive state. As part of the launch cycle, the system creates a process and main thread for your app and calls your app's `main` function on that main thread. The default `main` function that comes with your Xcode project promptly hands control over to the UIKit framework, which does most of the work in initializing your app and preparing it to run.

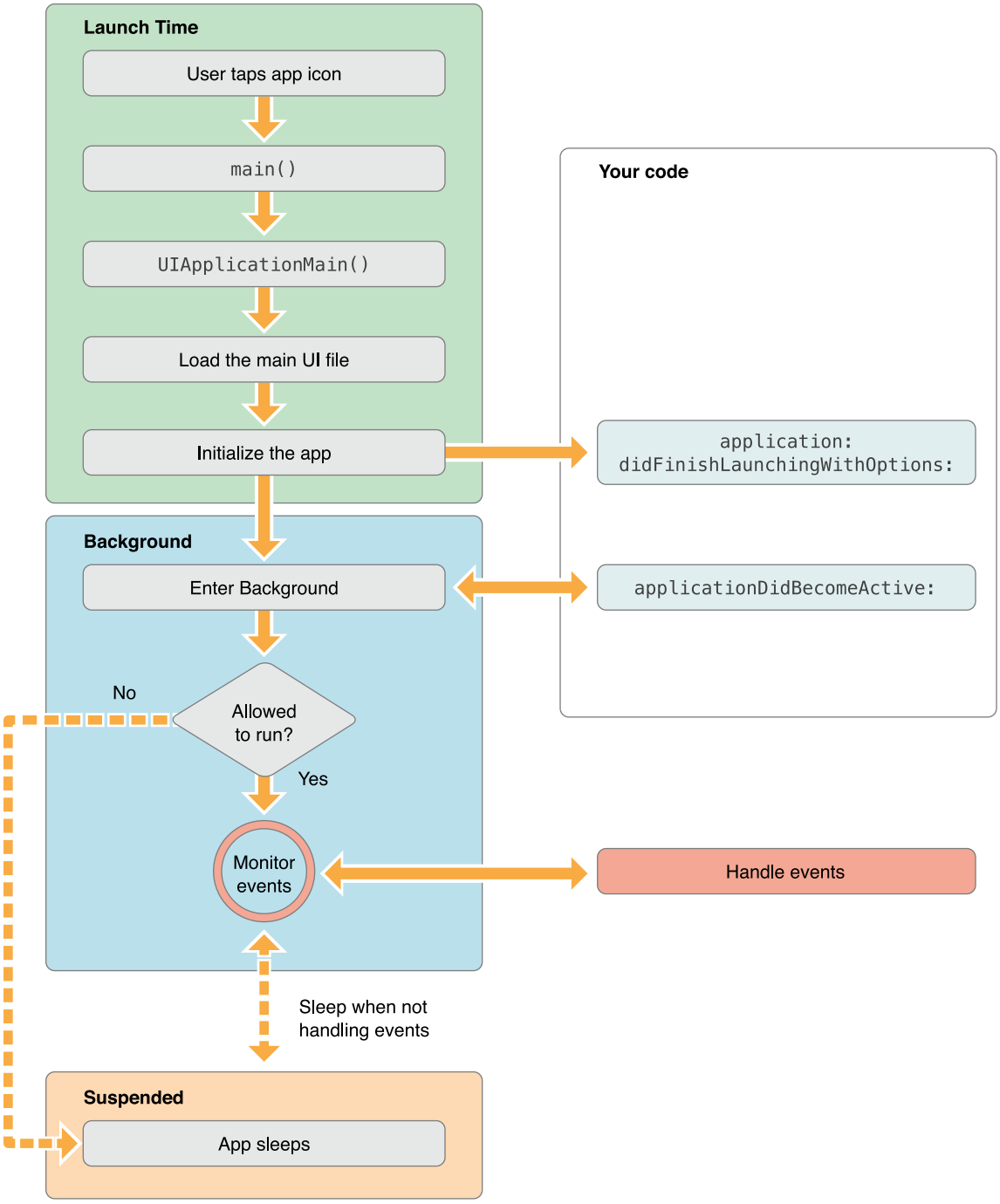
Figure 3-2 shows the sequence of events that occurs when an app is launched into the foreground, including the app delegate methods that are called.

Figure 3-2 Launching an app into the foreground



If your app is launched into the background instead—usually to handle some type of background event—the launch cycle changes slightly to the one shown in Figure 3-3. The main difference is that instead of your app being made active, it enters the background state to handle the event and then is suspended shortly afterward. When launching into the background, the system still loads your app's user interface files but it does not display the app's window.

Figure 3-3 Launching an app into the background



To determine whether your app is launching into the foreground or background, check the `applicationState` property of the shared `UIApplication` object in your `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` delegate method. When the app is launched into the foreground, this property contains the value `UIApplicationStateInactive`. When the app is launched into the background, the property contains the value `UIApplicationStateBackground` instead. You can use this difference to adjust the launch-time behavior of your delegate methods accordingly.

> **Note:** When an app is launched so that it can open a URL, the sequence of startup events is slightly different from those shown in Figure 3-2 and Figure 3-3. For information about the startup sequences that occur when opening a URL, see "Handling URL Requests" (page 120).

About the main Function

Like any C-based app, the main entry point for an iOS app at launch time is the `main` function. In an iOS app, the `main` function is used only minimally. Its main job is to hand control to the UIKit framework. Therefore, any new project you create in Xcode comes with a default `main` function like the one shown in Listing 3-1. With few exceptions, you should never change the implementation of this function.

Listing 3-1 The `main` function of an iOS app

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


int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([MyAppDelegate
 class]));
    }
}
```

> **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 *Advanced Memory Management Programming Guide*.

The `UIApplicationMain` function takes four parameters and uses them to initialize the app. You should never have to change the default values passed into this function. Still, it is valuable to understand their purpose and how they start the app.

- The `argc` and `argv` parameters contain any launch-time arguments passed to the app from the system. These arguments are parsed by the UIKit infrastructure and can otherwise be ignored.
- The third parameter identifies the name of the principal app class. This is the class responsible for running the app. It is recommend that you specify `nil` for this parameter, which causes UIKit to use the `UIApplication` class.
- The fourth parameter identifies the class of your custom app delegate. Your app delegate is responsible for managing the high-level interactions between the system and your code. The Xcode template projects set this parameter to an appropriate value automatically.

Another thing the `UIApplicationMain` function does is load the app's main user interface file. The main interface file contains the initial view-related objects you plan to display in your app's user interface. For apps that use "Using Storyboards", this function loads the initial view controller from your storyboard and installs it in the window provided by your app delegate. For apps that use nib files, the function loads the nib file contents into memory but does not install them in your app's window; you must install them in the `application:willFinishLaunchingWithOptions:` method of your app delegate.

An app can have either a main storyboard file or a main nib file but it cannot have both. Storyboards are the preferred way to specify your app's user interface but are not supported on all versions of iOS. The name of your app's main storyboard file goes in the `UIMainStoryboardFile` key of your app's `Info.plist` file. (For nib-based apps, the name of your main nib file goes in the `NSMainNibFile` key instead.) Normally, Xcode sets the value of the appropriate key when you create your project, but you can change it later if needed.

For more information about the `Info.plist` file and how you use it to configure your app, see "The Information Property List File" (page 93).

What to Do at Launch Time

When your app is launched (either into the foreground or background), use your app delegate's `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods to do the following:

- Check the contents of the launch options dictionary for information about why the app was launched, and respond appropriately.
- Initialize the app's critical data structures.
- Prepare your app's window and views for display.

  Apps that use OpenGL ES should not use this method to prepare their drawing environment. Instead, they should defer any OpenGL ES drawing calls to the `applicationDidBecomeActive:` method.

If your app does not automatically load a main storyboard or nib file at launch time, you can use the `application:willFinishLaunchingWithOptions:` method to prepare your app's window for display. For apps that support both portrait and landscape orientations, always set up the root view controller of your main window in a portrait orientation. If the device is in a different orientation at launch time, the system tells the root view controller to rotate your views to the correct orientation before displaying the window.

Your `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods should always be as lightweight as possible to reduce your app's launch time. Apps are expected to launch and initialize themselves and start handling events in less than 5 seconds. If an app does not finish its launch cycle in a timely manner, the system kills it for being unresponsive. Thus, any tasks that might slow down your launch (such as accessing the network) should be executed asynchronously on a secondary thread.

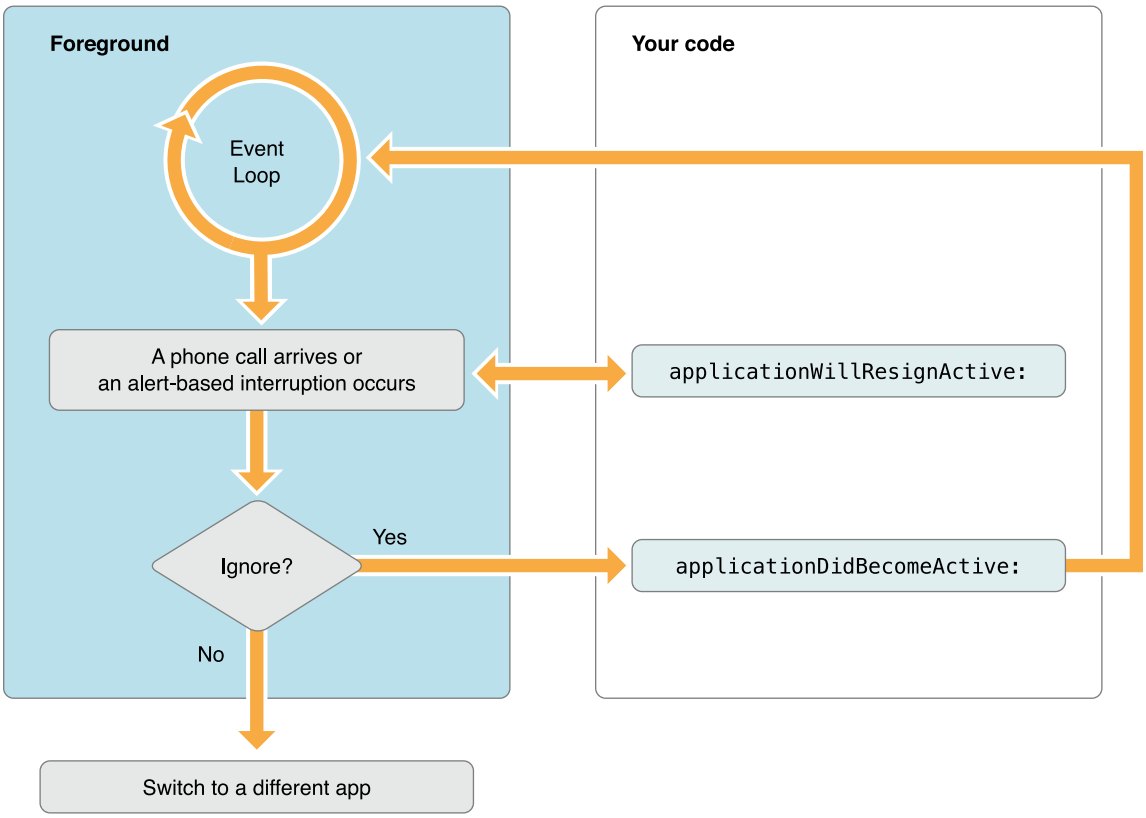
When launching into the foreground, the system also calls the `applicationDidBecomeActive:` method to finish the transition to the foreground. Because this method is called both at launch time and when transitioning from the background, use it to perform any tasks that are common to the two transitions.

When launching into the background, there should not be much for your app to do except get ready to handle whatever event arrived.

Responding to Interruptions

When an alert-based interruption occurs, such as an incoming phone call, the app moves temporarily to the inactive state so that the system can prompt the user about how to proceed. The app remains in this state until the user dismiss the alert. At this point, the app either returns to the active state or moves to the background state. Figure 3-4 shows the flow of events through your app when an alert-based interruption occurs.

Figure 3-4 Handling alert-based interruptions



In iOS 5, notifications that display a banner do not deactivate your app in the way that alert-based notifications do. Instead, the banner is laid along the top edge of your app window and your app continues receive touch events as before. However, if the user pulls down the banner to reveal the notification center, your app moves to the inactive state just as if an alert-based interruption had occurred. Your app remains in the inactive state until the user dismisses the notification center or launches another app. At this point, your app moves to the appropriate active or background state. The user can use the Settings app to configure which notifications display a banner and which display an alert.

Pressing the Sleep/Wake button is another type of interruption that causes your app to be deactivated temporarily. When the user presses this button, the system disables touch events, moves the app to the background but sets the value of the app's `applicationState` property to `UIApplicationStateInactive`

(as opposed to `UIApplicationStateBackground`), and finally locks the screen. A locked screen has additional consequences for apps that use data protection to encrypt files. Those consequences are described in "What to Do When an Interruption Occurs" (page 43).

What to Do When an Interruption Occurs

Alert-based interruptions result in a temporary loss of control by your app. Your app continues to run in the foreground, but it does not receive touch events from the system. (It does continue to receive notifications and other types of events, such as accelerometer events, though.) In response to this change, your app should do the following in its `applicationWillResignActive:` method:

- Stop timers and other periodic tasks.
- Stop any running metadata queries.
- Do not initiate any new tasks.
- Pause movie playback (except when playing back over AirPlay).
- Enter into a pause state if your app is a game.
- Throttle back OpenGL ES frame rates.
- Suspend any dispatch queues or operation queues executing non-critical code. (You can continue processing network requests and other time-sensitive background tasks while inactive.)

When your app is moved back to the active state, its `applicationDidBecomeActive:` method should reverse any of the steps taken in the `applicationWillResignActive:` method. Thus, upon reactivation, your app should restart timers, resume dispatch queues, and throttle up OpenGL ES frame rates again. However, games should not resume automatically; they should remain paused until the user chooses to resume them.

When the user presses the Sleep/Wake button, apps with files protected by the `NSFileProtectionComplete` protection option must close any references to those files. For devices configured with an appropriate password, pressing the Sleep/Wake button locks the screen and forces the system to throw away the decryption keys for files with complete protection enabled. While the screen is locked, any attempts to access the corresponding files will fail. So if you have such files, you should close any references to them in your `applicationWillResignActive:` method and open new references in your `applicationDidBecomeActive:` method.

Adjusting Your User Interface During a Phone Call

When the user takes a call and then returns to your app 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.

The best way to handle status bar height changes is to use view controllers to manage your views. When installed in your interface, view controllers automatically adjust the height of their managed views when the status bar frame size changes.

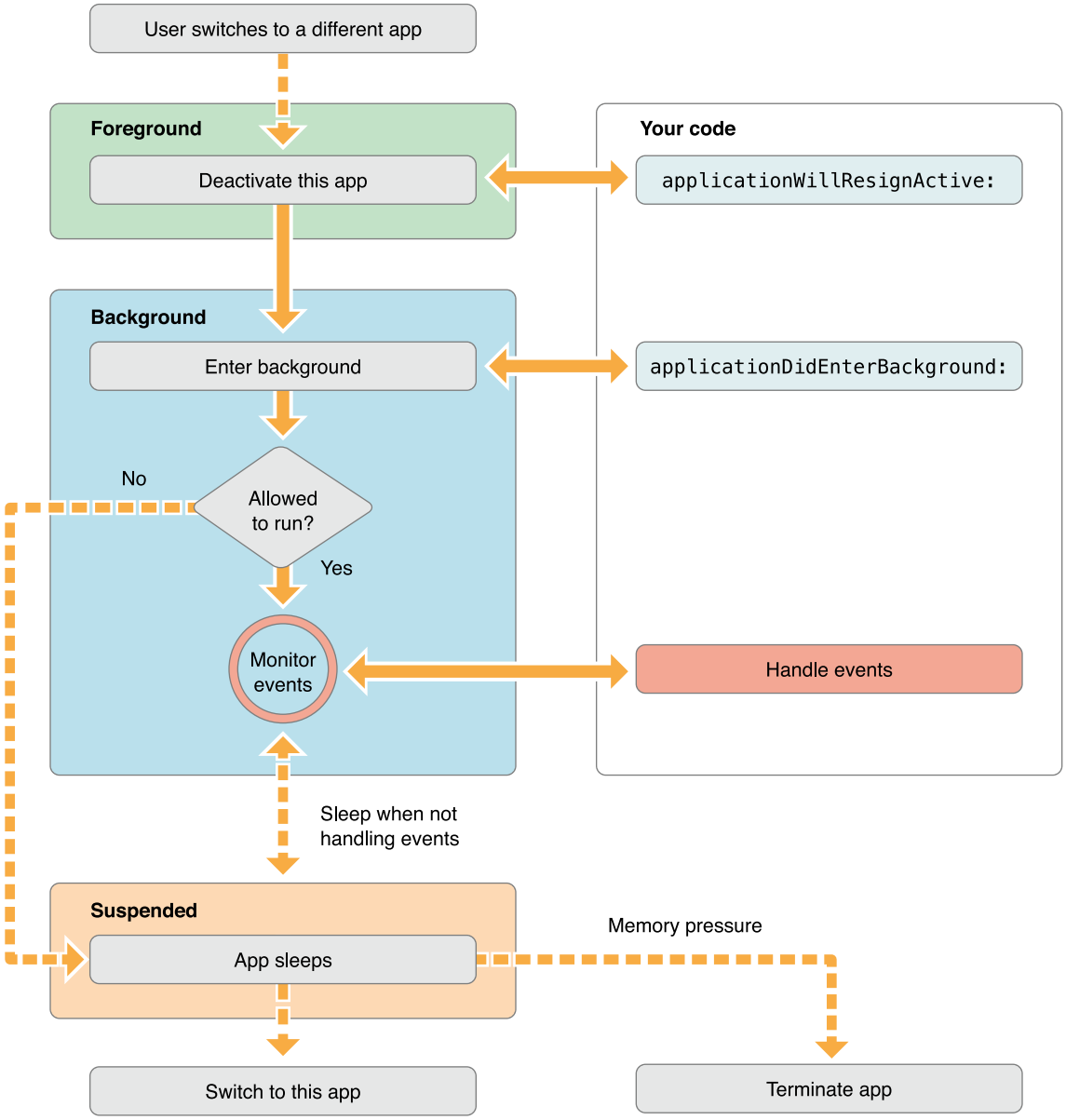
If your app does not use view controllers for some reason, you must respond to status bar frame changes manually by registering for the `UIApplicationDidChangeStatusBarFrameNotification` notification. Your handler for this notification should get the status bar height and use it to adjust the height of your app's views appropriately.

Moving to the Background

When the user presses the Home button, presses the Sleep/Wake button, or the system launches another app, the foreground app transitions to the inactive state and then to the background state. These transitions result in calls to the app delegate's `applicationWillResignActive:` and `applicationDidEnterBackground:` methods, as shown in Figure 3-5. After returning from the `applicationDidEnterBackground:` method,

most apps move to the suspended state shortly afterward. Apps that request specific background tasks (such as playing music) or that request a little extra execution time from the system may continue to run for a while longer.

Figure 3-5 Moving from the foreground to the background



> **Note:** Apps are moved to the background only on devices that support multitasking and only if those devices are running iOS 4.0 or later. In all other cases, the app is terminated (and thus purged from memory) instead of moved to the background.

What to Do When Moving to the Background

Apps can use their `applicationDidEnterBackground:` method to prepare for moving to the background state. When moving to the background, all apps should do the following:

- **Prepare to have their picture taken.** When the `applicationDidEnterBackground:` method returns, the system takes a picture of your app's user interface and uses the resulting image for transition animations. If any views in your interface contain sensitive information, you should hide or modify those views before the `applicationDidEnterBackground:` method returns.
- **Save user data and app state information.** All unsaved changes should be written to disk when entering the background. This step is necessary because your app might be quietly killed while in the background for any number of reasons. You can perform this operation from a background thread as needed.
- **Free up as much memory as possible.** For more information about what to do and why this is important, see "Memory Usage for Background Apps" (page 47).

Your app delegate's `applicationDidEnterBackground:` method has approximately 5 seconds to finish any tasks and return. In practice, this method should return as quickly as possible. If the method does not return before time runs out, your app is killed and purged from memory. If you still need more time to perform tasks, call the `beginBackgroundTaskWithExpirationHandler:` method to request background execution time and then start any long-running tasks in a secondary thread. Regardless of whether you start any background tasks, the `applicationDidEnterBackground:` method must still exit within 5 seconds.

> **Note:** The `UIApplicationDidEnterBackgroundNotification` notification is also sent to let interested parts of your app know that it is entering the background. Objects in your app can use the default notification center to register for this notification.

Depending on the features of your app, there are other things your app should do when moving to the background. For example, any active Bonjour services should be suspended and the app should stop calling OpenGL ES functions. For a list of things your app should do when moving to the background, see "Being a Responsible Background App" (page 63).

Memory Usage for Background Apps

Every app should free up as much memory as is practical upon entering the background. The system tries to keep as many apps in memory at the same time as it can, but when memory runs low it terminates suspended apps to reclaim that memory. Apps that consume large amounts of memory while in the background are the first apps to be terminated.

Practically speaking, your app should remove strong references to objects as soon as they are no longer needed. Removing strong references gives the compiler the ability to release the objects right away so that the corresponding memory can be reclaimed. However, if you want to cache some objects to improve performance, you can wait until the app transitions to the background before removing references to them.

Some examples of objects that you should remove strong references to as soon as possible include:

- Image objects
- Large media or data files that you can load again from disk
- Any other objects that your app does not need and can recreate easily later

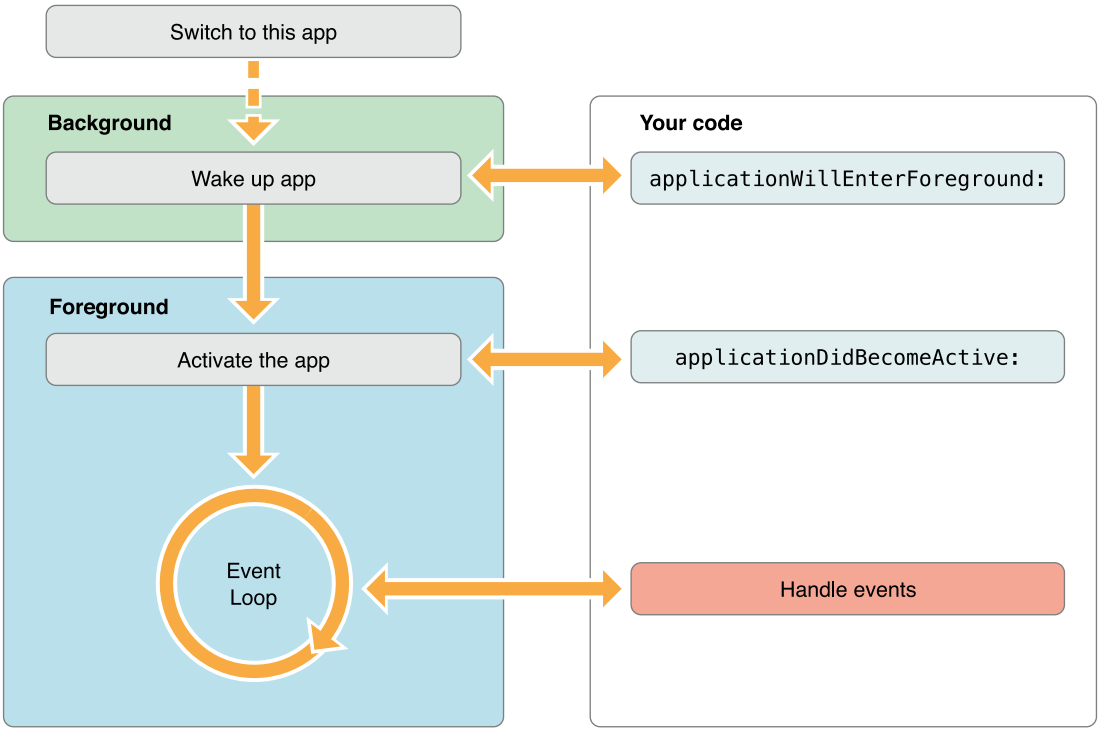
To help reduce your app's memory footprint, the system automatically purges some data allocated on behalf of your app when your app moves to the background.

- The system purges the backing store for all Core Animation layers. This effort does not remove your app's layer objects from memory, nor does it change the current layer properties. It simply prevents the contents of those layers from appearing onscreen, which given that the app is in the background should not happen anyway.
- It removes any system references to cached images. (If your app does not have a strong reference to the images, they are subsequently removed from memory.)
- It removes strong references to some other system-managed data caches.

Returning to the Foreground

Returning to the foreground is your app's chance to restart the tasks that it stopped when it moved to the background. The steps that occur when moving to the foreground are shown in Figure 3-6. The `applicationWillEnterForeground:` method should undo anything that was done in your `applicationDidEnterBackground:` method, and the `applicationDidBecomeActive:` method should continue to perform the same activation tasks that it would at launch time.

Figure 3-6 Transitioning from the background to the foreground



Note: The `UIApplicationWillEnterForegroundNotification` notification is also available for tracking when your app reenters the foreground. Objects in your app can use the default notification center to register for this notification.

Processing Queued Notifications at Wakeup Time

An app in the suspended state must be ready to handle any queued notifications when it returns to a foreground or background execution state. A suspended app does not execute any code and therefore cannot process notifications related to orientation changes, time changes, preferences changes, and many others that would affect the app's appearance or state. To make sure these changes are not lost, the system queues many relevant notifications and delivers them to the app as soon as it starts executing code again (either in the foreground

or background). To prevent your app from becoming overloaded with notifications when it resumes, the system coalesces events and delivers a single notification (of each relevant type) that reflects the net change since your app was suspended.

Table 3-2 lists the notifications that can be coalesced and delivered to your app. Most of these notifications are delivered directly to the registered observers. Some, like those related to device orientation changes, are typically intercepted by a system framework and delivered to your app in another way.

Table 3-2 Notifications delivered to waking apps

Event	Notifications
An accessory is connected or disconnected.	`EAAccessoryDidConnectNotification` `EAAccessoryDidDisconnectNotification`
The device orientation changes.	`UIDeviceOrientationDidChangeNotification` In addition to this notification, view controllers update their interface orientations automatically.
There is a significant time change.	`UIApplicationSignificantTimeChangeNotification`
The battery level or battery state changes.	`UIDeviceBatteryLevelDidChangeNotification` `UIDeviceBatteryStateDidChangeNotification`
The proximity state changes.	`UIDeviceProximityStateDidChangeNotification`
The status of protected files changes.	`UIApplicationProtectedDataWillBecomeUnavailable` `UIApplicationProtectedDataDidBecomeAvailable`
An external display is connected or disconnected.	`UIScreenDidConnectNotification` `UIScreenDidDisconnectNotification`
The screen mode of a display changes.	`UIScreenModeDidChangeNotification`
Preferences that your app exposes through the Settings app changed.	`NSUserDefaultsDidChangeNotification`
The current language or locale settings changed.	`NSCurrentLocaleDidChangeNotification`
The status of the user's iCloud account changed.	`NSUbiquityIdentityDidChangeNotification`

Queued notifications are delivered on your app's main run loop and are typically delivered before any touch events or other user input. Most apps should be able to handle these events quickly enough that they would not cause any noticeable lag when resumed. However, if your app appears sluggish when it returns from the background state, use Instruments to determine whether your notification handler code is causing the delay.

An app returning to the foreground also receives view-update notifications for any views that were marked dirty since the last update. An app running in the background can still call the `setNeedsDisplay` or `setNeedsDisplayInRect:` methods to request an update for its views. However, because the views are not visible, the system coalesces the requests and updates the views only after the app returns to the foreground.

Handling iCloud Changes

If the status of iCloud changes for any reason, the system delivers a `NSUbiquityIdentityDidChangeNotification` notification to your app. The state of iCloud changes when the user logs into or out of an iCloud account or enables or disables the syncing of documents and data. This notification is your app's cue to update caches and any iCloud-related user interface elements to accommodate the change. For example, when the user logs out of iCloud, you should remove references to all iCloud–based files or data.

If your app has already prompted the user about whether to store files in iCloud, do not prompt again when the status of iCloud changes. After prompting the user the first time, store the user's choice in your app's local preferences. You might then want to expose that preference using a Settings bundle or as an option in your app. But do not repeat the prompt again unless that preference is not currently in the user defaults database.

Handling Locale Changes Gracefully

If a user changes the current language while your app is suspended, you can use the `NSCurrentLocaleDidChangeNotification` notification to force updates to any views containing locale-sensitive information, such as dates, times, and numbers when your app returns to the foreground. Of course, the best way to avoid language-related issues is to write your code in ways that make it easy to update views. For example:

- Use the `autoupdatingCurrentLocale` class method when retrieving `NSLocale` objects. This method returns a locale object that updates itself automatically in response to changes, so you never need to recreate it. However, when the locale changes, you still need to refresh views that contain content derived from the current locale.
- Re-create any cached date and number formatter objects whenever the current locale information changes.

For more information about internationalizing your code to handle locale changes, see *Internationalization Programming Topics*.

Responding to Changes in Your App's Settings

If your app has settings that are managed by the Settings app, it should observe the `NSUserDefaultsDidChangeNotification` notification. Because the user can modify settings while your app is suspended or in the background, you can use this notification to respond to any important changes in those settings. In some cases, responding to this notification can help close a potential security hole. For example, an email program should respond to changes in the user's account information. Failure to monitor these changes could cause privacy or security issues. Specifically, the current user might be able to send email using the old account information, even if the account no longer belongs to that person.

Upon receiving the `NSUserDefaultsDidChangeNotification` notification, your app should reload any relevant settings and, if necessary, reset its user interface appropriately. In cases where passwords or other security-related information has changed, you should also hide any previously displayed information and force the user to enter the new password.

App Termination

Although apps are generally moved to the background and suspended, if any of the following conditions are true, your app is terminated and purged from memory instead:

- The app is linked against a version of iOS earlier than 4.0.
- The app is deployed on a device running a version of iOS earlier than 4.0.
- The current device does not support multitasking; see "Determining Whether Multitasking Is Available" (page 54).
- The app includes the `UIApplicationExitsOnSuspend` key in its `Info.plist` file; see "Opting out of Background Execution" (page 65).

If your app is running (either in the foreground or background) at termination time, the system calls your app delegate's `applicationWillTerminate:` method so that you can perform any required cleanup. You can use this method to save user data or app state information that you would use to restore your app to its current state on a subsequent launch. Your method has approximately 5 seconds to perform any tasks and return. If it does not return in time, the app is killed and removed from memory.

Important: The `applicationWillTerminate:` method is not called if your app is currently suspended.

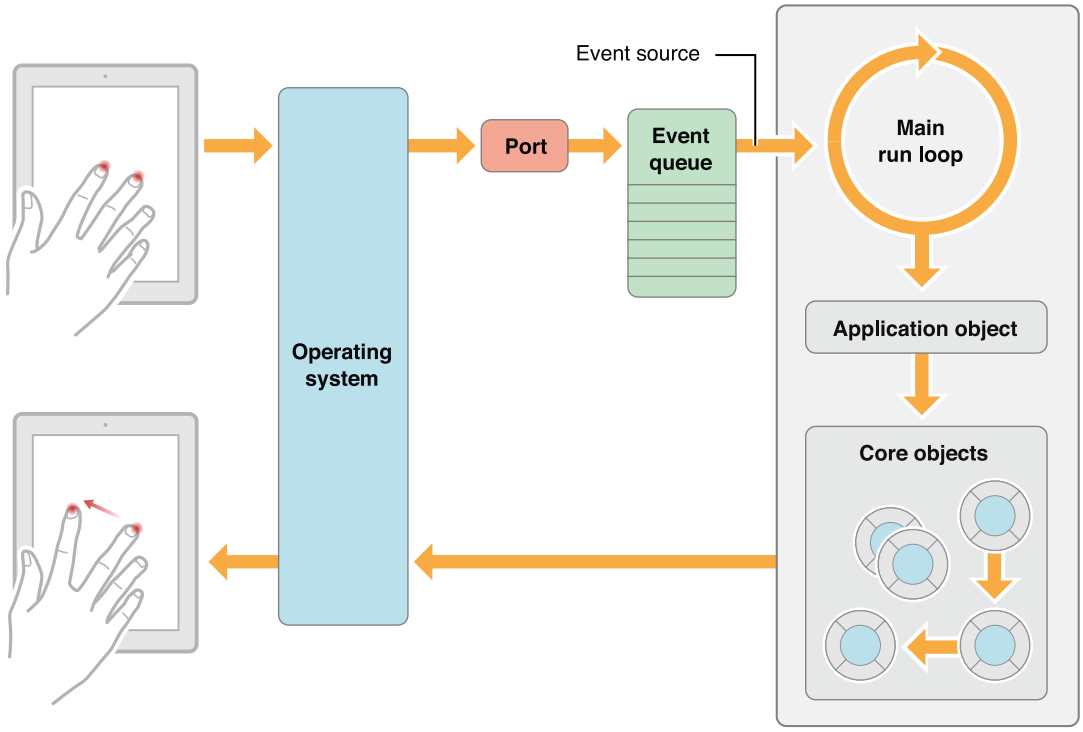
Even if you develop your app using iOS SDK 4 and later, you must still be prepared for your app to be killed without any notification. The user can kill apps explicitly using the multitasking UI. In addition, if memory becomes constrained, the system might remove apps from memory to make more room. Suspended apps are not notified of termination but if your app is currently running in the background state (and not suspended), the system calls the `applicationWillTerminate:` method of your app delegate. Your app cannot request additional background execution time from this method.

The Main Run Loop

The **main run loop** of your app is responsible for processing all user-related events. The `UIApplication` object sets up the main run loop at launch time and uses it to process events and handle updates to view-based interfaces. As the name suggests, the main run loop executes on the app's main thread. This behavior ensures that user-related events are processed serially in the order in which they were received.

Figure 3-7 shows the architecture of the main run loop and how user events result in actions taken by your app. As the user interacts with a device, events related to those interactions are generated by the system and delivered to the app via a special port set up by UIKit. Events are queued internally by the app and dispatched one-by-one to the main run loop for execution. The `UIApplication` object is the first object to receive the event and make the decision about what needs to be done. A touch event is usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred. Other events might take slightly different paths through various app objects.

Figure 3-7 Processing events in the main run loop



Many types of events can be delivered in an iOS app. The most common ones are listed in Table 3-3. Many of these event types are delivered using the main run loop of your app, but some are not. For example, accelerometer events are delivered directly to the accelerometer delegate object that you specify. For information about how to handle most types of events—including touch, remote control, motion, accelerometer, and gyroscopic events—see *Event Handling Guide for iOS*.

Table 3-3 Common types of events for iOS apps

Event type	Delivered to…	Notes
Touch	The view object in which the event occurred	Views are responder objects. Any touch events not handled by the view are forwarded down the responder chain for processing.
Remote control	First responder object	Remote control events are for controlling media playback and are generated by headphones and other accessories.
Motion	First responder object	Motion events reflect specific motion-related events (such as shaking a device) and are handled separately from other accelerometer-based events. .
Accelerometer Core Motion	The object you designate	Events related to the accelerometer and gyroscope hardware are delivered to the object you designate.
Redraw	The view that needs the update	Redraw events do not involve an event object but are simply calls to the view to draw itself. The drawing architecture for iOS is described in *Drawing and Printing Guide for iOS* .
Location	The object you designate	You register to receive location events using the Core Location framework. For more information about using Core Location, see *Location Awareness Programming Guide* .

Some events, such as touch and remote control events, are handled by your app's responder objects. Responder objects are everywhere in your app. (The `UIApplication` object, your view objects, and your view controller objects are all examples of responder objects.) Most events target a specific responder object but can be passed to other responder objects (via the responder chain) if needed to handle an event. For example, a view that does not handle an event can pass the event to its superview or to a view controller.

Touch events occurring in controls (such as buttons) are handled differently than touch events occurring in many other types of views. There are typically only a limited number of interactions possible with a control, and so those interactions are repackaged into action messages and delivered to an appropriate target object. This target-action design pattern makes it easy to use controls to trigger the execution of custom code in your app.

Background Execution and Multitasking

In iOS 4 and later, multitasking allows apps to continue running in the background even after the user switches to another app while still preserving battery life as much as possible. Most apps are moved to the suspended state shortly after entering the background. Only apps that provide important services to the user are allowed to continue running for any amount of time.

As much as possible, you are encouraged to avoid executing in the background and let your app be suspended. If you find you need to perform background tasks, here are some guidelines for when that is appropriate:

- You need to implement at least one of several specific user services.
- You need to perform a single finite-length task.
- You need to use notifications to alert the user to some relevant piece of information when your app is not running.

The system keeps suspended apps in memory for as long as possible, removing them only when the amount of free memory gets low. Remaining in memory means that subsequent launches of your app are much faster. At the same time, being suspended means your app does not drain the device's battery as fast.

Determining Whether Multitasking Is Available

Apps must be prepared to handle situations where multitasking (and therefore background execution) is not available. Even if your app is specifically built for iOS 4 and later, some devices running iOS 4 may not support multitasking. And multitasking is never available on devices running iOS 3 and earlier. If your app supports these earlier versions of iOS, it must be prepared to run without multitasking.

If the presence or absence of multitasking changes the way your app behaves, check the `multitaskingSupported` property of the `UIDevice` class to determine whether multitasking is available before performing the relevant task. For apps built for iOS 4 and later, this property is always available. However, if your app supports earlier versions of the system, you must check to see whether the property itself is available before accessing it, as shown in Listing 3-2.

Listing 3-2 Checking for background support in earlier versions of iOS

```
UIDevice* device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
   backgroundSupported = device.multitaskingSupported;
```

Executing a Finite-Length Task in the Background

Apps that are transitioning to the background can request an extra amount of time to finish any important last-minute tasks. To request background execution time, call the `beginBackgroundTaskWithExpirationHandler:` method of the `UIApplication` class. If your app moves to the background while the task is in progress, or if your app was already in the background, this method delays the suspension of your app. This can be important if your app is performing some important task, such as writing user data to disk or downloading an important file from a network server.

The way to use the `beginBackgroundTaskWithExpirationHandler:` method is to call it before starting the task you want to protect. Every call to this method must be balanced by a corresponding call to the `endBackgroundTask:` method to mark the end of the task. Because apps are given only a limited amount of time to finish background tasks, you must call this method before time expires; otherwise the system will terminate your app. To avoid termination, you can also provide an expiration handler when starting a task and call the `endBackgroundTask:` method from there. (You can use the value in the `backgroundTimeRemaining` property of the app object to see how much time is left.)

> **Important:** An app can have any number of tasks running at the same time. Each time you start a task, the `beginBackgroundTaskWithExpirationHandler:` method returns a unique identifier for the task. You must pass this same identifier to the `endBackgroundTask:` method when it comes time to end the task.

Listing 3-3 shows how to start a long-running task when your app transitions to the background. In this example, the request to start a background task includes an expiration handler just in case the task takes too long. The task itself is then submitted to a dispatch queue for asynchronous execution so that the `applicationDidEnterBackground:` method can return normally. The use of blocks simplifies the code needed to maintain references to any important variables, such as the background task identifier. The `bgTask` variable is a member variable of the class that stores a pointer to the current background task identifier and is initialized prior to its use in this method.

Listing 3-3 Starting a background task at quit time

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    bgTask = [application beginBackgroundTaskWithExpirationHandler:^{
        // Clean up any unfinished task business by marking where you.
        // stopped or ending the task outright.
        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    }];
```

```
    // Start the long-running task and return immediately.
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^{

        // Do the work associated with the task, preferably in chunks.

        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    });
}
```

Note: Always provide an expiration handler when starting a task, but if you want to know how much time your app has left to run, get the value of the `backgroundTimeRemaining` property of `UIApplication`.

In your own expiration handlers, you can include additional code needed to close out your task. However, any code you include must not take too long to execute because, by the time your expiration handler is called, your app is already very close to its time limit. For this reason, perform only minimal cleanup of your state information and end the task.

Scheduling the Delivery of Local Notifications

Notifications are a way for an app that is suspended, is in the background, or is not running to get the user's attention. Apps can use local notifications to display alerts, play sounds, badge the app's icon, or a combination of the three. For example, an alarm clock app might use local notifications to play an alarm sound and display an alert to disable the alarm. When a notification is delivered to the user, the user must decide if the information warrants bringing the app back to the foreground. (If the app is already running in the foreground, local notifications are delivered quietly to the app and not to the user.)

To schedule the delivery of a local notification, create an instance of the `UILocalNotification` class, configure the notification parameters, and schedule it using the methods of the `UIApplication` class. The local notification object contains information about the type of notification to deliver (sound, alert, or badge) and the time (when applicable) at which to deliver it. The methods of the `UIApplication` class provide options for delivering notifications immediately or at the scheduled time.

Listing 3-4 shows an example that schedules a single alarm using a date and time that is set by the user. This example configures only one alarm at a time and cancels the previous alarm before scheduling a new one. (Your own apps can have no more than 128 local notifications active at any given time, any of which can be configured to repeat at a specified interval.) The alarm itself consists of an alert box and a sound file that is played if the app is not running or is in the background when the alarm fires. If the app is active and therefore running in the foreground, the app delegate's `application:didReceiveLocalNotification:` method is called instead.

Listing 3-4 Scheduling an alarm notification

```
- (void)scheduleAlarmForDate:(NSDate*)theDate
{
    UIApplication* app = [UIApplication sharedApplication];
    NSArray*    oldNotifications = [app scheduledLocalNotifications];

    // Clear out the old notification before scheduling a new one.
    if ([oldNotifications count] > 0)
        [app cancelAllLocalNotifications];

    // Create a new notification.
    UILocalNotification* alarm = [[UILocalNotification alloc] init];
    if (alarm)
    {
        alarm.fireDate = theDate;
        alarm.timeZone = [NSTimeZone defaultTimeZone];
        alarm.repeatInterval = 0;
        alarm.soundName = @"alarmsound.caf";
        alarm.alertBody = @"Time to wake up!";

        [app scheduleLocalNotification:alarm];
    }
}
```

Sound files used with local notifications have the same requirements as those used for push notifications. Custom sound files must be located inside your app's main bundle and support one of the following formats: Linear PCM, MA4, µ-Law, or a-Law. You can also specify the sound name `default` to play the default alert sound for the device. When the notification is sent and the sound is played, the system also triggers a vibration on devices that support it.

You can cancel scheduled notifications or get a list of notifications using the methods of the `UIApplication` class. For more information about these methods, see *UIApplication Class Reference*. For additional information about configuring local notifications, see *Local and Push Notification Programming Guide*.

Implementing Long-Running Background Tasks

For tasks that require more execution time to implement, you must request specific permissions to run them in the background without their being suspended. In iOS, only specific app types are allowed to run in the background:

- Apps that play audible content to the user while in the background, such as a music player app
- Apps that keep users informed of their location at all times, such as a navigation app
- Apps that support Voice over Internet Protocol (VoIP)
- Newsstand apps that need to download and process new content
- Apps that receive regular updates from external accessories

Apps that implement these services must declare the services they support and use system frameworks to implement the relevant aspects of those services. Declaring the services lets the system know which services you use, but in some cases it is the system frameworks that actually prevent your application from being suspended.

Declaring Your App's Supported Background Tasks

Support for some types of background execution must be declared in advance by the app that uses them. An app declares support for a service using its `Info.plist` file. Add the `UIBackgroundModes` key to your `Info.plist` file and set its value to an array containing one or more of the following strings:

- `audio`—The app plays audible content to the user while in the background. (This content includes streaming audio or video content using AirPlay.)
- `location`—The app keeps users informed of their location, even while it is running in the background.
- `voip`—The app provides the ability for the user to make phone calls using an Internet connection.
- `newsstand-content`—The app is a Newsstand app that downloads and processes magazine or newspaper content in the background.

- `external-accessory`—The app works with a hardware accessory that needs to deliver updates on a regular schedule through the External Accessory framework.
- `bluetooth-central`—The app works with a Bluetooth accessory that needs to deliver updates on a regular schedule through the Core Bluetooth framework.
- `bluetooth-peripheral`—The app supports Bluetooth communication in peripheral mode through the Core Bluetooth framework.

Each of the preceding values lets the system know that your app should be woken up at appropriate times to respond to relevant events. For example, an app that begins playing music and then moves to the background still needs execution time to fill the audio output buffers. Including the `audio` key tells the system frameworks that they should continue playing and make the necessary callbacks to the app at appropriate intervals. If the app does not include this key, any audio being played by the app stops when the app moves to the background.

Tracking the User's Location

There are several ways to track the user's location in the background, most of which do not actually require your app to run continuously in the background:

- The significant-change location service (Recommended)
- Foreground-only location services
- Background location services

The significant-change location service is highly recommended for apps that do not need high-precision location data. With this service, location updates are generated only when the user's location changes significantly; thus, it is ideal for social apps or apps that provide the user with noncritical, location-relevant information. If the app is suspended when an update occurs, the system wakes it up in the background to handle the update. If the app starts this service and is then terminated, the system relaunches the app automatically when a new location becomes available. This service is available in iOS 4 and later, and it is available only on devices that contain a cellular radio.

The foreground-only and background location services both use the standard location Core Location service to retrieve location data. The only difference is that the foreground-only location services stop delivering updates if the app is ever suspended, which is likely to happen if the app does not support other background services or tasks. Foreground-only location services are intended for apps that only need location data while they are in the foreground.

An app that provides continuous location updates to the user (even when in the background) can enable background location services by including the `UIBackgroundModes` key (with the `location` value) in its `Info.plist` file. The inclusion of this value in the `UIBackgroundModes` key does not preclude the system

from suspending the app, but it does tell the system that it should wake up the app whenever there is new location data to deliver. Thus, this key effectively lets the app run in the background to process location updates whenever they occur.

> **Important:** You are encouraged to use the standard services sparingly or use the significant location change service instead. Location services require the active use of an iOS device's onboard radio hardware. Running this hardware continuously can consume a significant amount of power. If your app does not need to provide precise and continuous location information to the user, it is best to minimize the use of location services.

For information about how to use each of the different location services in your app, see *Location Awareness Programming Guide*.

Playing Background Audio

An app that plays audio continuously (even while the app is running in the background) can register as a background audio app by including the `UIBackgroundModes` key (with the value `audio`) in its `Info.plist` file. Apps that include this key must play audible content to the user while in the background.

Typical examples of background audio apps include:

- Music player apps
- Apps that support audio or video playback over AirPlay
- VoIP apps

When the `UIBackgroundModes` key contains the `audio` value, the system's media frameworks automatically prevent the corresponding app from being suspended when it moves to the background. As long as it is playing audio or video content, the app continues to run in the background. However, if the app stops playing the audio or video, the system suspends it.

You can use any of the system audio frameworks to initiate the playback of background audio, and the process for using those frameworks is unchanged. (For video playback over AirPlay, you can use the Media Player or AV Foundation framework to present your video.) Because your app is not suspended while playing media files, callbacks operate normally while your app is in the background. In your callbacks, though, you should do only the work necessary to provide data for playback. For example, a streaming audio app would need to download the music stream data from its server and push the current audio samples out for playback. You should not perform any extraneous tasks that are unrelated to playback.

Because more than one app may support audio, the system limits which apps can play audio at any given time. The foreground app always has permission to play audio. In addition, one or more background apps may also be allowed to play some audio content depending on the configuration of their audio session objects. You

should always configure your app's audio session object appropriately and work carefully with the system frameworks to handle interruptions and other types of audio-related notifications. For information on how to configure audio session objects for background execution, see *Audio Session Programming Guide*.

Implementing a VoIP App

A **Voice over Internet Protocol (VoIP)** app allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an app needs to maintain a persistent network connection to its associated service so that it can receive incoming calls and other relevant data. Rather than keep VoIP apps awake all the time, the system allows them to be suspended and provides facilities for monitoring their sockets for them. When incoming traffic is detected, the system wakes up the VoIP app and returns control of its sockets to it.

To configure a VoIP app, you must do the following:

1. Add the `UIBackgroundModes` key to your app's `Info.plist` file. Set the value of this key to an array that includes the `voip` value.
2. Configure one of the app's sockets for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to install a handler to be executed periodically. Your app can use this handler to maintain its service connection.
4. Configure your audio session to handle transitions to and from active use.

Including the `voip` value in the `UIBackgroundModes` key lets the system know that it should allow the app to run in the background as needed to manage its network sockets. An app with this key is also relaunched in the background immediately after system boot to ensure that the VoIP services are always available.

Most VoIP apps also need to be configured as background audio apps to deliver audio while in the background. Therefore, you should include both the `audio` and `voip` values to the `UIBackgroundModes` key. If you do not do this, your app cannot play audio while it is in the background. For more information about the `UIBackgroundModes` key, see *Information Property List Key Reference*.

For specific information about the steps you must take to implement a VoIP app, see "Tips for Developing a VoIP App" (page 115).

Downloading Newsstand Content in the Background

A Newsstand app that downloads new magazine or newspaper issues can register to perform those downloads in the background. When your server sends a push notification to indicate that a new issue is available, the system checks to see whether your app has the `UIBackgroundModes` key with the `newsstand-content` value. If it does, the system launches your app, if it is not already running, so that it can initiate the downloading of the new issue.

When you use the Newsstand Kit framework to initiate a download, the system handles the download process for your app. The system continues to download the file even if your app is suspended or terminated. When the download operation is complete, the system transfers the file to your app sandbox and notifies your app. If the app is not running, this notification wakes it up and gives it a chance to process the newly downloaded file. If there are errors during the download process, your app is similarly woken up to handle them.

For information about how to download content using the Newsstand Kit framework, see *Newsstand Kit Framework Reference*.

Communicating with an External Accessory

Apps that work with external accessories can ask to be woken up if the accessory delivers an update when the app is suspended. This support is important for some types of accessories that deliver data at regular intervals, such as heart-rate monitors. When an app includes the `UIBackgroundModes` key with the `external-accessory` value in its `Info.plist` file, the external accessory framework keeps open any active sessions for the corresponding accessories. (In iOS 4 and earlier, these sessions are closed automatically when the app is suspended.) In addition, new data arriving from the accessory causes the system to wake up the app to process that data. The system also wakes up the app to process accessory connection and disconnection notifications.

Any app that supports the background processing of accessory updates must follow a few basic guidelines:

- Apps must provide an interface that allows the user to start and stop the delivery of accessory update events. That interface should then open or close the accessory session as appropriate.
- Upon being woken up, the app has around 10 seconds to process the data. Ideally, it should process the data as fast as possible and allow itself to be suspended again. However, if more time is needed, the app can use the `beginBackgroundTaskWithExpirationHandler:` method to request additional time; it should do so only when absolutely necessary, though.

Communicating with a Bluetooth Accessory

Apps that work with Bluetooth peripherals can ask to be woken up if the peripheral delivers an update when the app is suspended. This support is important for Bluetooth-le accessories that deliver data at regular intervals, such as a Bluetooth heart rate belt. When an app includes the `UIBackgroundModes` key with the `bluetooth-central` value in its `Info.plist` file, the Core Bluetooth framework keeps open any active sessions for the corresponding peripheral. In addition, new data arriving from the peripheral causes the system to wake up the app so that it can process the data. The system also wakes up the app to process accessory connection and disconnection notifications.

In iOS 6, an app can also operate in peripheral mode with Bluetooth accessories. If the app wants to respond to accessory-related changes using peripheral mode in the background, it must link against the Core Bluetooth framework and include the `UIBackgroundModes` key with the `bluetooth-peripheral` value in its `Info.plist` file. This key lets the Core Bluetooth framework wake the app up briefly in the background so that it can handle accessory-related requests. Apps woken up for these events should process them and return as quickly as possible so that the app can be suspended again.

Any app that supports the background processing of Bluetooth data must be session-based and follow a few basic guidelines:

- Apps must provide an interface that allows the user to start and stop the delivery of Bluetooth events. That interface should then open or close the session as appropriate.
- Upon being woken up, the app has around 10 seconds to process the data. Ideally, it should process the data as fast as possible and allow itself to be suspended again. However, if more time is needed, the app can use the `beginBackgroundTaskWithExpirationHandler:` method to request additional time; it should do so only when absolutely necessary, though.

Being a Responsible Background App

The foreground app always has precedence over background apps when it comes to the use of system resources and hardware. Apps running in the background need to be prepared for this discrepancy and adjust their behavior when running in the background. Specifically, apps moving to the background should follow these guidelines:

- **Do not make any OpenGL ES calls from your code.** You must not create an `EAGLContext` object or issue any OpenGL ES drawing commands of any kind while running in the background. Using these calls causes your app to be killed immediately. Apps must also ensure that any previously submitted commands have completed before moving to the background. For information about how to handle OpenGL ES when moving to and from the background, see "Implementing a Multitasking-aware OpenGL ES Application" in *OpenGL ES Programming Guide for iOS*.
- **Cancel any Bonjour-related services before being suspended.** When your app moves to the background, and before it is suspended, it should unregister from Bonjour and close listening sockets associated with any network services. A suspended app cannot respond to incoming service requests anyway. Closing out those services prevents them from appearing to be available when they actually are not. If you do not close out Bonjour services yourself, the system closes out those services automatically when your app is suspended.

- **Be prepared to handle connection failures in your network-based sockets.** The system may tear down socket connections while your app is suspended for any number of reasons. As long as your socket-based code is prepared for other types of network failures, such as a lost signal or network transition, this should not lead to any unusual problems. When your app resumes, if it encounters a failure upon using a socket, simply reestablish the connection.
- **Save your app state before moving to the background.** During low-memory conditions, background apps may be purged from memory to free up space. Suspended apps are purged first, and no notice is given to the app before it is purged. As a result, apps should take advantage of the state preservation mechanism in iOS 6 and later to save their interface state to disk. For information about how to support this feature, see "State Preservation and Restoration" (page 67).
- **Remove strong references to unneeded objects when moving to the background.** If your app maintains a large in-memory cache of objects (especially images), remove all strong references to those caches when moving to the background. For more information, see "Memory Usage for Background Apps" (page 47).
- **Stop using shared system resources before being suspended.** Apps that interact with shared system resources such as the Address Book or calendar databases should stop using those resources before being suspended. Priority for such resources always goes to the foreground app. When your app is suspended, if it is found to be using a shared resource, the app is killed.
- **Avoid updating your windows and views.** While in the background, your app's windows and views are not visible, so you should not try to update them. Although creating and manipulating window and view objects in the background does not cause your app to be killed, consider postponing this work until you return to the foreground.
- **Respond to connect and disconnect notifications for external accessories.** For apps that communicate with external accessories, the system automatically sends a disconnection notification when the app moves to the background. The app must register for this notification and use it to close out the current accessory session. When the app moves back to the foreground, a matching connection notification is sent, giving the app a chance to reconnect. For more information on handling accessory connection and disconnection notifications, see *External Accessory Programming Topics*.
- **Clean up resources for active alerts when moving to the background.** In order to preserve context when switching between apps, the system does not automatically dismiss action sheets (`UIActionSheet`) or alert views (`UIAlertView`) when your app moves to the background. It is up to you to provide the appropriate cleanup behavior prior to moving to the background. For example, you might want to cancel the action sheet or alert view programmatically or save enough contextual information to restore the view later (in cases where your app is terminated).

  For apps linked against a version of iOS earlier than 4.0, action sheets and alerts are still dismissed at quit time so that your app's cancellation handler has a chance to run.

- **Remove sensitive information from views before moving to the background.** When an app transitions to the background, the system takes a snapshot of the app's main window, which it then presents briefly when transitioning your app back to the foreground. Before returning from your `applicationDidEnterBackground:` method, you should hide or obscure passwords and other sensitive personal information that might be captured as part of the snapshot.
- **Do minimal work while running in the background.** The execution time given to background apps is more constrained than the amount of time given to the foreground app. If your app plays background audio or monitors location changes, you should focus on that task only and defer any nonessential tasks until later. Apps that spend too much time executing in the background can be throttled back by the system or killed.

If you are implementing a background audio app, or any other type of app that is allowed to run in the background, your app responds to incoming messages in the usual way. In other words, the system may notify your app of low-memory warnings when they occur. And in situations where the system needs to terminate apps to free even more memory, the app calls its delegate's `applicationWillTerminate:` method to perform any final tasks before exiting.

Opting out of Background Execution

If you do not want your app to run in the background at all, you can explicitly opt out of background by adding the `UIApplicationExitsOnSuspend` key (with the value `YES`) to your app's `Info.plist` file. When an app opts out, it cycles between the not-running, inactive, and active states and never enters the background or suspended states. When the user presses the Home button to quit the app, the `applicationWillTerminate:` method of the app delegate is called and the app has approximately 5 seconds to clean up and exit before it is terminated and moved back to the not-running state.

Opting out of background execution is strongly discouraged but may be the preferred option under certain conditions. Specifically, if coding for the background adds significant complexity to your app, terminating the app might be a simpler solution. Also, if your app consumes a large amount of memory and cannot easily release any of it, the system might kill your app quickly anyway to make room for other apps. Thus, opting to terminate, instead of switching to the background, might yield the same results and save you development time and effort.

> **Note:** Explicitly opting out of background execution is necessary only if your app is linked against iOS SDK 4 and later. Apps linked against earlier versions of the SDK do not support background execution as a rule and therefore do not need to opt out explicitly.

For more information about the keys you can include in your app's `Info.plist` file, see *Information Property List Key Reference*.

Concurrency and Secondary Threads

The system creates your app's main thread but your app can create additional threads as needed to perform other tasks. The preferred way to create threads is to let the system do it for you by using Grand Central Dispatch queues and operation queues. Both types of queue provide an asynchronous execution model for tasks that you define. When you submit a task to a queue, the system spins up a thread and executes your task on that thread. Letting the system manage the threads simplifies your code and allows the system to manage the threads in the most efficient way available.

You should use queues whenever possible to move work off of your app's main thread. Because the main thread is responsible for processing touch and drawing events, you should never perform lengthy tasks on it. For example, you should never wait for a network response on your app's main thread. It is much better to make the request asynchronously using a queue and process the results when they arrive.

Another good time to move tasks to secondary threads is launch time. Launched apps have a limited amount of time (around 5 seconds) to do their initialization and start processing events. If you have launch-time tasks that can be deferred or executed on a secondary thread, you should move them off the main thread right away and use the main thread only to present your user interface and start handling events.

For more information about using dispatch and operation queues to execute tasks, see *Concurrency Programming Guide*.

State Preservation and Restoration

Even if your app supports background execution, it cannot run forever. At some point, the system might need to terminate your app to free up memory for the current foreground app. However, the user should never have to care if an app is already running or was terminated. From the user's perspective, quitting an app should just seem like a temporary interruption. When the user returns to an app, that app should always return the user to the last point of use, so that the user can continue with whatever task was in progress. This behavior provides a better experience for the user and with the state restoration support built in to UIKit is relatively easy to achieve.

The state preservation system in UIKit provides a simple but flexible infrastructure for preserving and restoring the state of your app's view controllers and views. The job of the infrastructure is to drive the preservation and restoration processes at the appropriate times. To do that, UIKit needs help from your app. Only you understand the content of your app, and so only you can write the code needed to save and restore that content. And when you update your app's UI, only you know how to map older preserved content to the newer objects in your interface.

There are three places where you have to think about state preservation in your app:

- Your app delegate object, which manages the app's top-level state
- Your app's view controller objects, which manage the overall state for your app's user interface
- Your app's custom views, which might have some custom data that needs to be preserved

UIKit allows you to choose which parts of your user interface you want to preserve. And if you already have custom code for handling state preservation, you can continue to use that code and migrate portions to the UIKit state preservation system as needed.

The Preservation and Restoration Process

State preservation and restoration is an opt-in feature and requires help from your app to work. Your app essentially provides UIKit with a list of objects and lets UIKit handle the tedious aspects of preserving and restoring those objects at appropriate times. Because UIKit handles so much of the process, it helps to understand what it does behind the scenes so that you know how your custom code fits into the overall scheme.

When thinking about state preservation and restoration, it helps to separate the two processes first. State preservation occurs when your app moves to the background. At that time, UIKit queries your app's views and view controllers to see which ones should be preserved and which ones should not. For each object that should be preserved, UIKit writes preservation-related data to an on-disk file. The next time your app launches from scratch, UIKit looks for that file and, if it is present, uses it to try and restore your app's state. During the restoration process, UIKit uses the preserved data to reconstitute your interface. The creation of actual objects is handled by your code. Because your app might load objects from a storyboard file automatically, only your code knows which objects need to be created and which might already exist and can simply be returned. After those objects are created, UIKit uses the on-disk data to restore the objects to their previous state.

During the preservation and restoration process, your app has a handful of responsibilities.

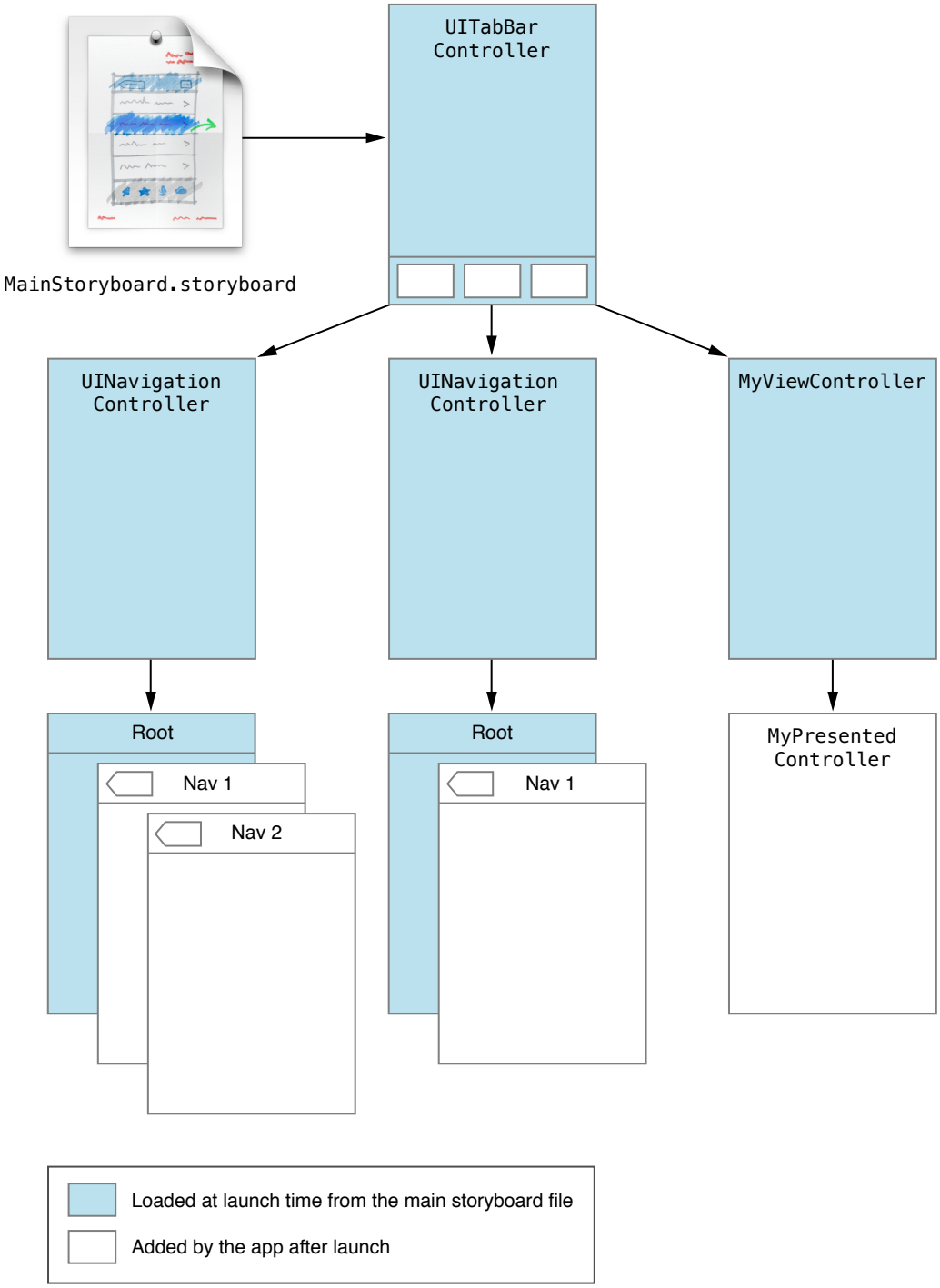
- During preservation, your app is responsible for:
    - Telling UIKit that it supports state preservation.
    - Telling UIKit which view controllers and views should be preserved.
    - Encoding relevant data for any preserved objects.
- During restoration, your app is responsible for:
    - Telling UIKit that it supports state restoration.
    - Providing (or creating) the objects that are requested by UIKit.
    - Decoding the state of your preserved objects and using it to return the object to its previous state.

Of your app's responsibilities, the most significant are telling UIKit which objects to preserve and providing those objects during subsequent launches. Those two behaviors are where you should spend most of your time when designing your app's preservation and restoration code. They are also where you have the most control over the actual process. To understand why that is the case, it helps to look at an example.

Figure 4-1 shows the view controller hierarchy of a tab bar interface after the user has interacted with several of the tabs. As you can see, some of the view controllers are loaded automatically as part of the app's main storyboard file but some of the view controllers were presented or pushed onto the view controllers in different

tabs. Without state restoration, only the view controllers from the main storyboard file would be restored during subsequent launches. By adding support for state restoration to your app, you can preserve all of the view controllers.

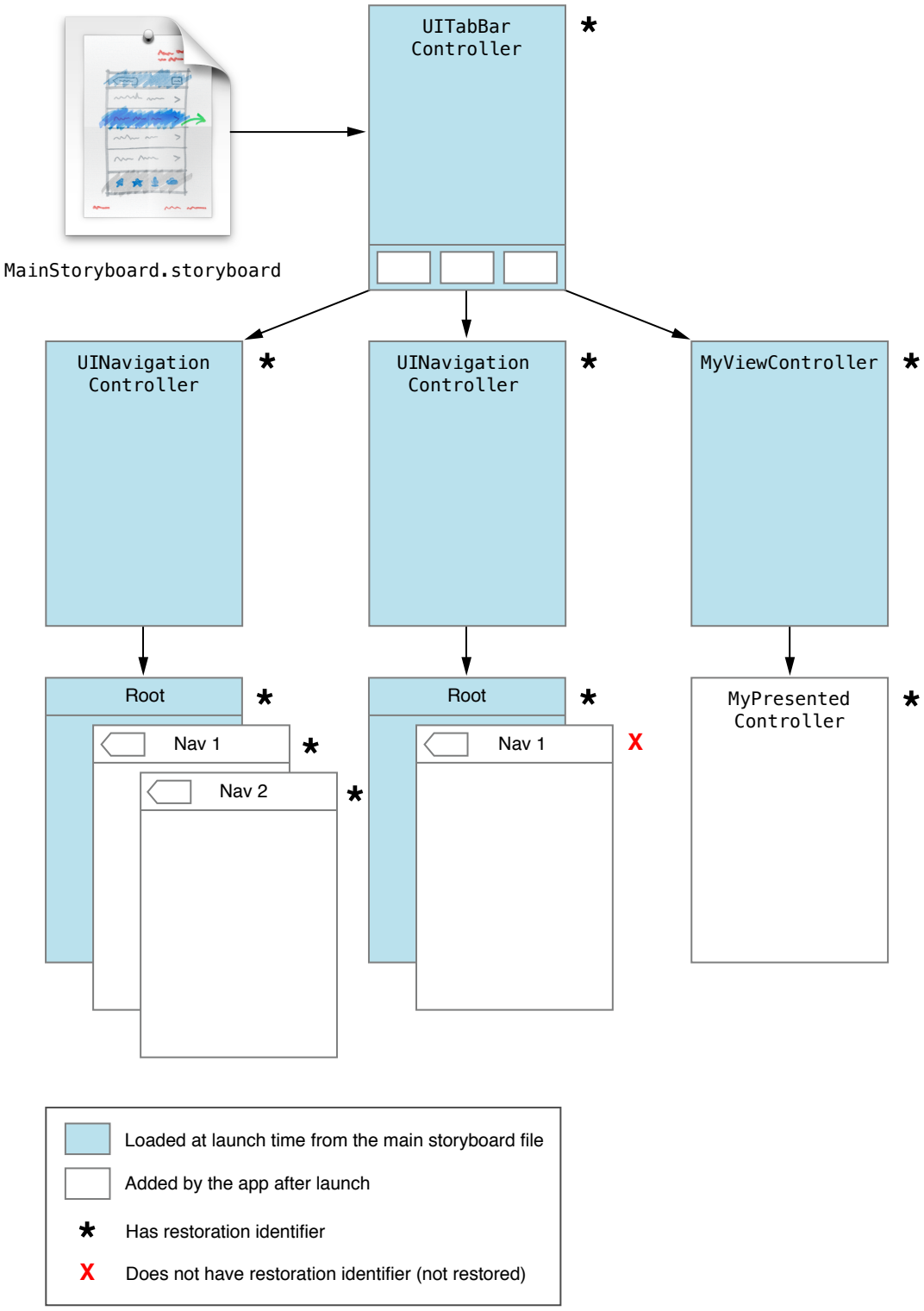
Figure 4-1 A sample view controller hierarchy



UIKit preserves only those objects that have a restoration identifier. A **restoration identifier** is a string that identifies the view or view controller to UIKit and your app. The value of this string is significant only to your code but the presence of this string tells UIKit that it needs to preserve the tagged object. During the preservation process, UIKit walks your app's view controller hierarchy and preserves all objects that have a restoration

identifier. If a view controller does not have a restoration identifier, that view controller and all of its views and child view controllers are not preserved. Figure 4-2 shows an updated version of the previous view hierarchy, now with restoration identifies applied to most (but not all) of the view controllers.

Figure 4-2 Adding restoration identifies to view controllers



Depending on your app, it might or might not make sense to preserve every view controller. If a view controller presents transitory information, you might not want to return to that same point on restore, opting instead to return the user to a more stable point in your interface.

For each view controller you choose to preserve, you also need to decide on how you want to restore it later. UIKit offers two ways to recreate objects. You can let your app delegate recreate it or you can assign a restoration class to the view controller and let that class recreate it. A **restoration class** implements the `UIViewControllerRestoration` protocol and is responsible for finding or creating a designated object at restore time. Here are some tips for when to use each one:

- **If the view controller is always loaded from your app's main storyboard file at launch time, do not assign a restoration class.** Instead, let your app delegate find the object or take advantage of UIKit's support for implicitly finding restored objects.
- **For view controllers that are not loaded from your main storyboard file at launch time, assign a restoration class.** The simplest option is to make each view controller its own restoration class.

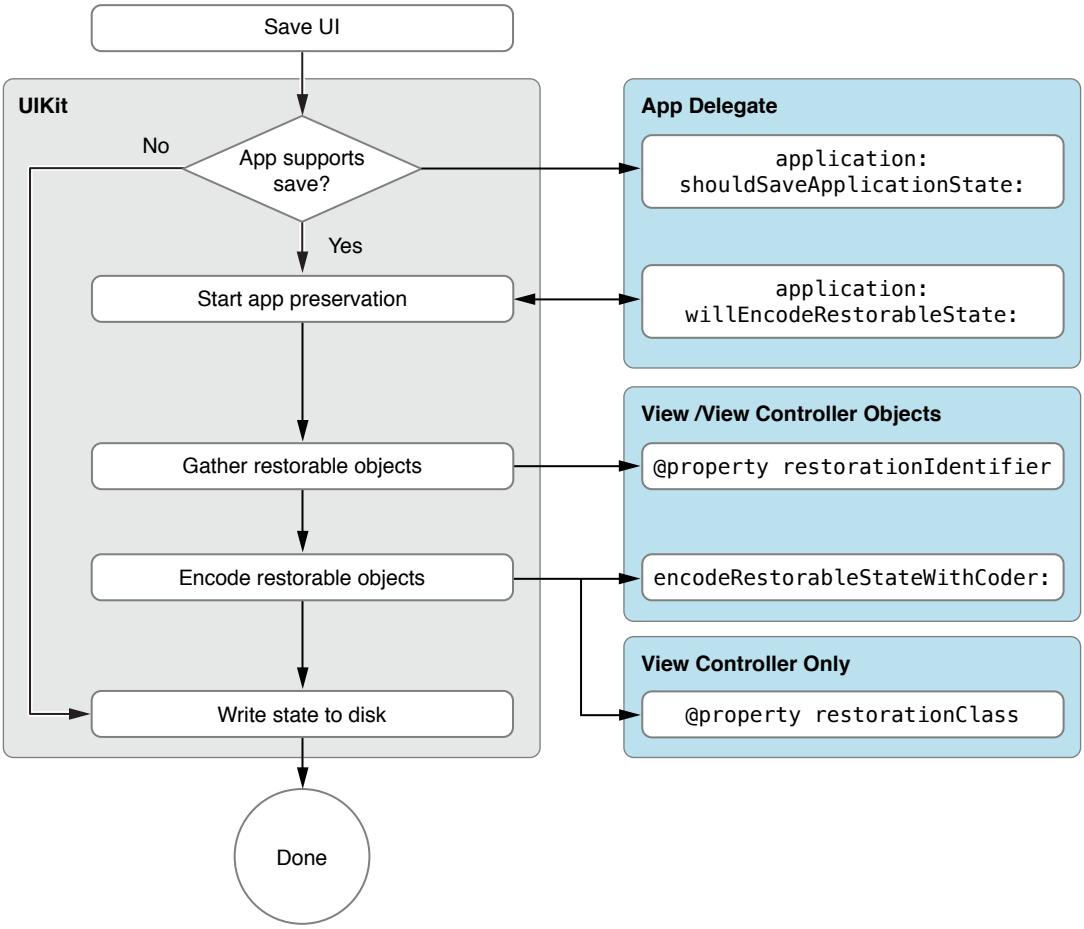
During the preservation process, UIKit identifies the objects to save and writes each affected object's state to disk. Each view controller object is given a chance to write out any data it wants to save. For example, a tab view controller saves the identity of the selected tab. UIKit also saves information such as the view controller's restoration class to disk. And if any of the view controller's views has a restoration identifier, UIKit asks them to save their state information too.

The next time the app is launched, UIKit loads the app's main storyboard or nib file as usual, calls the app delegate's `application:willFinishLaunchingWithOptions:` method, and then tries to restore the app's previous state. The first thing it does is ask your app to provide the set of view controller objects that match the ones that were preserved. If a given view controller had an assigned restoration class, that class is asked to provide the object; otherwise, the app delegate is asked to provide it.

Flow of the Preservation Process

Figure 4-3 shows the high-level events that happen during state preservation and shows how the objects of your app are affected. Before preservation even occurs, UIKit asks your app delegate if it *should* occur by calling the `application:shouldSaveApplicationState:` method. If that method returns `YES`, UIKit begins gathering and encoding your app's views and view controllers. When it is finished, it writes the encoded data to disk.

Figure 4-3 High-level flow interface preservation



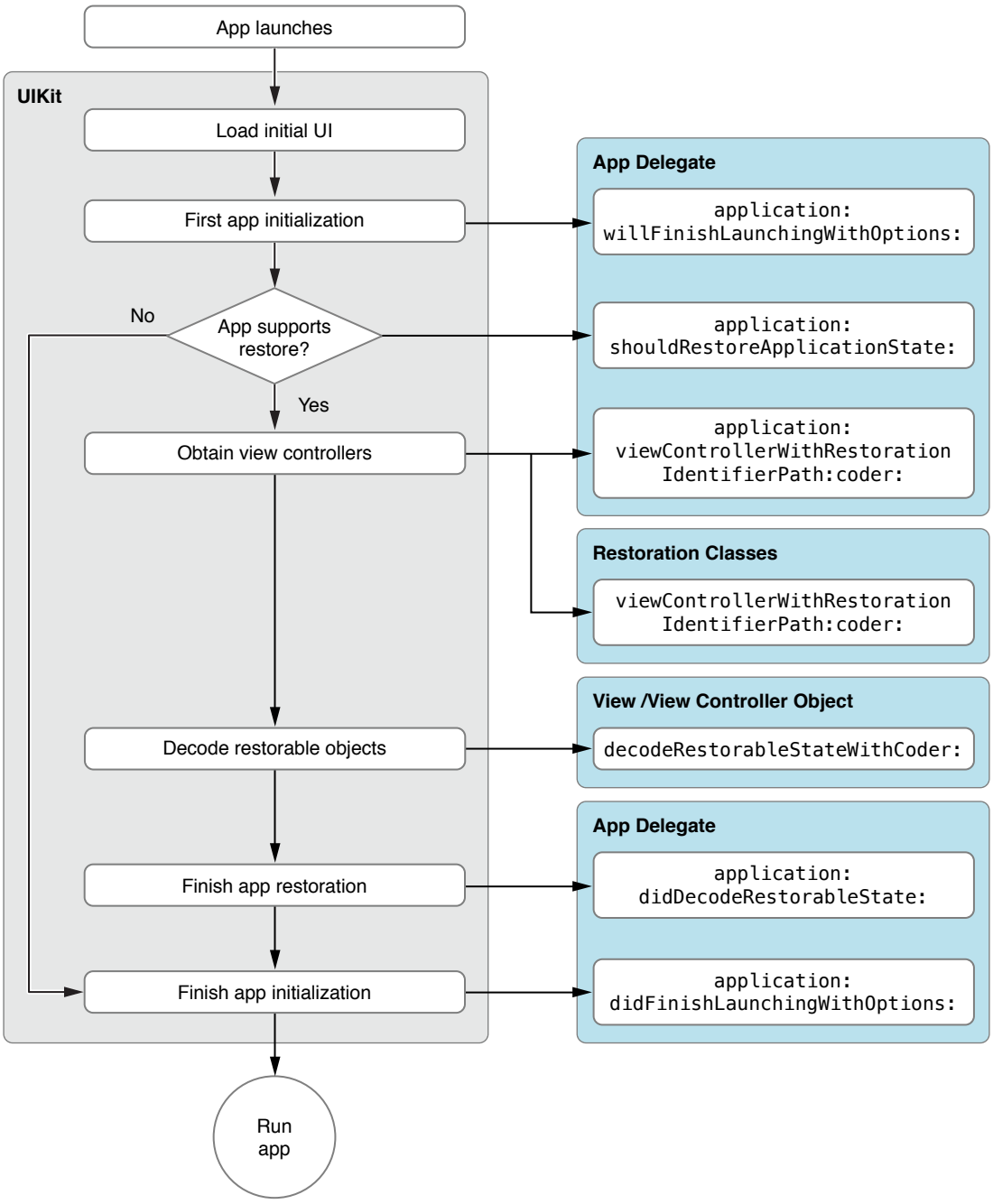
The next time your app launches, the system automatically looks for a preserved state file, and if present, uses it to restore your interface. Because this state information is only relevant between the previous and current launch cycles of your app, the file is typically discarded after your app finishes launching. The file is also discarded any time there is an error restoring your app. For example, if your app crashes during the restoration process, the system automatically throws away the state information during the next launch cycle to avoid another crash.

Flow of the Restoration Process

Figure 4-4 shows the high-level events that happen during state restoration and shows how the objects of your app are affected. After the standard initialization and UI loading is complete, UIKit asks your app delegate if state restoration should occur at all by calling the `application:shouldRestoreApplicationState:`

method. This is your app delegate's opportunity to examine the preserved data and determine if state restoration is possible. If it is, UIKit uses the app delegate and restoration classes to obtain references to your app's view controllers. Each object is then provided with the data it needs to restore itself to its previous state.

Figure 4-4 High-level flow for restoring your user interface



Although UIKit helps restore the individual view controllers, it does not automatically restore the relationships between those view controllers. Instead, each view controller is responsible for encoding enough state information to return itself to its previous state. For example, a navigation controller encodes information about the order of the view controllers on its navigation stack. It then uses this information later to return those view controllers to their previous positions on the stack. Other view controllers that have embedded child view controllers are similarly responsible for encoding any information they need to restore their children later.

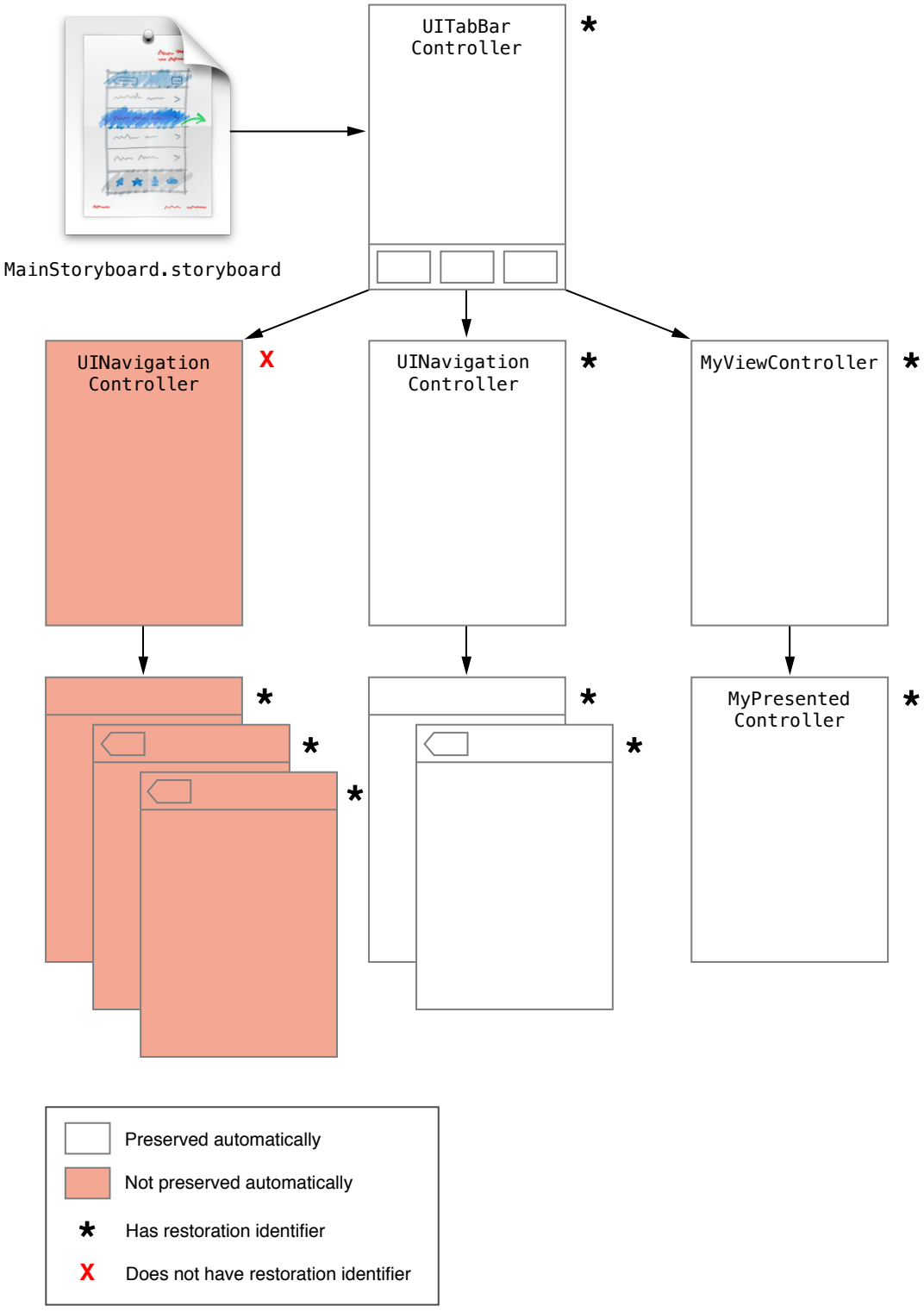
> **Note:** Not all view controllers need to encode their child view controllers. For example, tab bar controllers do not encode information about their child view controllers. Instead, it is assumed that your app follows the usual pattern of creating the appropriate child view controllers prior to creating the tab bar controller itself.

Because you are responsible for recreating your app's view controllers, you have some flexibility to change your interface during the restoration process. For example, you could reorder the tabs in a tab bar controller and still use the preserved data to return each tab to its previous state. Of course, if you make dramatic changes to your view controller hierarchy, such as during an app update, you might not be able to use the preserved data.

What Happens When You Exclude Groups of View Controllers?

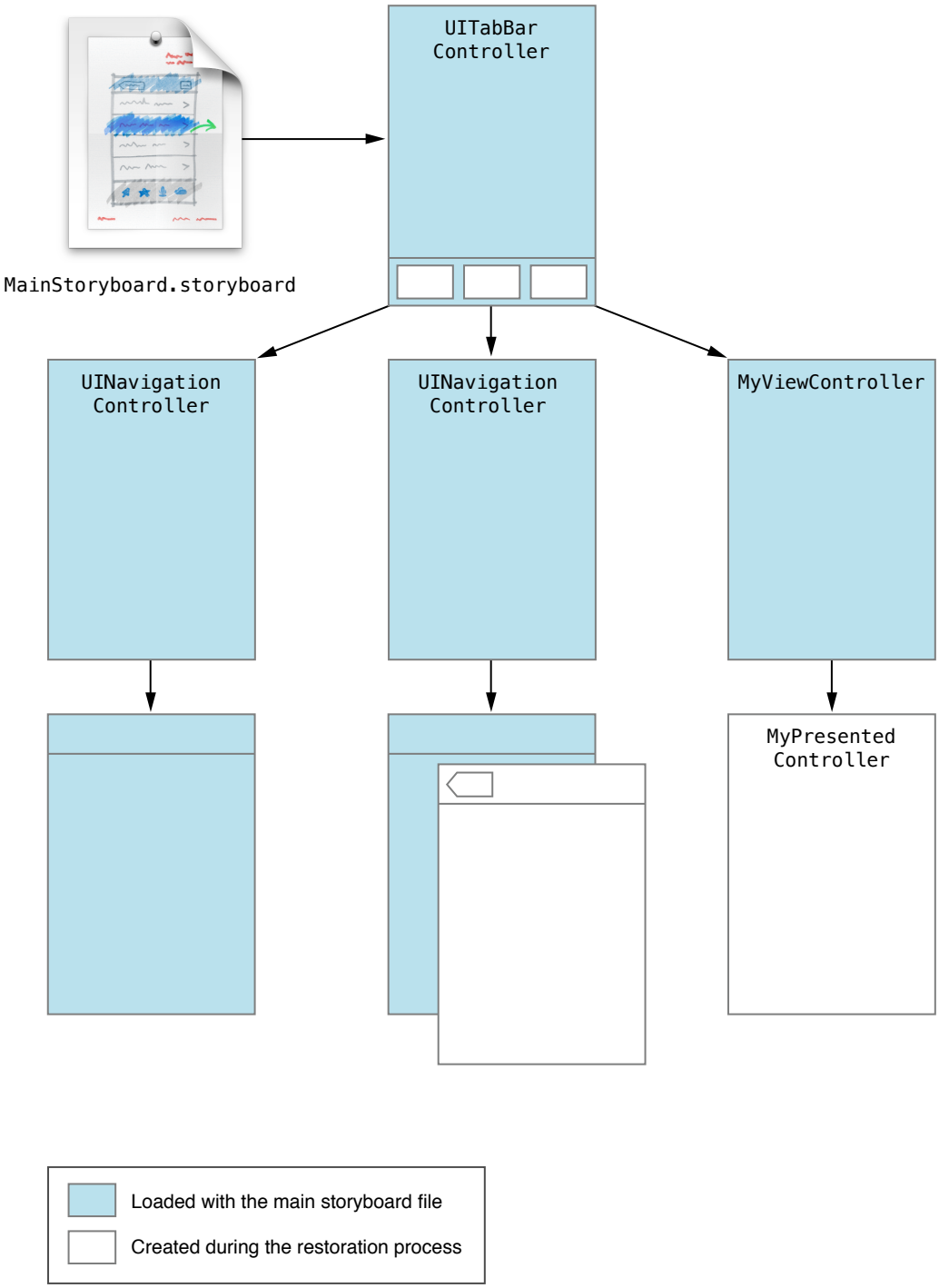
When the restoration identifier of a view controller is `nil`, that view controller and any child view controllers it manages are not preserved automatically. For example, in Figure 4-6, because a navigation controller did not have a restoration identifier, it and all of its child view controllers and views are omitted from the preserved data.

Figure 4-5 Excluding view controllers from the automatic preservation process



Even if you decide not to preserve view controllers, that does not mean all of those view controllers disappear from the view hierarchy altogether. At launch time, your app might still create the view controllers as part of its default setup. For example, if any view controllers are loaded automatically from your app's storyboard file, they would still appear, albeit in their default configuration, as shown in Figure 4-6.

Figure 4-6 Loading the default set of view controllers



Something else to realize is that even if a view controller is not preserved automatically, you can still encode a reference to that view controller and preserve it manually. In Figure 4-5 (page 79), the three child view controllers of the first navigation controller have restoration identifiers, even though there parent navigation controller does not. If your app delegate (or any preserved object) encodes a reference to those view controllers, their state is preserved. Even though their order in the navigation controller is not saved, you could still use those references to recreate the view controllers and install them in the navigation controller during subsequent launch cycles.

Checklist for Implementing State Preservation and Restoration

Supporting state preservation and restoration requires modifying your app delegate and view controller objects to encode and decode the state information. If your app has any custom views that also have preservable state information, you need to modify those objects too.

When adding state preservation and restoration to your code, use the following list to remind you of the code you need to write.

- (Required) Implement the `application:shouldSaveApplicationState:` and `application:shouldRestoreApplicationState:` methods in your app delegate; see "Enabling State Preservation and Restoration in Your App" (page 82).
- (Required) Assign restoration identifiers to each view controller you want to preserve. a non empty string to their `restorationIdentifier` property; see "Marking Your View Controllers for Preservation" (page 83).

  If you want to save the state of specific views too, assign non empty strings to their `restorationIdentifier` properties; see "Preserving the State of Your Views" (page 86).
- (Required) Assign restoration classes to the appropriate view controllers. (If you do not do this, your app delegate is asked to provide the corresponding view controller at restore time.) See "Restoring Your View Controllers at Launch Time" (page 83).
- (Recommended) Encode and decode the state of your views and view controllers using the `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods of those objects; see "Encoding and Decoding Your View Controller's State" (page 85).
- Encode and decode any version information or additional state information for your app using the `application:willEncodeRestorableStateWithCoder:` and `application:didDecodeRestorableStateWithCoder:` methods of your app delegate; see "Preserving Your App's High-Level State" (page 89).

- Objects that act as data sources for table views and collection views should implement the `UIDataSourceModelAssociation` protocol. Although not required, this protocol helps preserve the selected and visible items in those types of views. See "Implementing Preservation-Friendly Data Sources" (page 89).

Enabling State Preservation and Restoration in Your App

State preservation and restoration is not an automatic feature and apps must opt-in to use it. Apps indicate their support for the feature by implementing the following methods in their app delegate:

```
application:shouldSaveApplicationState:
application:shouldRestoreApplicationState:
```

Normally, your implementations of these methods just return `YES` to indicate that state preservation and restoration can occur. However, apps that want to preserve and restore their state conditionally can return `NO` in situations where the operations should not occur. For example, after releasing an update to your app, you might want to return `NO` from your `application:shouldRestoreApplicationState:` method if your app is unable to usefully restore the state from a previous version.

Preserving the State of Your View Controllers

Preserving the state of your app's view controllers should be your main goal. View controllers define the structure of your user interface. They manage the views needed to present that interface and they coordinate the getting and setting of the data that backs those views. To preserve the state of a single view controller, you must do the following:

- (Required) Assign a restoration identifier to the view controller; see "Marking Your View Controllers for Preservation" (page 83).
- (Required) Provide code to create or locate new view controller objects at launch time; see "Restoring Your View Controllers at Launch Time" (page 83).
- (Optional) Implement the `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods to encode and restore any state information that cannot be recreated during a subsequent launch; see "Encoding and Decoding Your View Controller's State" (page 85).

Marking Your View Controllers for Preservation

UIKit preserves only those view controllers whose `restorationIdentifier` property contains a valid string object. For view controllers that you know you want to preserve, set the value of this property when you initialize the view controller object. If you load the view controller from a storyboard or nib file, you can set the restoration identifier there.

Choosing an appropriate value for restoration identifiers is important. During the restoration process, your code uses the restoration identifier to determine which view controller to retrieve or create. If every view controller object is based on a different class, you can use the class name for the restoration identifier. However, if your view controller hierarchy contains multiple instances of the same class, you might need to choose different names based on each view usage.

When it asks you to provide a view controller, UIKit provides you with the restoration path of the view controller object. A **restoration path** is the sequence of restoration identifiers starting at the root view controller and walking down the view controller hierarchy to the current object. For example, imagine you have a tab bar controller whose restoration identifier is `TabBarControllerID`, and the first tab contains a navigation controller whose identifier is `NavControllerID` and whose root view controller's identifier is `MyViewController`. The full restoration path for the root view controller would be `TabBarControllerID/NavControllerID/MyViewController`.

The restoration path for every object must be unique. If a view controller has two child view controllers, each child must have a different restoration identifier. However, two view controllers with different parent objects may use the same restoration identifier because the rest of the restoration path provides the needed uniqueness. Some UIKit view controllers, such as navigation controllers, automatically disambiguate their child view controllers, allowing you to use the same restoration identifiers for each child. For more information about the behavior of a given view controller, see the corresponding class reference.

At restore time, you use the provided restoration path to determine which view controller to return to UIKit. For more information on how you use restoration identifiers and restoration paths to restore view controllers, see "Restoring Your View Controllers at Launch Time" (page 83).

Restoring Your View Controllers at Launch Time

During the restoration process, UIKit asks your app to create (or locate) the view controller objects that comprise your preserved user interface. UIKit adheres to the following process when trying to locate view controllers:

1. **If the view controller had a restoration class, UIKit asks that class to provide the view controller.** UIKit calls the `viewControllerWithRestorationIdentifierPath:coder:` method of the associated restoration class to retrieve the view controller. If that method returns `nil`, it is assumed that the app does not want to recreate the view controller and UIKit stops looking for it.

2. **If the view controller did not have a restoration class, UIKit asks the app delegate to provide the view controller.** UIKit calls the `application:viewControllerWithRestorationIdentifierPath:coder:` method of your app delegate to look for view controllers without a restoration class. If that method returns `nil`, UIKit tries to find the view controller implicitly.
3. **If a view controller with the correct restoration path already exists, UIKit uses that object.** If your app creates view controllers at launch time (either programmatically or by loading them from a resource file) and assigns restoration identifiers to them, UIKit finds them implicitly through their restoration paths.
4. **If the view controller was originally loaded from a storyboard file, UIKit uses the saved storyboard information to locate and create it.** UIKit saves information about a view controller's storyboard inside the restoration archive. At restore time, it uses that information to locate the same storyboard file and instantiate the corresponding view controller if the view controller was not found by any other means.

It is worth noting that if you specify a restoration class for a view controller, UIKit does not try to find your view controller implicitly. If the `viewControllerWithRestorationIdentifierPath:coder:` method of your restoration class returns `nil`, UIKit stops trying to locate your view controller. This gives you control over whether you really want to create the view controller. If you do not specify a restoration class, UIKit does everything it can to find the view controller for you, creating it as necessary from your app's storyboard files.

If you choose to use a restoration class, the implementation of your `viewControllerWithRestorationIdentifierPath:coder:` method should create a new instance of the class, perform some minimal initialization, and return the resulting object. Listing 4-1 shows an example of how you might use this method to load a view controller from a storyboard. Because the view controller was originally loaded from a storyboard, this method uses the `UIStateRestorationViewControllerStoryboardKey` key to get the name of the original storyboard file from the archive. Note that this method does not try to configure the view controller's data fields. That step occurs later when the view controller's state is decoded.

Listing 4-1 Creating a new view controller during restoration

```
+ (UIViewController*) viewControllerWithRestorationIdentifierPath:(NSArray
*)identifierComponents
                          coder:(NSCoder *)coder {
   MyViewController* vc;
   NSString* storyboardName = [coder
decodeObjectForKey:UIStateRestorationViewControllerStoryboardKey];
   UIStoryboard* sb = [UIStoryboard storyboardWithName:storyboardName bundle:nil];
   if (storyboardName) {
      vc = (MyViewController*)[sb
instantiateViewControllerWithIdentifier:@"MyViewController"];
```

```
        thePush.restorationIdentifier = [identifierComponents lastObject];
        thePush.restorationClass = [MyViewController class];
    }
     return vc;
}
```

Reassigning the restoration identifier and restoration class, as in the preceding example, is a good habit to adopt when creating new view controllers. The simplest way to restore the restoration identifier is to grab the last item in the `identifierComponents` array and assign it to your view controller.

For objects that were already loaded from your app's main storyboard file at launch time, do not create a new instance of each object. Instead, implement the `application:viewControllerWithRestorationIdentifierPath:coder:` method of your app delegate and use it to return the appropriate objects or let UIKit find those objects implicitly.

Encoding and Decoding Your View Controller's State

For each object slated for preservation, UIKit calls the object's `encodeRestorableStateWithCoder:` method to give it a chance to save its state. During the decode process, a matching call to the `decodeRestorableStateWithCoder:` method is made to decode that state and apply it to the object. The implementation of these methods is optional, but recommended, for your view controllers. You can use them to save and restore the following types of information:

- References to any data being displayed (not the data itself)
- For a container view controller, references to its child view controllers
- Information about the current selection
- For view controllers with a user-configurable view, information about the current configuration of that view.

In your encode and decode methods, you can encode any values supported by the coder, including other objects. For all objects except views and view controllers, the object must adopt the `NSCoding` protocol and use the methods of that protocol to write its state. For views and view controllers, the coder does not use the methods of the `NSCoding` protocol to save the object's state. Instead, the coder saves the restoration identifier of the object and adds it to the list of preservable objects, which results in that object's `encodeRestorableStateWithCoder:` method being called.

The `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods of your view controllers should always call `super` at some point in their implementation. Calling `super` gives the parent class a chance to save and restore any additional information. Listing 4-2 shows a sample implementation of these methods that save a numerical value used to identify the specified view controller.

Listing 4-2 Encoding and decoding a view controller's state.

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];

    [coder encodeInt:self.number forKey:MyViewControllerNumber];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder {
    [super decodeRestorableStateWithCoder:coder];

    self.number = [coder decodeIntForKey:MyViewControllerNumber];
}
```

Coder objects are not shared during the encode and decode process. Each object with preservable state receives its own coder that it can use to read or write data. The use of unique coders means that you do not have to worry about key namespace collisions among your own objects. However, you must still avoid using some special key names that UIKit provides. Specifically, each coder contains the `UIApplicationStateRestorationBundleVersionKey` and `UIApplicationStateRestorationUserInterfaceIdiomKey` keys, which provide information about the bundle version and current user interface idiom. Coders associated with view controllers may also contain the `UIStateRestorationViewControllerStoryboardKey` key, which identifies the storyboard from which that view controller originated.

For more information about implementing your encode and decode methods for your view controllers, see *UIViewController Class Reference*.

Preserving the State of Your Views

If a view has state information worth preserving, you can save that state with the rest of your app's view controllers. Because they are usually configured by their owning view controller, most views do not need to save state information. The only time you need to save a view's state is when the view itself can be altered by

the user in a way that is independent of its data or the owning view controller. For example, scroll views save the current scroll position, which is information that is not interesting to the view controller but which does affect how the view presents itself.

To designate that a view's state should be saved, you do the following:

- Assign a valid string to the view's `restorationIdentifier` property.
- Use the view from a view controller that also has a valid restoration identifier.
- For table views and collection views, assign a data source that adopts the `UIDataSourceModelAssociation` protocol.

As with view controllers, assigning a restoration identifier to a view tells the system that the view object has state that your app wants to save. The restoration identifier can also be used to locate the view later.

Like view controllers, views define methods for encoding and decoding their custom state. If you create a view with state worth saving, you can use these methods to read and write any relevant data.

UIKit Views with Preservable State

In order to save the state of any view, including both custom and standard system views, you must assign a restoration identifier to the view. Views without a restoration identifier are not added to the list of preservable objects by UIKit.

The following UIKit views have state information that can be preserved:

- `UICollectionView`
- `UIImageView`
- `UIScrollView`
- `UITableView`
- `UITextField`
- `UITextView`
- `UIWebView`

Other frameworks may also have views with preservable state. For information about whether a view saves state information and what state it saves, see the reference for the corresponding class.

Preserving the State of a Custom View

If you are implementing a custom view that has restorable state, implement the `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods and use them to encode and decode that state. Use those methods to save only the data that cannot be easily reconfigured by other means. For example, use these methods to save data that is modified by user interactions with the view. Do not use these methods to save the data being presented by the view or any data that the owning view controller can configure easily.

Listing 4-3 shows an example of how to preserve and restore the selection for a custom view that contains editable text. In the example, the range is accessible using the `selectionRange` and `setSelectionRange:` methods, which are custom methods the view uses to manage the selection. Encoding the data only requires writing it to the provided coder object. Restoring the data requires reading it and applying it to the view.

Listing 4-3 Preserving the selection of a custom text view

```
// Preserve the text selection
- (void) encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];

    NSRange range = [self selectionRange];
    [coder encodeInt:range.length forKey:kMyTextViewSelectionRangeLength];
    [coder encodeInt:range.location forKey:kMyTextViewSelectionRangeLocation];
}

// Restore the text selection.
- (void) decodeRestorableStateWithCoder:(NSCoder *)coder {
   [super decodeRestorableStateWithCoder:coder];
   if ([coder containsValueForKey:kMyTextViewSelectionRangeLength] &&
            [coder containsValueForKey:kMyTextViewSelectionRangeLocation]) {
      NSRange range;
      range.length = [coder decodeIntForKey:kMyTextViewSelectionRangeLength];
      range.location = [coder decodeIntForKey:kMyTextViewSelectionRangeLocation];
      if (range.length > 0)
         [self setSelectionRange:range];
   }
}
```

Implementing Preservation-Friendly Data Sources

Because the data displayed by a table or collection view can change, both classes save information about the current selection and visible cells only if their data source implements the `UIDataSourceModelAssociation` protocol. This protocol provides a way for a table or collection view to identify the content it contains without relying on the index path of that content. Thus, regardless of where the data source places an item during the next launch cycle, the view still has all the information it needs to locate that item.

In order to implement the `UIDataSourceModelAssociation` protocol successfully, your data source object must be able to identify items between subsequent launches of the app. This means that any identification scheme you devise must be invariant for a given piece of data. This is essential because the data source must be able to retrieve the same piece of data for the same identifier each time it is requested. Implementing the protocol itself is a matter of mapping from a data item to its unique ID and back again.

Apps that use Core Data can implement the protocol by taking advantage of object identifiers. Each object in a Core Data store has a unique object identifier that can be converted into a URI and used to locate the object later. If your app does not use Core Data, you need to devise your own form of unique identifiers if you want to support state preservation for your views.

> **Note:** Remember that implementing the `UIDataSourceModelAssociation` protocol is only necessary to preserve attributes such as the current selection in a table or collection view. This protocol is not used to preserve the actual data managed by your data source. It is your app's responsibility to ensure that its data is saved at appropriate times.

Preserving Your App's High-Level State

In addition to the data preserved by your app's view controllers and views, UIKit provides hooks for you to save any miscellaneous data needed by your app. Specifically, the `UIApplicationDelegate` protocol includes the following methods for you to override:

- `application:willEncodeRestorableStateWithCoder:`
- `application:didDecodeRestorableStateWithCoder:`

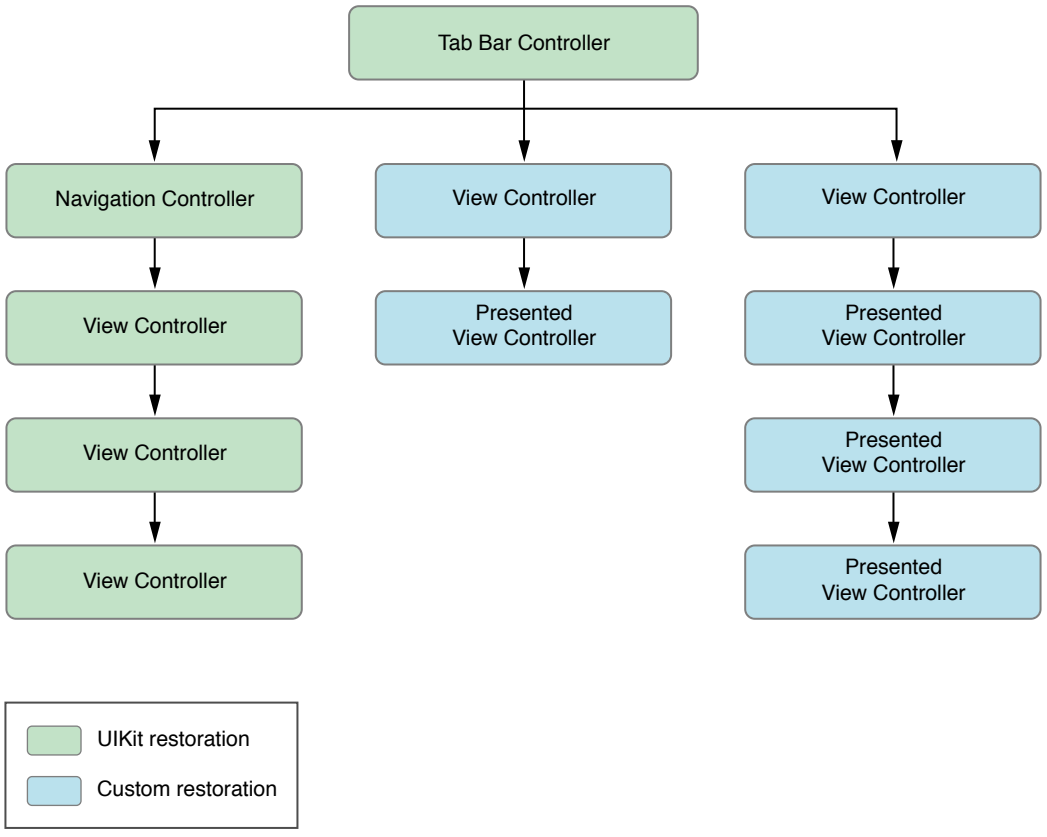
If your app contains state that does not live in a view controller, but that needs to be preserved, you can use the preceding methods to save and restore it. The `application:willEncodeRestorableStateWithCoder:` method is called at the very beginning of the preservation process so that you can write out any high-level app state, such as the current version of your user interface. The `application:didDecodeRestorableStateWithCoder:` method is called at the end of the restoration state so that you can decode any data and perform any final cleanup that your app requires.

Mixing UIKit's State Preservation with Your Own Custom Mechanisms

If your app already implements its own custom state preservation and restoration mechanism, you can continue to use that mechanism and migrate your code to use UIKit's support over time. The design of UIKit's preservation mechanism allows you to pick and choose which view controllers you want to preserve. Thus, you can designate that only portions of your interface should be restored by UIKit, leaving the rest to be handled by your app's current process.

Figure 4-7 shows a sample view hierarchy containing a tab bar controller and the view controllers in its assorted tabs. In this sample, because the tab bar controller has a restoration identifier associated with it, UIKit saves the state of the tab bar controller and all other child view controllers that also have a restoration identifier. Your app's custom code would then need to preserve the state of the remaining view controllers. During restoration, a similar process occurs. UIKit restores all of the view controllers that it preserved while your custom code restores the rest.

Figure 4-7 UIKit handles the root view controller



If you prefer to have your own code manage the root view controller of your app, the save and restore process differs slightly. Because UIKit would not automatically save any view controllers, you need to encode them manually in the `application:willEncodeRestorableStateWithCoder:` method of your app delegate.

When you use the `encodeObject:forKey:` method of the coder to encode a view controller object, the coder uses the view controller's `encodeRestorableStateWithCoder:` method to do the encoding. This process allows you to write arbitrary view controllers to the state preservation archive managed by UIKit.

When you decode archived view controllers during the next launch cycle, you must still be prepared to provide an instance of each view controller to UIKit. When you call the `decodeObjectForKey:` method to decode your view controller, UIKit calls the `application:viewControllerWithRestorationIdentifierPath:coder:` method of your app delegate to retrieve the view controller object first. Only after UIKit has the view controller object does it call the `decodeRestorableStateWithCoder:` method to return the view controller to its previous state. Your code can use the `application:viewControllerWithRestorationIdentifierPath:coder:` method to create the view controller and install it in your app's view controller hierarchy.

Tips for Saving and Restoring State Information

As you add support for state preservation and restoration to your app, consider the following guidelines:

- **Encode version information along with the rest of your app's state.** During the preservation process, it is recommended that you encode a version string or number that identifies the current revision of your app's user interface. You can encode this state in the `application:willEncodeRestorableStateWithCoder:` method of your app delegate. When your app delegate's `application:shouldRestoreApplicationState:` method is called, you can retrieve this information from the provided coder and use it to determine if state preservation is possible.
- **Do not include objects from your data model in your app's state.** Apps should continue to save their data separately in iCloud or to local files on disk. Never use the state restoration mechanism to save that data. Preserved interface data may be deleted if problems occur during a restore operation. Therefore, any preservation-related data you write to disk should be considered purgeable.
- **The state preservation system expects you to use view controllers in the ways they were designed to be used.** The view controller hierarchy is created through a combination of view controller containment and by presenting one view controller from another. If your app displays the view of a view controller by another means—for example, by adding it to another view without creating a containment relationship between the corresponding view controllers—the preservation system will not be able to find your view controller to preserve it.
- **Remember that you might not want to preserve all view controllers.** In some cases, it might not make sense to preserve a view controller. For example, if the user left your app while it was displaying a view controller to change the user's password, you might want to cancel the operation and restore the app to the previous screen. In such a case, you would not preserve the view controller that asks for the new password information.

- **Avoid swapping view controller classes during the restoration process.** The state preservation system encodes the class of the view controllers it preserves. During restoration, if your app returns an object whose class does not match (or is not a subclass of) the original object, the system does not ask the view controller to decode any state information. Thus, swapping out the old view controller for a completely different one does not restore the full state of the object.
- **Be aware that the system automatically deletes an app's preserved state when the user force quits the app.** Deleting the preserved state information when the app is killed is a safety precaution. (The system also deletes preserved state if the app crashes at launch time as a similar safety precaution.) If you want to test your app's ability to restore its state, you should not use the multitasking bar to kill the app during debugging. Instead, use Xcode to kill the app or kill the app programmatically by installing a temporary command or gesture to call `exit` on demand.

App-Related Resources

Aside from the images and media files your app presents on screen, there are some specific resources that iOS itself requires your app to provide. The system uses these resources to determine how to present your app on the user's home screen and, in some cases, how to facilitate interactions with other parts of the system.

App Store Required Resources

There are several things that you are required to provide in your app bundle before submitting it to the App Store:

- Your app must have an `Info.plist` file. This file contains information that the system needs to interact with your app. Xcode creates a version of this file automatically but most apps need to modify this file in some way. For information on how to configure this file, see "The Information Property List File" (page 93).
- Your app's `Info.plist` file must include the `UIRequiredDeviceCapabilities` key. The App Store uses this key to determine whether or not a user can run your app on a specific device. For information on how to configure this key, see "Declaring the Required Device Capabilities" (page 94).
- You must include one or more icons in your app bundle. The system uses these icons when presenting your app on the device's home screen. For information about how to specify app icons, see "App Icons" (page 98).
- Your app must include at least one image to be displayed while your app is launching. The system displays this image to provide the user with immediate feedback that your app is launching. For information about launch images, see "App Launch (Default) Images" (page 100).

The Information Property List File

The information property list (`Info.plist`) file contains critical information about your app's configuration and must be included in your app bundle. Every new project you create in Xcode has a default `Info.plist` file configured with some basic information about your project. You can modify this file to specify additional configuration details for your app.

Your app's `Info.plist` file *must* include the following keys:

- `UIRequiredDeviceCapabilities`—The App Store uses this key to determine the capabilities of your app and to prevent it from being installed on devices that do not support features your app requires. For more information about this key, see "Declaring the Required Device Capabilities" (page 94).
- `CFBundleIcons`—This is the preferred key for specifying your app's icon files. Older projects might include the `CFBundleIconFiles` key instead. Both keys have essentially the same purpose but the `CFBundleIcons` key is preferred because it allows you to organize your icons more efficiently. (The `CFBundleIcons` key is also required for Newsstand apps.)
- `UISupportedInterfaceOrientations`—This key is included by Xcode automatically and is set to an appropriate set of default values. However, you should add or remove values based on the orientations that your app actually supports.

You might also want to include the following keys in your app's `Info.plist` file, depending on the behavior of your app:

- `UIBackgroundModes`—Include this key if your app supports executing in the background using one of the defined modes; see "Implementing Long-Running Background Tasks" (page 58).
- `UIFileSharingEnabled`—Include this key if you want to expose the contents of your sandbox's `Documents` directory in iTunes.
- `UIRequiresPersistentWiFi`—Include this key if your app requires a Wi-Fi connection.
- `UINewsstandApp`—Include this key if your app presents content from the Newsstand app.

The `Info.plist` file itself is a property list file that you can edit manually or using Xcode. Each new Xcode project contains a file called <*project_name*>`-Info.plist`, where <*project_name*> is the name of your Xcode project. This file is the template that Xcode uses to generate the actual `Info.plist` file at build time. When you select this file, Xcode displays the property list editor that you can use to add or remove keys or change the value of a key. For information about how to configure the contents of this file, see *Property List Editor Help*.

For details about the keys you can include in the `Info.plist` file, see *Information Property List Key Reference*.

Declaring the Required Device Capabilities

The `UIRequiredDeviceCapabilities` key lets you declare the hardware or specific capabilities that your app needs in order to run. All apps are required to have this key in their `Info.plist` file. The App Store uses the contents of this key to prevent users from downloading your app onto a device that cannot possibly run it.

The value of the `UIRequiredDeviceCapabilities` key is either an array or a dictionary that contains additional keys identifying features your app requires (or specifically prohibits). If you specify the value of the key using an array, the presence of a key indicates that the feature is required; the absence of a key indicates that the feature is not required and that the app can run without it. If you specify a dictionary instead, each key in the dictionary must have a Boolean value that indicates whether the feature is required or prohibited. A value of `true` indicates the feature is required and a value of `false` indicates that the feature must *not* be present on the device. If a given capability is optional for your app, do not include the corresponding key in the dictionary.

Table 5-1 lists the keys that you can include in the array or dictionary for the `UIRequiredDeviceCapabilities` key. You should include keys only for the features that your app absolutely requires. If your app can run without a specific feature, do not include the corresponding key.

Table 5-1 Dictionary keys for the `UIRequiredDeviceCapabilities` key

Key	Description
`accelerometer`	Include this key if your app requires (or specifically prohibits) the presence of accelerometers on the device. Apps use the Core Motion framework to receive accelerometer events. You do not need to include this key if your app detects only device orientation changes.
`armv6`	Include this key if your app is compiled only for the armv6 instruction set. (iOS 3.1 and later)
`armv7`	Include this key if your app is compiled only for the armv7 instruction set. (iOS 3.1 and later)
`auto-focus-camera`	Include this key if your app requires (or specifically prohibits) autofocus capabilities in the device's still camera. Although most developers should not need to include this key, you might include it if your app supports macro photography or requires sharper images in order to perform some sort of image processing.
`bluetooth-le`	Include this key if your app requires (or specifically prohibits) the presence of Bluetooth low-energy hardware on the device. (iOS 5 and later.)
`camera-flash`	Include this key if your app requires (or specifically prohibits) the presence of a camera flash for taking pictures or shooting video. Apps use the `UIImagePickerController` interface to control the enabling of this feature.
`front-facing-camera`	Include this key if your app requires (or specifically prohibits) the presence of a forward-facing camera. Apps use the `UIImagePickerController` interface to capture video from the device's camera.

Key	Description
`gamekit`	Include this key if your app requires (or specifically prohibits) Game Center. (iOS 4.1 and later)
`gps`	Include this key if your app requires (or specifically prohibits) the presence of GPS (or AGPS) hardware when tracking locations. (You should include this key only if you need the higher accuracy offered by GPS hardware.) If you include this key, you should also include the `location-services` key. You should require GPS only if your app needs location data more accurate than the cellular or Wi-fi radios might otherwise provide.
`gyroscope`	Include this key if your app requires (or specifically prohibits) the presence of a gyroscope on the device. Apps use the Core Motion framework to retrieve information from gyroscope hardware.
`location-services`	Include this key if your app requires (or specifically prohibits) the ability to retrieve the device's current location using the Core Location framework. (This key refers to the general location services feature. If you specifically need GPS-level accuracy, you should also include the `gps` key.)
`magnetometer`	Include this key if your app requires (or specifically prohibits) the presence of magnetometer hardware. Apps use this hardware to receive heading-related events through the Core Location framework.
`microphone`	Include this key if your app uses the built-in microphone or supports accessories that provide a microphone.
`opengles-1`	Include this key if your app requires (or specifically prohibits) the presence of the OpenGL ES 1.1 interfaces.
`opengles-2`	Include this key if your app requires (or specifically prohibits) the presence of the OpenGL ES 2.0 interfaces.
`peer-peer`	Include this key if your app requires (or specifically prohibits) peer-to-peer connectivity over a Bluetooth network. (iOS 3.1 and later)
`sms`	Include this key if your app requires (or specifically prohibits) the presence of the Messages app. You might require this feature if your app opens URLs with the `sms` scheme.
`still-camera`	Include this key if your app requires (or specifically prohibits) the presence of a camera on the device. Apps use the `UIImagePickerController` interface to capture images from the device's still camera.

Key	Description
`telephony`	Include this key if your app requires (or specifically prohibits) the presence of the Phone app. You might require this feature if your app opens URLs with the `tel` scheme.
`video-camera`	Include this key if your app requires (or specifically prohibits) the presence of a camera with video capabilities on the device. Apps use the `UIImagePickerController` interface to capture video from the device's camera.
`wifi`	Include this key if your app requires (or specifically prohibits) access to the networking features of the device.

For detailed information on how to create and edit property lists, see *Information Property List Key Reference*.

Declaring Your App's Supported Document Types

If your app is able to open existing or custom file types, your `Info.plist` file should include information about those types. Declaring file types is how you let the system know that your app is able to open files of the corresponding type. The system uses this information to direct file requests to your app at appropriate times. For example, if the Mail app receives an attachment, the system can direct that attachment to your app to open.

When declaring your app's supported file types, you typically do not configure keys in your `Info.plist` file directly. In the Info tab of your target settings, there is a Document Settings section that you can use to specify your app's supported types. Each document type that you add to this section can represent one file type or several file types. For example, you can define a single document type that represents only PNG images or one that represents PNG, JPG, and GIF images. The decision to represent one file type or multiple file types depends on how your app presents the files. If it presents all of the files in the same way—that is, with the same icon and with the same basic code path—then you can use one document type for multiple file types. If the code paths or icons are different for each file type, you should declare different document types for each.

For each document type, you must provide the following information at a minimum:

- **A name.** This is a localizable string that can be displayed to the user if needed.
- **An icon.** All files associated with a document type share the same icon.
- **The file types.** These are uniform type identifier (UTI) strings that identify the supported file types. For example, to specify the PNG file type, you would specify the `public.png` UTI. UTIs are the preferred way to specify file types because they are less fragile than filename extensions and other techniques used to identify files.

Ultimately, Xcode converts your document type information into a set of keys and adds them to the `CFBundleDocumentTypes` key in your app's `Info.plist` file. The `CFBundleDocumentTypes` key contains an array of dictionaries, where each dictionary represents one of your declared document types and includes the name, icon, file type, and other information you specified.

For more information on the keys you use to declare your app's document types, see *Information Property List Key Reference*. For information about how to open files passed to your app by the system, see "Handling URL Requests" (page 120).

App Icons

Every app must provide an icon to be displayed on a device's Home screen and in the App Store. An app may actually specify several different icons for use in different situations. For example, an app can provide a small icon to use when displaying search results and can provide a high-resolution icon for devices with Retina displays.

Regardless of how many different icons your app has, you specify them using the `CFBundleIcons` key in the `Info.plist` file. The value of that key is an array of strings, each of which contains the filename of one of your icons. The filenames can be anything you want, but all image files must be in the PNG format and must reside in the top level of your app bundle. (Avoid using interlaced PNGs.) When the system needs an icon, it choose the image file whose size most closely matches the intended usage.

Table 5-2 lists the dimensions of the icons you can include with your app. For apps that run on devices with Retina displays, two versions of each icon should be provided, with the second one being a high-resolution version of the original. The names of the two icons should be the same except for the inclusion of the string `@2x` in the filename of the high-resolution image. You can find out more about specifying and loading high-resolution image resources in *Drawing and Printing Guide for iOS*. For a complete list of app-related icons and detailed information about the usage and preparation of your icons, see *iOS Human Interface Guidelines*.

Table 5-2 Sizes for images in the `CFBundleIcons` key

Icon	Idiom	Size	Usage
App icon (required)	iPhone	57 x 57 pixels 114 x 114 pixels (@2x)	This is the main icon for apps running on iPhone and iPod touch.
App icon (required)	iPad	72 x 72 pixels 144 x 144 pixels (@2x)	This is the main icon for apps running on iPad.

Icon	Idiom	Size	Usage
App icon for the App Store (required)	iPhone/iPad	512 x 512 1024 x 1024 (@2x)	iTunes uses this icon when presenting your app for distribution. These files must be included at the top level of your app bundle and the names of the files must be `iTunesArtwork` and `iTunesArtwork@2x` (no filename extension).
Small icon for Spotlight search results and Settings (recommended)	iPhone	29 x 29 pixels 58 x 58 pixels (@2x)	This is the icon displayed in conjunction with search results on iPhone and iPod touch. This icon is also used by the Settings app on all devices.
Small icon for Spotlight search results and Settings (recommended)	iPad	50 x 50 pixels 100 x 100 pixels (@2x)	This is the icon displayed in conjunction with search results on iPad.

When specifying icon files using the `CFBundleIcons` key, it is best to omit the filename extensions of your image files. If you include a filename extension, you must explicitly add the names of all image files (including any high-resolution variants). When you omit the filename extension, the system automatically detects high-resolution variants of your file, even if they are not included in the array.

If your iPhone app is running in iOS 3.1.3 or earlier, the system does not look for icons using your `Info.plist` file. The `CFBundleIcons` key was introduced in iOS 5.0 and the `CFBundleIconFiles` key was introduced in iOS 3.2. Instead of using these keys, the system looks for icon files with specific filenames. Although the sizes of the icons are the same as those in Table 5-2 (page 98), if your app supports deployment on iOS 3.1.3 and earlier, you must use the following filenames when naming your icons:

- `Icon.png`. The name for the app icon on iPhone or iPod touch.
- `Icon-72.png`. The name for the app icon on iPad.
- `Icon-Small.png`. The name for the search results icon on iPhone and iPod touch. This file is also used for the Settings icon on all devices.
- `Icon-Small-50.png`. The name of the search results icon on iPad.

Important: The use of fixed filenames for your app icons is for compatibility with earlier versions of iOS only. Even if you use these fixed icon filenames, your app should continue to include the `CFBundleIcons` or `CFBundleIconFiles` key in your app's `Info.plist` file.

For more information about the `CFBundleIcons` key, see *Information Property List Key Reference*. For information about creating your app icons, see *iOS Human Interface Guidelines*.

App Launch (Default) Images

When the system launches an app, it temporarily displays a static launch image on the screen. Your app provides this image, with the image contents usually containing a prerendered version of your app's default user interface. The purpose of this image is to give the user immediate feedback that the app launched. It also gives your app time to initialize itself and prepare its initial set of views for display. When your app is ready to run, the system removes the image and displays your app's windows and views.

Every app must provide at least one launch image. This image is typically in a file named `Default.png` that displays your app's initial screen in a portrait orientation. However, you can also provide other launch images to be used under different launch conditions. All launch images must be PNG files and must reside in the top level of your app's bundle directory. (Avoid using interlaced PNGs.) The name of each launch image indicates its purpose and how it is used. The format for launch image filenames is as follows:

<basename> *<usage_specific_modifiers>* *<scale_modifier>* *<device_modifier>* `.png`

The *<basename>* portion of the filename is either the string `Default` or a custom string that you specify using the `UILaunchImageFile` key in your app's `Info.plist` file. The *<scale_modifier>* portion is the optional string `@2x` and should be included only for images intended for use on Retina displays. Other optional modifiers may also be included in the name, and several standard modifiers are discussed in the sections that follow.

Table 5-3 lists the dimensions for launch images in iOS apps. For all dimensions, the image width is listed first, followed by the image height. For precise information about which size launch image to use and how to prepare your launch images, see *iOS Human Interface Guidelines*.

Table 5-3 Typical launch image dimensions

Device	Portrait	Landscape
iPhone and iPod touch	320 x 480 pixels 640 x 960 pixels (`@2x`)	Not supported
iPhone 5 and iPod touch (5th generation)	640 x 1136 pixels (`@2x`)	Not supported

Device	Portrait	Landscape
iPad	768 x 1004 pixels 1536 x 2008 pixels (@2x)	1024 x 748 pixels 2048 x 1496 pixels (@2x)

To demonstrate the naming conventions, suppose your iOS app's `Info.plist` file included the `UILaunchImageFile` key with the value `MyLaunchImage`. The standard resolution version of the launch image would be named `MyLaunchImage.png` and would be in a portrait orientation (320 x 480). The high-resolution version of the same launch image would be named `MyLaunchImage@2x.png`. If you did not specify a custom launch image name, these files would need to be named `Default.png` and `Default@2x.png`, respectively.

To specify default launch images for iPhone 5 and iPod touch (5th generation) devices, include the modifier string `–568h` immediately after the <*basename*> portion of the filename. Because these devices have Retina displays, the `@2x` modifier must always be included with launch images for the devices. For example, the default launch image name for a device is `Default–568h@2x.png`. (If your app has the `UILaunchImageFile` key in its `Info.plist` file, replace the `Default` portion of the string with your custom string.) The `–568h` modifier should always be the first one in the list. You can also insert other modifiers after the `–568h` string as described below.

For more information about the `UILaunchImageFile` key, see *Information Property List Key Reference*.

Providing Launch Images for Different Orientations

In iOS 3.2 and later, an iPad app can provide both landscape and portrait versions of its launch images. Each orientation-specific launch image must include a special modifier string in its filename. The format for orientation-specific launch image filenames is as follows:

<*basename*> <*orientation_modifier*> <*scale_modifier*> <*device_modifier*> `.png`

Table 5-4 lists the possible modifiers you can specify for the <*orientation_modifier*> value in your image filenames. As with all launch images, each file must be in the PNG format. These modifiers are supported for launch images used in iPad apps only; they are not supported for apps running on iPhone or iPod touch devices.

Table 5-4 Launch image orientation modifiers

Modifier	Description
`–PortraitUpsideDown`	Specifies an upside-down portrait version of the launch image. A file with this modifier takes precedence over a file with the `–Portrait` modifier for this specific orientation.

Modifier	Description
`–LandscapeLeft`	Specifies a left-oriented landscape version of the launch image. A file with this modifier takes precedence over a file with the `–Landscape` modifier for this specific orientation.
`–LandscapeRight`	Specifies a right-oriented landscape version of the launch image. A file with this modifier takes precedence over a file with the `–Landscape` modifier for this specific orientation.
`–Portrait`	Specifies the generic portrait version of the launch image. This image is used for right-side up portrait orientations and takes precedence over the `Default.png` image file (or your custom-named replacement for that file). If a file with the `–PortraitUpsideDown` modifier is not specified, this file is also used for upside-down portrait orientations as well.
`–Landscape`	Specifies the generic landscape version of the launch image. If a file with the `–LandscapeLeft` or `–LandscapeRight` modifier is not specified, this image is used instead. This image takes precedence over the `Default.png` image file (or your custom-named replacement for that file).
(none)	If you provide a launch image file with no orientation modifier, that file is used when no other orientation-specific launch image is available. For apps running on systems earlier than iOS 3.2, you must name this file `Default.png`.

For example, if you specify the value `MyLaunchImage` in the `UILaunchImageFile` key, the custom landscape and portrait launch images for your iPad app would be named `MyLaunchImage–Landscape.png` and `MyLaunchImage–Portrait.png`. If you do not specify a custom launch image filename, you would use the names `Default–Landscape.png` and `Default–Portrait.png`.

No matter which launch image is displayed by the system, your app always launches in a portrait orientation initially and then rotates as needed to the correct orientation. Therefore, your `application:didFinishLaunchingWithOptions:` method should always assume a portrait orientation when setting up your window and views. Shortly after the `application:didFinishLaunchingWithOptions:` method returns, the system sends any necessary orientation-change notifications to your app's window, giving it and your app's view controllers a chance to reorient views using the standard process.

For more information about how your view controllers manage the rotation process, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Providing Device-Specific Launch Images

Universal apps must provide launch images for both the iPhone and iPad idioms. Because iPhone apps require only one launch image (`Default.png`), whereas iPad apps typically require different images for portrait and landscape orientations, you can usually do without device-specific modifiers. However, if you create multiple launch images for each idiom, the names of device-specific image files are likely to collide. In that situation, you can append a device modifier to filenames to indicate that they are for a specific platform only. The following device modifiers are recognized for launch images in iOS 4.0 and later:

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

Because device modifiers are not supported in iOS 3.2, the minimal set of launch images needed for a universal app (running in iOS 3.2 and later) would need to be named `Default.png` and `Default~iphone.png`. In that case, the `Default.png` file would contain the iPad launch image (for all orientations) and the `Default~iphone.png` file would contain the iPhone version of the image. (To support high-resolution displays, you would also need to include a `Default@2x~iphone.png` launch image.)

> **Note:** If you are using the `UILaunchImageFile` key in your `Info.plist` file to specify a custom base name for your launch image files, add device-specific versions as needed to differentiate the launch images on different devices. For example, specify a `UILaunchImageFile~ipad` key to specify a different base name for iPad launch images. Specifying different base names lets a universal app avoid naming conflicts among its launch images. For more information on how to apply device modifiers to keys in the `Info.plist` file, see *Information Property List Key Reference*.

Providing Launch Images for Custom URL Schemes

If your app supports one or more custom URL schemes, it can also provide a custom launch image for each URL scheme. When the system launches your app to handle a URL, it displays the launch image associated with the scheme of the given URL. In this case, the format for your launch image filenames are as follows:

<basename>`-`*<url_scheme>* *<scale_modifier>* *<device_modifier>* `.png`

The *<url_scheme>* modifier is a string representing the name of your URL scheme name. For example, if your app supports a URL scheme with the name `myscheme`, the system looks for an image with the name `Default-myscheme.png` (or `Default-myscheme@2x.png` for Retina displays) in the app's bundle. If the app's `Info.plist` file includes the `UILaunchImageFile` key, the base name portion changes from `Default` to the custom string you provide in that key.

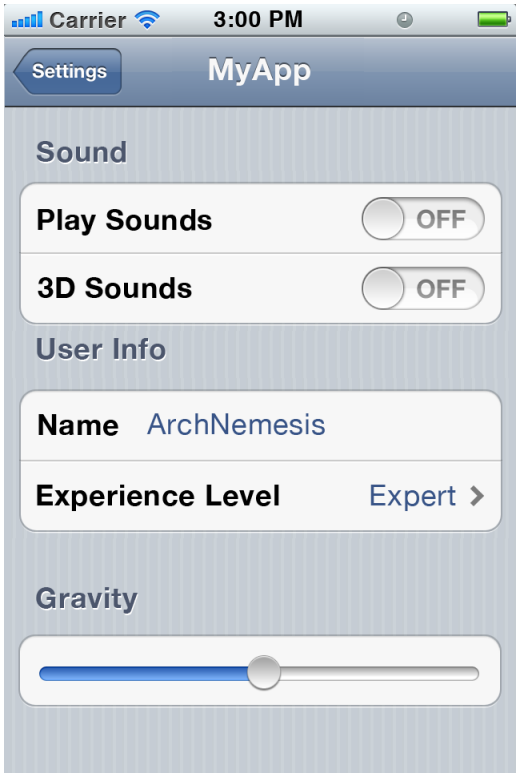
Note: You can combine a URL scheme modifier with orientation modifiers. If you do this, the format for the filename is *<basename>*–*<url_scheme>* *<orientation_modifier>* *<scale_modifier>* *<device_modifier>*.png.For more information about the launch orientation modifiers, see "Providing Launch Images for Different Orientations" (page 101).

In addition to including the launch images at the top level of your bundle, you can also include localized versions of your launch images in your app's language-specific project subdirectories. For more information on localizing resources in your app, see "Localized Resource Files" (page 105).

The Settings Bundle

Apps that want to display preferences in the Settings app must include a Settings bundle resource. A **Settings bundle** is a specially formatted bundle that sits at the top of your app's bundle directory and contains the data needed to display your app's preferences. Figure 5-1 shows an example of custom preferences displayed for an app.

Figure 5-1 Custom preferences displayed by the Settings app



> **Note:** Because changing preferences in the Settings app requires leaving your app, you should use a Settings bundle only for preferences that the user changes infrequently. Frequently changed settings should be included directly inside your app.

Xcode provides support for creating a Settings bundle resource and adding it to your app. Inside the Settings bundle, you place one or more property list files and any images associated with your preferences. Each property-list file contains special keys and values that tell the Settings app how to display different pages of your preferences. Changes to your app's preferences are stored in the user defaults database and are accessible to your app using an `NSUserDefaults` object.

For detailed information about how to create a Settings bundle, see *Preferences and Settings Programming Guide*.

Localized Resource Files

Because iOS apps are distributed in many countries, localizing your app's content can help you reach many more customers. Users are much more likely to use an app when it is localized for their native language. When you factor your user-facing content into resource files, localizing that content is a relatively simple process.

Before you can localize your content, you must internationalize your app in order to facilitate the localization process. Internationalizing your app involves factoring out any user-facing content into localizable resource files and providing language-specific project (`.lproj`) directories for storing that content. It also means using appropriate technologies (such as date and number formatters) when working with language-specific and locale-specific content.

For a fully internationalized app, the localization process creates new sets of language-specific resource files for you to add to your project. A typical iOS app requires localized versions of the following types of resource files:

- Storyboard files (or nib files)—Storyboards can contain text labels and other content that need to be localized. You might also want to adjust the position of interface items to accommodate changes in text length. (Similarly, nib files can contain text that needs to be localized or layout that needs to be updated.)
- Strings files—Strings files (so named because of their `.strings` filename extension) contain localized text that you plan to display in your app.
- Image files—You should avoid localizing images unless the images contain culture-specific content. And you should never store text directly in your image files. Instead, store text in a strings file and composite that text with your image-based content at runtime..

- Video and audio files—You should avoid localizing multimedia files unless they contain language-specific or culture-specific content. For example, you would want to localize a video file that contained a voice-over track.

For information about the internationalization and localization process, see *Internationalization Programming Topics*. For information about the proper way to use resource files in your app, see *Resource Programming Guide*.

Loading Resources Into Your App

Putting resources into your bundle is the first step but at runtime, you need to be able to load those resources into memory and use them. Resource management is broken down basically into two steps:

1. Locate the resource.
2. Load the resource.
3. Use the resource.

To locate resources, you use an `NSBundle` object. A bundle object understands the structure of your app's bundle and knows how to locate resources inside it. Bundle objects even use the current language settings to choose an appropriately localized version of the resource. The `pathForResource:ofType:` method is one of several `NSBundle` methods that you can use to retrieve the location of resource files.

Once you have the location of a resource file, you have to decide the most appropriate way to load it into memory. Common resource types usually have a corresponding class that you use to load the resource:

- To load view controllers (and their corresponding views) from a storyboard, use the `UIStoryboard` class.
- To load an image, use the methods of the `UIImage` class.
- To load string resources, use the `NSLocalizedString` and related macros defined in Foundation framework.
- To load the contents of a property list, use the `dictionaryWithContentsOfURL:` method of `NSDictionary`, or use the `NSPropertyListSerialization` class.
- To load binary data files, use the methods of the `NSData` class.
- To load audio and video resources, use the classes of the Assets Library, Media Player, or AV Foundation frameworks.

The following example shows how to load an image stored in a resource file in the app's bundle. The first line gets the location of the file in the app's bundle (also known as the main bundle here). The second line creates a `UIImage` object using the data in the file at that location.

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

For more information about resources and how to access them from your app, see *Resource Programming Guide*.

Advanced App Tricks

Many app-related tasks depend on the type of app you are trying to create. This chapter shows you how to implement some of the common behaviors found in iOS apps.

Creating a Universal App

A universal app is a single app that is optimized for iPhone, iPod touch, and iPad devices. Providing a single binary that adapts to the current device offers the best user experience but, of course, involves extra work on your part. Because of the differences in device screen sizes, most of your window, view, and view controller code for iPad is likely to be very different from the code for iPhone and iPod touch. In addition, there are things you must do to ensure your app runs correctly on each device type.

Xcode provides built-in support for configuring universal apps. When you create a new project, you can select whether you want to create a device-specific project or a universal project. After you create your project, you can change the supported set of devices for your app target using the Summary pane. When changing from a single-device project to a universal project, you must fill in the information for the device type for which you are adding support.

The following sections highlight the changes you must make to an existing app to ensure that it runs smoothly on any type of device.

Updating Your Info.plist Settings

Most of the existing keys in a universal app's `Info.plist` file should remain the same. However, for any keys that require different values on iPhone versus iPad devices, you can add device modifiers to the key name. When reading the keys of your `Info.plist` file, the system interprets each key using the following format:

key_root`~`<*platform*>`~`<*device*>

In this format, the *key_root* portion represents the original name of the key. The <*platform*> and <*device*> portions are both optional endings that you can use for keys that are specific to a platform or device. For apps that run only on iOS, you can omit the platform string. (The `iphoneos` platform string is used to distinguish apps written for iOS from those written for Mac OS X.) To apply a key to a specific device, use one of the following values:

- `iphone`—The key applies to iPhone devices.

- `ipod`—The key applies to iPod touch devices.
- `ipad`—The key applies to iPad devices.

For example, to indicate that you want your app to launch in a portrait orientation on iPhone and iPod touch devices but in landscape-right on iPad, you would configure your `Info.plist` with the following keys:

```
<key>UIInterfaceOrientation</key>
<string>UIInterfaceOrientationPortrait</string>
<key>UIInterfaceOrientation~ipad</key>
<string>UIInterfaceOrientationLandscapeRight</string>
```

Notice that in the preceding example, there is an iPad-specific key and a default key without any device modifiers. Continue to use the default key to specify the most common (or default) value and add a specific version with a device-specific modifier when you need to change that value. This guarantees that there is always a value available for the system to examine. For example, if you were to replace the default key with an iPhone-specific and iPad-specific version of the `UIInterfaceOrientation` key, the system would not know the preferred starting orientation for iPod devices.

For more information about the keys you can include in your `Info.plist` file, see *Information Property List Key Reference*

Implementing Your View Controllers and Views

The largest amount of effort that goes into creating universal apps is designing your user interface. Because of the different screen sizes, apps often need completely separate versions of their interface for each device idiom. This means creating new view hierarchies but might also mean creating completely different view controller objects to manage those views.

For views, the main modification is to redesign your view hierarchies to support the larger screen. Simply scaling existing views may work but often yields poor results. Your new interface should make use of the available space and take advantage of new interface elements where appropriate. Doing so is better because it results in an interface that feels more natural to the user, and does not just feel like an iPhone app on a larger screen.

For view controllers, follow these guidelines:

- Consider defining separate view controller classes for iPhone and iPad devices. Using separate view controllers is often easier than trying to create one view controller that supports both platforms. If there is a significant amount of shared code, you could always put the shared code in a base class and then implement custom subclasses to address device-specific issues.

- If you use a single view controller class for both platforms, your code must support both iPhone and iPad screen sizes. (For an app that uses nib files, this might mean choosing which nib file to load based on the current device idiom.) Similarly, your view controller code must be able to handle differences between the two platforms.

For views, follow these guidelines:

- Consider using separate sets of views for iPhone and iPad devices. For custom views, this means defining different versions of your class for each device.
- If you choose to use the same custom view for both devices, make sure your `drawRect:` and `layoutSubviews` methods especially work properly on both devices.

For information about the view controllers you can use in your apps, see *View Controller Programming Guide for iOS*.

Updating Your Resource Files

Because resource files are often used to implement portions of your app's user interface, you need to make the following changes:

- In addition to the `Default.png` file displayed when your app launches on iPhone devices, you must add new launch images for iPad devices as described in "Providing Launch Images for Different Orientations" (page 101).
- If you use images, you may need to add larger (or higher-resolution) versions to support iPad devices.
- If you use storyboard or nib files, you need to provide a new set of files for iPad devices.
- You must size your app icons appropriately for iPad, as described in "App Icons" (page 98).

When using different resource files for each platform, you can conditionally load those resources just as you would conditionally execute code. For more information about how to use runtime checks, see "Using Runtime Checks to Create Conditional Code Paths" (page 110).

Using Runtime Checks to Create Conditional Code Paths

If your code needs to follow a different path depending on the underlying device type, use the `userInterfaceIdiom` property of `UIDevice` to determine which path to take. This property provides an indication of the style of interface to create: iPad or iPhone. Because this property is available only in iOS 3.2 and later, apps that support earlier versions of iOS need to check for the availability of this property before accessing it. Of course, the simplest way to check this property is to use the `UI_USER_INTERFACE_IDIOM` macro, which performs the necessary runtime checks for you.

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
     // The device is an iPad running iOS 3.2 or later.
}
else {
     // The device is an iPhone or iPod touch.
}
```

Supporting Multiple Versions of iOS

Any app that supports a range of iOS versions must use runtime checks to prevent the use of newer APIs on older versions of iOS that do not support them. For example, if you build your app using new features in iOS 6 but your app still supports iOS 5, runtime checks allow you to use recently introduced features when they are available and to follow alternate code paths when they are not. Failure to include such checks will cause your app to crash when it tries to use new symbols that are not available on the older operating system.

There are several types of checks that you can make:

- To determine whether a method is available on an existing class, use the `instancesRespondToSelector:` class method or the `respondsToSelector:` instance method.
- Apps that link against iOS SDK 4.2 and later can use the weak linking support introduced in that version of the SDK. This support lets you check for the existence of a given `Class` object to determine whether you can use that class. For example:

  ```
  if ([UIPrintInteractionController class]) {
     // Create an instance of the class and use it.
  }
  else {
     // The print interaction controller is not available.
  }
  ```

  To use this feature, you must build your app using LLVM and Clang and the app's deployment target must be set to iOS 3.1 or later.
- Apps that link against iOS SDK 4.1 and earlier must use the `NSClassFromString` function to see whether a class is defined. If the function returns a value other than `nil`, you may use the class. For example:

  ```
  Class splitVCClass = NSClassFromString(@"UISplitViewController");
  ```

```
if (splitVCClass)
{
   UISplitViewController* mySplitViewController = [[splitVCClass alloc]
init];
   // Configure the split view controller.
}
```

- To determine whether a C-based function is available, perform a Boolean comparison of the function name to `NULL`. If the symbol is not `NULL`, you can use the function. For example:

```
if (UIGraphicsBeginPDFPage != NULL)
{
    UIGraphicsBeginPDFPage();
}
```

For more information and examples of how to write code that supports multiple deployment targets, see *SDK Compatibility Guide*.

Launching in Landscape Mode

Apps that uses only landscape orientations for their interface must explicitly ask the system to launch the app in that orientation. Normally, apps launch in portrait mode and rotate their interface to match the device orientation as needed. For apps that support both portrait and landscape orientations, always configure your views for portrait mode and then let your view controllers handle any rotations. If, however, your app supports landscape but not portrait orientations, perform the following tasks to make it launch in landscape mode initially:

- Add the `UIInterfaceOrientation` key to your app's `Info.plist` file and set the value of this key to either `UIInterfaceOrientationLandscapeLeft` or `UIInterfaceOrientationLandscapeRight`.
- Lay out your views in landscape mode and make sure that their layout or autosizing options are set correctly.
- Override your view controller's `shouldAutorotateToInterfaceOrientation:` method and return `YES` for the left or right landscape orientations and `NO` for portrait orientations.

Important: Apps should always use view controllers to manage their window-based content.

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

Note: To launch a view controller–based app in landscape mode in versions of iOS before 2.1, you need to apply a 90-degree rotation to the transform of the app's root view in addition to all the preceding steps.

Installing App-Specific Data Files at First Launch

You can use your app's first launch cycle to set up any data or configuration files required to run. App-specific data files should be created in the `Library/Application Support/<bundleID>/` directory of your app sandbox, where *<bundleID>* is your app's bundle identifier. You can further subdivide this directory to organize your data files as needed. You can also create files in other directories, such as to your app's iCloud container directory or to the local `Documents` directory, depending on your needs.

If your app's bundle contains data files that you plan to modify, you must copy those files out of the app bundle and modify the copies. You must not modify any files inside your app bundle. Because iOS apps are code signed, modifying files inside your app bundle invalidates your app's signature and prevents your app from launching in the future. Copying those files to the `Application Support` directory (or another writable directory in your sandbox) and modifying them there is the only way to use such files safely.

For more information about the directories of the iOS app sandbox and the proper location for files, see *File System Programming Guide*.

Protecting Data Using On-Disk Encryption

In iOS 4 and later, apps can use the data protection feature to add a level of security to their on-disk data. Data protection uses the built-in encryption hardware present on specific devices (such as the iPhone 3GS and iPhone 4) to store files in an encrypted format on disk. While the user's device is locked, protected files are inaccessible even to the app that created them. The user must explicitly unlock the device (by entering the appropriate passcode) at least once before your app can access one of its protected files.

Data protection is available on most iOS devices and is subject to the following requirements:

- The file system on the user's device must support data protection. This is true for newer devices, but for some earlier devices, the user might have to reformat the device's disk and restore any content from a backup.
- The user must have an active passcode lock set for the device.

To protect a file, your app must add an attribute to the file indicating the desired level of protection. Add this attribute using either the `NSData` class or the `NSFileManager` class. When writing new files, you can use the `writeToFile:options:error:` method of `NSData` with the appropriate protection value as one of the write options. For existing files, you can use the `setAttributes:ofItemAtPath:error:` method of `NSFileManager` to set or change the value of the `NSFileProtectionKey`. When using these methods, your app can specify one of the following protection levels for the file:

- No protection—The file is not encrypted on disk. You can use this option to remove data protection from an accessible file. Specify the `NSDataWritingFileProtectionNone` option (`NSData`) or the `NSFileProtectionNone` attribute (`NSFileManager`).
- Complete—The file is encrypted and inaccessible while the device is locked. Specify the `NSDataWritingFileProtectionComplete` option (`NSData`) or the `NSFileProtectionComplete` attribute (`NSFileManager`).
- Complete unless already open—The file is encrypted. A closed file is inaccessible while the device is locked. After the user unlocks the device, your app can open the file and use it. If the user locks the device while the file is open, though, your app can continue to access it. Specify the `NSDataWritingFileProtectionCompleteUnlessOpen` option (`NSData`) or the `NSFileProtectionCompleteUnlessOpen` attribute (`NSFileManager`).
- Complete until first login—The file is encrypted and inaccessible until after the device has booted and the user has unlocked it once. Specify the `NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication` option (`NSData`) or the `NSFileProtectionCompleteUntilFirstUserAuthentication` attribute (`NSFileManager`).

If you protect a file, your app must be prepared to lose access to that file. When complete file protection is enabled, even your app loses the ability to read and write the file's contents when the user locks the device. Your app has several options for tracking when access to protected files might change, though:

- The app delegate can implement the `applicationProtectedDataWillBecomeUnavailable:` and `applicationProtectedDataDidBecomeAvailable:` methods.
- Any object can register for the `UIApplicationProtectedDataWillBecomeUnavailable` and `UIApplicationProtectedDataDidBecomeAvailable` notifications.
- Any object can check the value of the `protectedDataAvailable` property of the shared `UIApplication` object to determine whether files are currently accessible.

For new files, it is recommended that you enable data protection before writing any data to them. If you are using the `writeToFile:options:error:` method to write the contents of an `NSData` object to disk, this happens automatically. For existing files, adding data protection replaces an unprotected file with a new protected version.

Tips for Developing a VoIP App

A **Voice over Internet Protocol (VoIP)** app allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an app needs to maintain a persistent network connection to its associated service so that it can receive incoming calls and other relevant data. Rather than keep VoIP apps awake all the time, the system allows them to be suspended and provides facilities for monitoring their sockets for them. When incoming traffic is detected, the system wakes up the VoIP app and returns control of its sockets to it.

There are several requirements for implementing a VoIP app:

1. Add the `UIBackgroundModes` key to your app's `Info.plist` file. Set the value of this key to an array that includes the `voip` string.
2. Configure one of the app's sockets for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to install a handler to be executed periodically. Your app can use this handler to maintain its service connection.
4. Configure your audio session to handle transitions to and from active use.
5. To ensure a better user experience on iPhone, use the Core Telephony framework to adjust your behavior in relation to cell-based phone calls; see *Core Telephony Framework Reference*.
6. To ensure good performance for your VoIP app, use the System Configuration framework to detect network changes and allow your app to sleep as much as possible.

Including the `voip` value in the `UIBackgroundModes` key lets the system know that it should allow the app to run in the background as needed to manage its network sockets. This key also permits your app to play background audio (although including the `audio` value for the `UIBackgroundModes` key is still encouraged). An app with this key is also relaunched in the background immediately after system boot to ensure that the VoIP services are always available. For more information about the `UIBackgroundModes` key, see *Information Property List Key Reference*.

Configuring Sockets for VoIP Usage

In order for your app to maintain a persistent connection while it is in the background, you must tag your app's main communication socket specifically for VoIP usage. Tagging this socket tells the system that it should take over management of the socket when your app is suspended. The handoff itself is totally transparent to your app. And when new data arrives on the socket, the system wakes up the app and returns control of the socket so that the app can process the incoming data.

You need to tag only the socket you use for communicating with your VoIP service. This is the socket you use to receive incoming calls or other data relevant to maintaining your VoIP service connection. Upon receipt of incoming data, the handler for this socket needs to decide what to do. For an incoming call, you likely want to post a local notification to alert the user to the call. For other noncritical data, though, you might just process the data quietly and allow the system to put your app back into the suspended state.

In iOS, most sockets are managed using streams or other high-level constructs. To configure a socket for VoIP usage, the only thing you have to do beyond the normal configuration is add a special key that tags the interface as being associated with a VoIP service. Table 6-1 lists the stream interfaces and the configuration for each.

Table 6-1 Configuring stream interfaces for VoIP usage

Interface	Configuration
`NSInputStream` and `NSOutputStream`	For Cocoa streams, use the `setProperty:forKey:` method to add the `NSStreamNetworkServiceType` property to the stream. The value of this property should be set to `NSStreamNetworkServiceTypeVoIP`.
`NSURLRequest`	When using the URL loading system, use the `setNetworkServiceType:` method of your `NSMutableURLRequest` object to set the network service type of the request. The service type should be set to `NSURLNetworkServiceTypeVoIP`.
`CFReadStreamRef` and `CFWriteStreamRef`	For Core Foundation streams, use the `CFReadStreamSetProperty` or `CFWriteStreamSetProperty` function to add the `kCFStreamNetworkServiceType` property to the stream. The value for this property should be set to `kCFStreamNetworkServiceTypeVoIP`.

> **Note:** When configuring your sockets, you need to configure only your main signaling channel with the appropriate service type key. You do not need to include this key when configuring your voice channels.

Because VoIP apps need to stay running in order to receive incoming calls, the system automatically relaunches the app if it exits with a nonzero exit code. (This type of exit could happen when there is memory pressure and your app is terminated as a result.) However, terminating the app also releases all of its sockets, including the one used to maintain the VoIP service connection. Therefore, when the app is launched, it always needs to create its sockets from scratch.

For more information about configuring Cocoa stream objects, see *Stream Programming Guide* . For information about using URL requests, see *URL Loading System Programming Guide* . And for information about configuring streams using the CFNetwork interfaces, see *CFNetwork Programming Guide* .

Installing a Keep-Alive Handler

To prevent the loss of its connection, a VoIP app typically needs to wake up periodically and check in with its server. To facilitate this behavior, iOS lets you install a special handler using the `setKeepAliveTimeout:handler:` method of `UIApplication`. You typically install this handler in the `applicationDidEnterBackground:` method of your app delegate. Once installed, the system calls your handler at least once before the timeout interval expires, waking up your app as needed to do so.

Your keep-alive handler executes in the background and should return as quickly as possible. Handlers are given a maximum of 10 seconds to perform any needed tasks and return. If a handler has not returned after 10 seconds, or has not requested extra execution time before that interval expires, the system suspends the app.

When installing your handler, specify the largest timeout value that is practical for your app's needs. The minimum allowable interval for running your handler is 600 seconds, and attempting to install a handler with a smaller timeout value will fail. Although the system promises to call your handler block before the timeout value expires, it does not guarantee the exact call time. To improve battery life, the system typically groups the execution of your handler with other periodic system tasks, thereby processing all tasks in one quick burst. As a result, your handler code must be prepared to run earlier than the actual timeout period you specified.

Configuring Your App's Audio Session

As with any background audio app, the audio session for a VoIP app must be configured properly to ensure the app works smoothly with other audio-based apps. Because audio playback and recording for a VoIP app are not used all the time, it is especially important that you create and configure your app's audio session

object only when it is needed. For example, you would create the audio session to notify the user of an incoming call or while the user was actually on a call. As soon as the call ends, you would then remove strong references to the audio session and give other audio apps the opportunity to play their audio.

For information about how to configure and manage an audio session for a VoIP app, see *Audio Session Programming Guide*.

Using the Reachability Interfaces to Improve the User Experience

Because VoIP apps rely heavily on the network, they should use the reachability interfaces of the System Configuration framework to track network availability and adjust their behavior accordingly. The reachability interfaces allow an app to be notified whenever network conditions change. For example, a VoIP app could close its network connections when the network becomes unavailable and recreate them when it becomes available again. The app could also use those kinds of changes to keep the user apprised about the state of the VoIP connection.

To use the reachability interfaces, you must register a callback function with the framework and use it to track changes. To register a callback function:

1. Create a `SCNetworkReachabilityRef` structure for your target remote host.
2. Assign a callback function to your structure (using the `SCNetworkReachabilitySetCallback` function) that processes changes in your target's reachability status.
3. Add that target to an active run loop of your app (such as the main run loop) using the `SCNetworkReachabilityScheduleWithRunLoop` function.

Adjusting your app's behavior based on the availability of the network can also help improve the battery life of the underlying device. Letting the system track the network changes means that your app can let itself go to sleep more often.

For more information about the reachability interfaces, see *System Configuration Framework Reference*.

Communicating with Other Apps

Apps that support custom URL schemes can use those schemes to receive messages. Some apps use URL schemes to initiate specific requests. For example, an app that wants to show an address in the Maps app can use a URL to launch that app and display the address. You can implement your own URL schemes to facilitate similar types of communications in your apps.

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 apps. The handlers for these schemes are fixed and cannot be changed. If your URL type includes a scheme that is identical to one defined by Apple, the Apple-provided app is launched instead of your app.

> **Note:** If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme.

To communicate with an app using a custom URL, create an `NSURL` object with some properly formatted content and pass that object to the `openURL:` method of the shared `UIApplication` object. The `openURL:` method launches the app that registered to receive URLs of that type and passes it the URL. At that point, control passes to the new app.

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

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

If your app defines a custom URL scheme, it should implement a handler for that scheme as described in "Implementing Custom URL Schemes" (page 119). 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

If your app can receive specially formatted URLs, you should register the corresponding URL schemes with the system. A custom URL scheme is a mechanism through which third-party apps can communicate with each other. Apps often use custom URL schemes to vend services to other apps. For example, the Maps app supports URLs for displaying specific map locations.

Registering Custom URL Schemes

To register a URL type for your app, include the `CFBundleURLTypes` key in your app's `Info.plist` file. The `CFBundleURLTypes` key contains an array of dictionaries, each of which defines a URL scheme the app supports. Table 6-2 describes the keys and values to include in each dictionary.

Table 6-2 Keys and values of the `CFBundleURLTypes` property

Key	Value
`CFBundleURLName`	A string containing the abstract name of the URL scheme. To ensure uniqueness, it is recommended that you specify a reverse-DNS style of identifier, for example, `com.acme.myscheme`. The string you specify is also used as a key in your app's `InfoPlist.strings` file. The value of the key is the human-readable scheme name.
`CFBundleURLSchemes`	An array of strings containing the URL scheme names—for example, `http`, `mailto`, `tel`, and `sms`.

Figure 6-1 shows the `Info.plist` file of an app that supports a custom scheme for creating "to-do" items. The URL types entry corresponds to the `CFBundleURLTypes` key added to the `Info.plist` file. Similarly, the "URL identifier" and "URL Schemes" entries correspond to the `CFBundleURLName` and `CFBundleURLSchemes` keys.

Figure 6-1 Defining a custom URL scheme in the `Info.plist` file

Key	Type	Value
▼ Information Property List	Dictionary	(14 items)
Localization native development reg	String	en
Executable file	String	${EXECUTABLE_NAME}
Icon file	String	
Bundle identifier	String	window.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Minimum system version	String	${MACOSX_DEPLOYMENT_TARGET}
Copyright (human-readable)	String	Copyright © 2012 Brooke Matteson. All rights reserved.
Main nib file base name	String	MainMenu
Principal class	String	NSApplication

Handling URL Requests

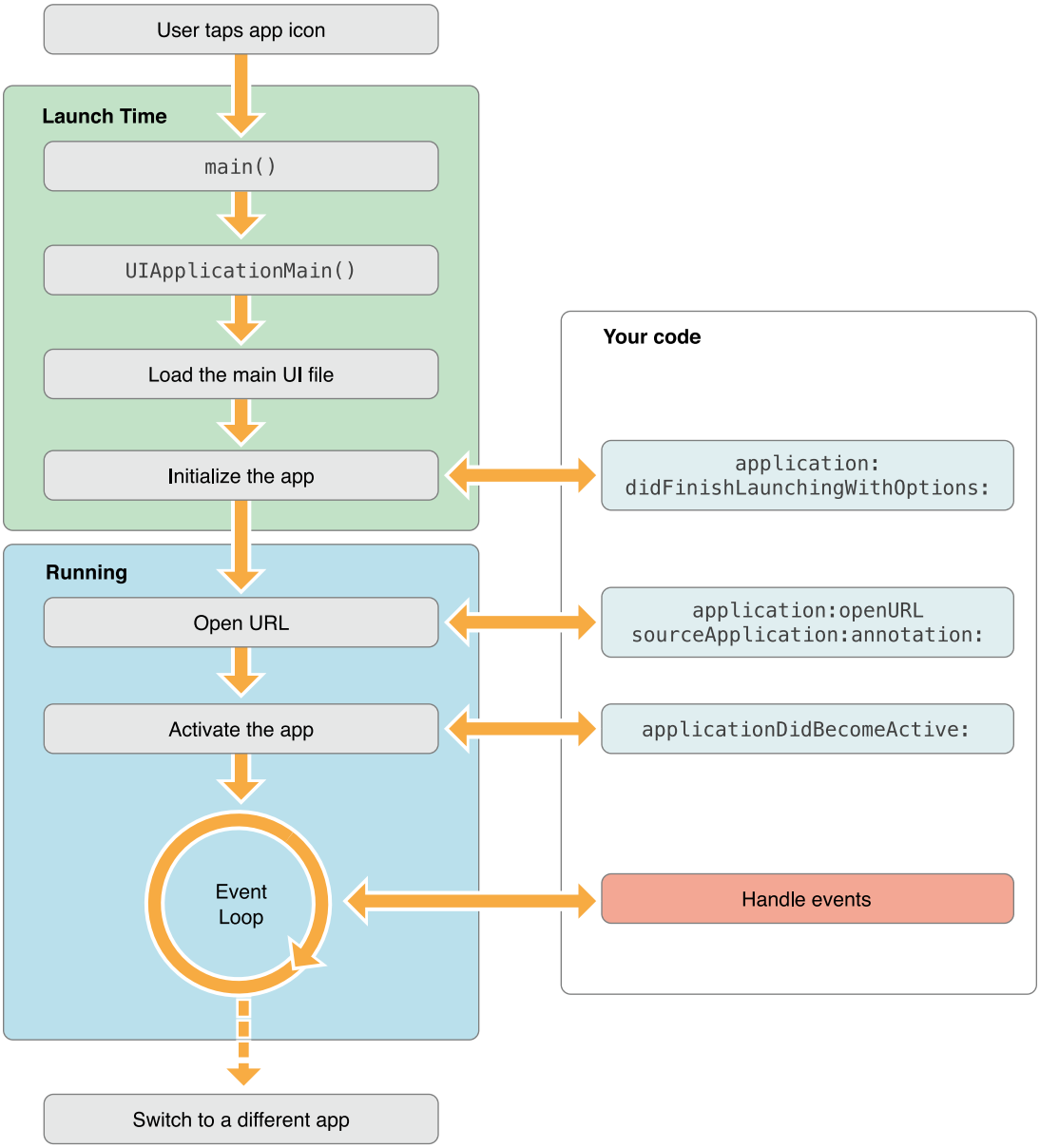
An app that has its own custom URL scheme must be able to handle URLs passed to it. All URLs are passed to your app delegate, either at launch time or while your app is running or in the background. To handle incoming URLs, your delegate should implement the following methods:

- Use the `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods to retrieve information about the URL and decide whether you want to open it. If either method returns `NO`, your app's URL handling code is not called.
- In iOS 4.2 and later, use the `application:openURL:sourceApplication:annotation:` method to open the file.
- In iOS 4.1 and earlier, use the `application:handleOpenURL:` method to open the file.

If your app is not running when a URL request arrives, it is launched and moved to the foreground so that it can open the URL. The implementation of your `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` method should retrieve the URL from its options dictionary and determine whether the app can open it. If it can, return `YES` and let your `application:openURL:sourceApplication:annotation:` (or `application:handleOpenURL:`)

method handle the actual opening of the URL. (If you implement both methods, both must return `YES` before the URL can be opened.) Figure 6-2 shows the modified launch sequence for an app that is asked to open a URL.

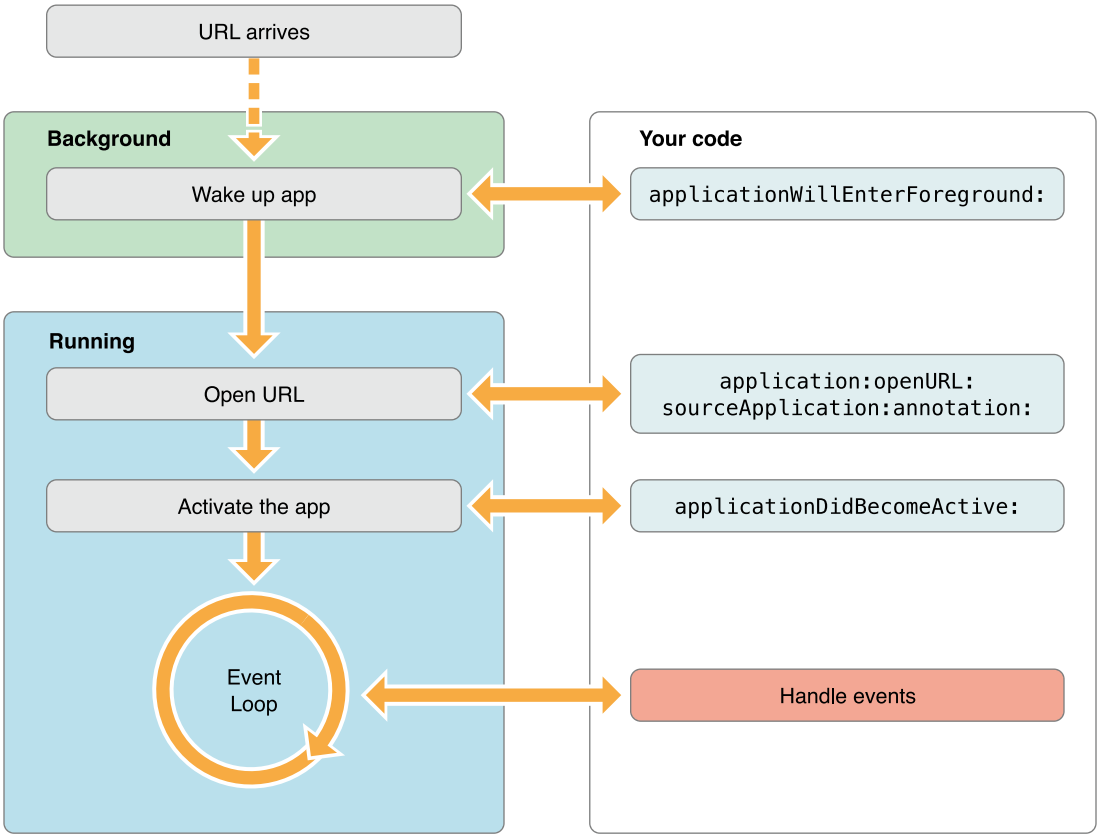
Figure 6-2 Launching an app to open a URL



If your app is running but is in the background or suspended when a URL request arrives, it is moved to the foreground to open the URL. Shortly thereafter, the system calls the delegate's `application:openURL:sourceApplication:annotation:` to check the URL and open it. If your delegate

does not implement this method (or the current system version is iOS 4.1 or earlier), the system calls your delegate's `application:handleOpenURL:` method instead. Figure 6-3 shows the modified process for moving an app to the foreground to open a URL.

Figure 6-3 Waking a background app to open a URL



Note: Apps that support custom URL schemes can specify different launch images to be displayed when launching the app to handle a URL. For more information about how to specify these launch images, see "Providing Launch Images for Custom URL Schemes" (page 103).

All URLs are passed to your app in an `NSURL` object. It is up to you to define the format of the URL, but the `NSURL` class conforms to the RFC 1808 specification and therefore supports most URL formatting conventions. Specifically, the class includes methods that return the various parts of a URL as defined by RFC 1808, including the user, password, query, fragment, and parameter strings. 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 6-1, the passed-in URL object conveys app-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 app.

This example assumes that the user is using a Gregorian calendar. If your app supports non-Gregorian calendars, you need to design your URL scheme accordingly and be prepared to handle those other calendar types in your code.

Listing 6-1 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
            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]) {
                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 = [s[NSCalendar alloc]
initWithCalendarIdentifier:NSGregorianCalendar];
            NSDate *itemDate = [calendar dateFromComponents:dateComps];
            if (!itemDate) {
                return NO;
            }
            item.dateDue = itemDate;
        }

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

Be sure to validate the input you get from URLs passed to your app; see "Validating Input and Interprocess Communication" 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*.

Showing and Hiding the Keyboard

The appearance of the keyboard is tied to the responder status of views. If a view is able to become the first responder, the system shows the keyboard whenever that view actually becomes the first responder. When the user taps another view that does not support becoming the first responder, the system hides the keyboard if it is currently visible. In UIKit, only views that support text entry can become the first responder by default. Other views must override the `canBecomeFirstResponder` method and return `YES` if they want the keyboard to be shown.

When a view becomes the first responder, the keyboard is shown by default, but you can replace the keyboard for views that support custom forms of input. Every responder object has an `inputView` property that contains the view to be displayed when the responder becomes the first responder. When this property is `nil`, the system displays the standard keyboard. When this property is not `nil`, the system displays the view you provide instead.

Normally, user taps dictate which view becomes the first responder in your app, but you can force a view to become the first responder too. Calling the `becomeFirstResponder` method any responder object causes that object to try to become the first responder. If that responder object is able to become the first responder, the custom input view (or the standard keyboard) is shown automatically.

For more information about using the keyboard, see *Text, Web, and Editing Programming Guide for iOS*.

Turning Off Screen Locking

If an iOS-based device does not receive touch events for a specified period of time, the system turns off the screen and disables the touch sensor. Locking the screen is an important way to save power. As a result, you should generally leave this feature enabled. However, for an app that does not rely on touch events, such as a game that uses the accelerometers for input, disable screen locking to prevent the screen from going dark while the app is running. However, even in this case, disable screen locking only while the user is actively engaged with the app. For example, if the user pauses a game, reenable screen locking to allow the screen to turn off.

To disable screen locking, set the `idleTimerDisabled` property of the shared `UIApplication` object to `YES`. Be sure to reset this property to `NO` when your app does not need to prevent screen locking.

Performance Tuning

At each step in the development of your app, you should consider the implications of your design choices on the overall performance of your app. The operating environment for iOS apps is more constrained than that for Mac OS X apps. The following sections describe the factors you should consider throughout the development process.

Make App Backups More Efficient

Backups occur wirelessly via iCloud or when the user syncs the device with iTunes. During backups, files are transferred from the device to the user's computer or iCloud account. The location of files in your app sandbox determines whether or not those files are backed up and restored. If your application creates many large files that change regularly and puts them in a location that is backed up, backups could be slowed down as a result. As you write your file-management code, you need to be mindful of this fact.

App Backup Best Practices

You do not have to prepare your app in any way for backup and restore operations. Devices with an active iCloud account have their app data backed up to iCloud at appropriate times. And for devices that are plugged into a computer, iTunes performs an incremental backup of the app's data files. However, iCloud and iTunes do not back up the contents of the following directories:

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

To prevent the syncing process from taking a long time, be selective about where you place files inside your app's home directory. Apps that store large files can slow down the process of backing up to iTunes or iCloud. These apps can also consume a large amount of a user's available storage, which may encourage the user to delete the app or disable backup of that app's data to iCloud. With this in mind, you should store app data according to the following guidelines:

- Critical data should be stored in the *<Application_Home>*`/Documents` directory. Critical data is any data that cannot be recreated by your app, such as user documents and other user-generated content.

- Support files include files your application downloads or generates and that your application can recreate as needed. The location for storing your application's support files depends on the current iOS version.
    - In iOS 5.1 and later, store support files in the <*Application_Home*>`/Library/Application Support` directory and add the `NSURLIsExcludedFromBackupKey` attribute to the corresponding `NSURL` object using the `setResourceValue:forKey:error:` method. (If you are using Core Foundation, add the `kCFURLIsExcludedFromBackupKey` key to your `CFURLRef` object using the `CFURLSetResourcePropertyForKey` function.) Applying this attribute prevents the files from being backed up to iTunes or iCloud. If you have a large number of support files, you may store them in a custom subdirectory and apply the extended attribute to just the directory.
    - In iOS 5.0 and earlier, store support files in the <*Application_Home*>`/Library/Caches` directory to prevent them from being backed up. If you are targeting iOS 5.0.1, see *How do I prevent files from being backed up to iCloud and iTunes?* for information about how to exclude files from backups.
- Cached data should be stored in the <*Application_Home*>`/Library/Caches` directory. Examples of files you should put in the `Caches` directory include (but are not limited to) database cache files and downloadable content, such as that used by magazine, newspaper, and map apps. Your app should be able to gracefully handle situations where cached data is deleted by the system to free up disk space.
- Temporary data should be stored in the <*Application_Home*>`/tmp` directory. Temporary data comprises any data that you do not need to persist for an extended period of time. Remember to delete those files when you are done with them so that they do not continue to consume space on the user's device.

Although iTunes backs up the app bundle itself, it does not do this during every sync operation. Apps purchased directly from a device are backed up when that device is next synced with iTunes. Apps are not backed up during subsequent sync operations, though, unless the app bundle itself has changed (because the app was updated, for example).

For additional guidance about how you should use the directories in your app, see *File System Programming Guide*.

Files Saved During App Updates

When a user downloads an app update, iTunes installs the update in a new app directory. It then moves the user's data files from the old installation over to the new app 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`

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

Use Memory Efficiently

Because the iOS virtual memory model does not include disk swap space, apps are 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 app. In addition, apps running under multitasking must share system memory with all other running apps. Therefore, make it a high priority to reduce the amount of memory used by your app.

There is a direct correlation between the amount of free memory available and the relative performance of your app. 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 suspended apps, code pages, or other nonvolatile resources from memory. However, removing those apps and resources from memory may be only a temporary fix, especially if they 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.

Observe Low-Memory Warnings

When the system dispatches a low-memory warning to your app, respond immediately. iOS notifies all running apps whenever the amount of free memory dips below a safe threshold. (It does not notify suspended apps.) If your app receives this warning, it must free up as much memory as possible. The best way to do this is to remove strong references to caches, image objects, and other data objects that can be recreated later.

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

- Implement the `applicationDidReceiveMemoryWarning:` method of your app 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. For example, the default behavior of the `UIViewController` class is to purge its view if that view is not currently visible; subclasses can supplement the default behavior by purging additional data structures. An app that maintains a cache of images might respond by releasing any images that are not currently onscreen.

If your data model includes known purgeable resources, you can have a corresponding manager object register for the `UIApplicationDidReceiveMemoryWarningNotification` notification and remove strong references to its purgeable resources directly. Handling this notification directly avoids the need to route all memory warning calls through the app delegate.

Note: You can test your app's behavior under low-memory conditions using the Simulate Memory Warning command in iOS Simulator.

Reduce Your App's Memory Footprint

Starting off with a low footprint gives you more room for expanding your app later. Table 7-1 lists some tips on how to reduce your app's overall memory footprint.

Table 7-1 Tips for reducing your app's memory footprint

Tip	Actions to take
Eliminate memory leaks.	Because memory is a critical resource in iOS, your app should have no memory leaks. You can use the Instruments app to track down leaks in your code, both in 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 disk but must be loaded into memory before they can be used. Property list files and images can be made smaller with some very simple actions. To reduce the space used by property list files, write those files out in a binary format using the `NSPropertyListSerialization` class. For images, compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iOS apps—use the `pngcrush` tool.)
Use Core Data or SQLite for large data sets.	If your app 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 framework was introduced in iOS 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 app right away. In addition, if you end up not using the resource, loading it wastes memory for no good purpose.
Build your program using the Thumb option.	Adding the `-mthumb` compiler flag can reduce the size of your code by up to 35%. However, if your app contains floating-point–intensive code modules and you are building your app for ARMv6, you should disable the Thumb option. If you are building your code for ARMv7, you should leave Thumb enabled.

Allocate Memory Wisely

Table 7-2 lists tips for improving memory usage in your app.

Table 7-2 Tips for allocating memory

Tip	Actions to take
Reduce your use of autoreleased objects.	With automatic reference counting (ARC), it is better to alloc/init objects and let the compiler release them for you at the appropriate time. This is true even for temporary objects that in the past you might have autoreleased to prevent them from living past the scope of the current method.
Impose size limits on resources.	Avoid loading a large resource file when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iOS-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 app may be unable to complete the calculations. Your apps should avoid such sets whenever possible and work on problems with known memory limits.

For detailed information about ARC and memory management, see *Transitioning to ARC Release Notes*.

Move Work off the Main Thread

Be sure to limit the type of work you do on the main thread of your app. The main thread is where your app handles touch events and other user input. To ensure that your app is always responsive to the user, you should never use the main thread to perform long-running or potentially unbounded tasks, 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 use Grand Central Dispatch (GCD) or operation objects to perform tasks asynchronously.

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

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

Floating-Point Math Considerations

The processors found in iOS-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:** If you build your app for ARMv6 and your code uses floating-point math extensively, 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. If you build your app for ARMv7, you should always enable the Thumb option.

In iOS 4 and later, you can also use the functions of the Accelerate framework to perform complex mathematical calculations. This framework contains high-performance vector-accelerated libraries for digital signal processing and linear algebra mathematics. You can apply these libraries to problems involving audio and video processing, physics, statistics, cryptography, and complex algebraic equations.

Reduce Power Consumption

Power consumption on mobile devices is always an issue. The power management system in iOS conserves power by shutting down any hardware features that are not currently being used. You can help improve battery life by optimizing your use of the following features:

- The CPU
- Wi-Fi, Bluetooth, 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 app's algorithms using Instruments. But 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 app does not need the screen to stay on, let the system turn it off. If your app experiences side effects as a result of the screen being turned 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 generally takes less power to perform a set of calculations all at once than it does to perform them in small chunks over an extended period of time. Doing small bits of work periodically requires waking up the CPU more often and getting it into a state where it can perform your tasks.
- Avoid accessing the disk too frequently. For example, if your app 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 is 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 app 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 *Event Handling Guide for iOS*.

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-intensive operation you can perform. You can minimize that time 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 simply is 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 app 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 cellular 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 *Location Awareness Programming Guide*.

The Instruments app includes several instruments for gathering power-related information. You can use these instruments to gather general information about power consumption and to gather specific measurements for hardware such as the Wi-Fi and Bluetooth radios, GPS receiver, display, and CPU. For more information about using these instruments, see *Instruments User Guide*.

Tune Your Code

iOS comes with several apps for tuning the performance of your app. Most of these tools run on Mac OS X and are suitable for tuning some aspects of your code while it runs in iOS Simulator. For example, you can use 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 Simulator, you should then use the Instruments app 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 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*.

Improve File Access Times

Minimize the amount of data you write to the disk. File operations are relatively slow and involve writing to the flash drive, 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, and 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 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, especially if the amount of data you are manipulating could grow to more than a few megabytes.

Avoid writing cache files to disk. The only exception to this rule is when your app quits and you need to write state information that can be used to put your app back into the same state when it is next launched.

Tune Your Networking Code

The networking stack in iOS includes several interfaces for communicating over the radio hardware of iOS 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.

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

Implementing code to receive or transmit data across the network is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving data 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 using chatty protocols.
- Transmit data packets in bursts whenever you can.

Cellular and Wi-Fi radios are designed to power down when there is no activity. Depending on the radio, though, doing so can take several seconds. If your app 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, packets can be lost at any time. Therefore, 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 immediately invokes browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This situation 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 app accesses the network using the Wi-Fi radios, you must notify the system of that fact by including the `UIRequiresPersistentWiFi` key in the app's `Info.plist` file. The inclusion of this key lets the system know that it should display the network selection dialog 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 app is running.

To prevent the Wi-Fi hardware from using too much power, iOS has a built-in timer that turns off the hardware completely after 30 minutes if no running app has requested its use through the `UIRequiresPersistentWiFi` key. If the user launches an app that includes the key, iOS effectively disables the timer for the duration of the app's life cycle. As soon as that app quits or is suspended, 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 app 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 Figure 6-1 (page 120).

The Airplane Mode Alert

If your app launches while the device is in airplane mode, the system may display an alert to notify the user of that fact. The system displays this alert only when all of the following conditions are met:

- Your app's information property list (`Info.plist`) file contains the `UIRequiresPersistentWiFi` key and the value of that key is set to `true`.
- Your app 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.

The iOS Environment

The iOS environment affects several aspects of how you design your app. Understanding some key aspects should help you when writing your code.

Specialized System Behaviors

The iOS system is based on the same technologies used by Mac OS X, namely the Mach kernel and BSD interfaces. Thus, iOS apps run in a UNIX-based system and have full support for threads, sockets, and many of the other technologies typically available at that level. However, there are places where the behavior of iOS differs from that of Mac OS X.

The Virtual Memory System

To manage program memory, iOS uses essentially the same virtual memory system found in Mac OS X. In iOS, each program still has its own virtual address space, but unlike Mac OS X, the amount of usable virtual memory is constrained by the amount of physical memory available. This is because iOS does not support paging to disk when memory gets full. Instead, the virtual memory system simply releases read-only memory pages, such as code pages, when it needs more space. Such pages can always be loaded back into memory later if they are needed again.

If memory continues to be constrained, the system may send low-memory notifications to any running apps, asking them to free up additional memory. All apps 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 app, see "Observe Low-Memory Warnings" (page 129).

The Automatic Sleep Timer

One way iOS saves battery power is through the automatic sleep timer. When the system does not detect touch events for an extended period of time, it dims the screen initially and eventually turns it off altogether.

If you are creating an app that does not use touch inputs, such as a game that relies on the accelerometers, you can disable the automatic sleep timer to prevent the screen from dimming. You should use this timer sparingly and reenable it as soon as possible to conserve power. Only apps that display visual content and do not rely on touch inputs should ever disable the timer. Audio apps or apps that do not need to present visual content should not disable the timer.

The process for disabling the timer is described in "Turning Off Screen Locking" (page 126). For additional tips on how to save power in your app, see "Reduce Power Consumption" (page 132).

Multitasking Support

In iOS 4 and later, **multitasking** allows apps to run in the background even when they are not visible on the screen. Most background apps reside in memory but do not actually execute any code. These apps are suspended by the system shortly after entering the background to preserve battery life. Apps can ask the system for background execution time in a number of ways, though.

For an overview of multitasking and what you need to do to support it, see "Background Execution and Multitasking" (page 54).

Security

The security infrastructure in iOS is there to protect your app's data and the system as a whole. Security breaches can and will happen, so the first line of defense in iOS is to minimize the damage caused by such breaches by securing each app separately in its own sandbox. But iOS provides other technologies, such as encryption and certificate support, to help you protect your data at an even more fundamental level.

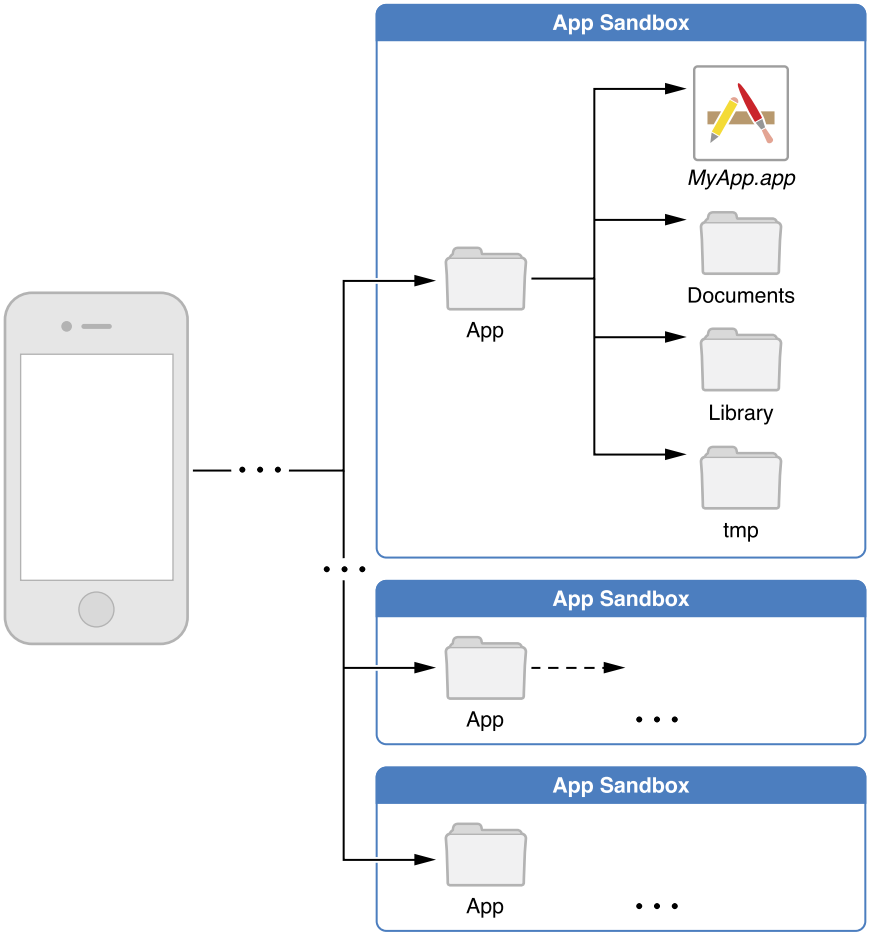
For an introduction to security and how it impacts the design of your app, see *Security Overview*.

The App Sandbox

For security reasons, iOS places each app (including its preferences and data) in a sandbox at install time. A sandbox is a set of fine-grained controls that limit the app's access to files, preferences, network resources, hardware, and so on. As part of the sandboxing process, the system installs each app in its own sandbox directory, which acts as the home for the app and its data.

To help apps organize their data, each sandbox directory contains several well-known subdirectories for placing files. Figure A-1 shows the basic layout of a sandbox directory. For detailed information about the sandbox directory and what belongs in each of its subdirectories, see *File System Programming Guide*.

Figure A-1 Sandbox directories in iOS



Important: The purpose of a sandbox is to limit the damage that a compromised app can cause to the system. Sandboxes do not prevent attacks from happening to a particular app and it is still your responsibility to code defensively to prevent attacks. For example, if your app does not validate user input and there is an exploitable buffer overflow in your input-handling code, an attacker could still hijack your app or cause it to crash. The sandbox only prevents the hijacked app from affecting other apps and other parts of the system.

Keychain Data

A **keychain** is a secure, encrypted container for passwords and other secrets. The keychain is intended for storing small amounts of sensitive data that are specific to your app. It is not intended as a general-purpose mechanism for encrypting and storing data.

Keychain data for an app is stored outside of the app's sandbox. When the user backs up app data using iTunes, the keychain data is also backed up. Before iOS 4.0, keychain data could only be restored to the device from which the backup was made. In iOS 4.0 and later, a keychain item that is password protected can be restored to a different device only if its accessibility is not set to `kSecAttrAccessibleAlwaysThisDeviceOnly` or any other value that restricts it to the current device. Upgrading an app does not affect that app's keychain data.

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

Document Revision History

This table describes the changes to *iOS App Programming Guide*.

Date	Notes
2012-09-19	Contains information about new features in iOS 6.
2012-03-07	Added information about the NSURL and CFURL keys used to prevent a file from being backed up.
2012-01-09	Updated the section that describes the behavior of apps in the background.
2011-10-12	Added information about features introduced in iOS 5.0. Reorganized book and added more design-level information. Added high-level information about iCloud and how it impacts the design of applications.
2011-02-24	Added information about using AirPlay in the background.
2010-12-13	Made minor editorial changes.
2010-11-15	Incorporated additional iPad-related design guidelines into this document. Updated the information about how keychain data is preserved and restored.
2010-08-20	Fixed several typographical errors and updated the code sample on initiating background tasks.
2010-06-30	Updated the guidance related to specifying application icons and launch images. Changed the title from *iPhone Application Programming Guide*.

Date	Notes
2010-06-14	Reorganized the book so that it focuses on the design of the core parts of your application. Added information about how to support multitasking in iOS 4 and later. For more information, see "Core App Objects" (page 17). Updated the section describing how to determine what hardware is available. Added information about how to support devices with high-resolution screens. Incorporated iPad-related information.
2010-02-24	Made minor corrections.
2010-01-20	Updated the "Multimedia Support" 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" chapter for iOS 3.1.
2009-06-17	Added information about using the compass interfaces. Moved information about OpenGL support to *OpenGL ES Programming Guide for iOS*. Updated the list of supported `Info.plist` keys.
2009-03-12	Updated for iOS 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.

Date	Notes
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 iOS information to *iOS Technology Overview*. Moved information about the standard system URL schemes to *Apple URL Scheme Reference*. Moved information about the development tools and how to configure devices to *Tools Workflow Guide for iOS*. 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 iOS and the development process for iPhone applications.

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