Examples and solutions for iPhone & iPad

iOS 4 IN ACTION

J. Harrington
B. Trebitowski
C. Allen
S. Appelcline

SAMPLE CHAPTER

MANNING

iOS 4 in Action

by Jocelyn Harrington, Brandon Trebitowski,
Christopher Allen, Shannon Appelcline

Chapter 1

brief contents

1

Introducing iOS 4 with iPhone and iPad

This chapter covers

- Understanding Apple's iPhone and iPad technology
- Installing the iOS 4 SDK
- Anatomy of iOS
- Turning your idea into an iOS application

The iPhone and iPad provide an unforgettable user experience. It's one of the rare technologies that's so intuitive that even a toddler can use it without a user manual. iOS provides a whole platform for developers. It comes with a huge global market and one integrated distribution place: the App Store. The iOS SDK offers a rich set of APIs for developers to turn their best ideas into killer applications. The new enhancements in iOS 4 allow developers to create applications faster and easier.

In this chapter, we'll first introduce iOS 4 and then go over the key specifications of the iPhone, iPad, and iPod Touch. We'll cover the anatomy of iOS, including frameworks, windows, views, and methods. We'll also cover events, memory management, and lifecycle management before providing tips on creating a successful application. Let's start the story with the iOS platform.

1.1 All for one and one for all: the iOS platform

The iPod Touch, iPhone, and iPad (and likely future generations of Apple devices) all use iOS 4.3.1 (at the time of writing). The iOS moniker may be a bit confusing at first, but having one OS for all these devices makes it an easy and rewarding platform on which to develop. Learn how to develop for it once using the iOS SDK, and you can adapt your applications to whichever devices you like. For example, you can determine that the application will support only the devices with GPS or camera.

Let's review a bit of history on iOS. The iOS SDK was first introduced in 2007 and released in March of 2008. The third major release, iOS 3.0, was released in 2009. Prior to iOS 4.2, there was a short, fragmented OS history on the iPhone and iPad; the iPhone was running on iOS 4.0 and the iPad was on iOS 3.2. With the new iOS, all the iOS-powered devices can once again run the same OS. For developers, the experience for application development is a lot smoother and easier. The most prominent feature on iOS 4 is that iOS supports multitasking services, including playing audio, push notifications, receiving location change events, and fast app switching. We'll cover the details later in this book.

The social experience is emphasized on iOS 4 with Game Center and iTunes 10 with Ping. Game Center allows developers to create social game experiences with the Game Kit framework. For end users, it's amazing to start multiplayer games through automatching, tracking their achievements, and so on.

There are differences in developing applications for the iPad as opposed to the iPhone, but they're primarily related to the varying amount of real estate available to each device, as illustrated in figure 1.1. Obviously, the iPad has a much bigger screen for display or interaction. The content focus is to provide a rich information presentation. In the UI design, you may want to distinguish the iPad from the iPhone. For the most part, you can run the examples in this book on either the iPad or the iPhone with little adaptation. (iPhone applications are fully compatible on the iPad as is; universal applications support different experiences depending on the platform they're being run on.)

One more thing: iOS 4.3 allows applications to support printing through Airprint. Imagine that you can edit your photo with the iPad or iPhone and tap the Print button to get the photo printed out on your wi-fi printer! We'll cover this function in detail later in chapter 11.

1.2 Understanding iPhone and iPad touch interaction

The iPhone and iPad use a multitouch-capable capacitive touchscreen. Users access the device by tapping around with their finger. But *a finger isn't a mouse*. Generally, a finger is larger and less accurate than a more traditional pointing device. This disallows certain traditional types of UI that depend on precise selection. For example, the iPhone and iPad don't have scrollbars. Selecting a scrollbar with a fat finger would either be an exercise in frustration or require a huge scrollbar that would take up a lot of the iPhone's precious screen real estate. Apple solved this problem by allowing

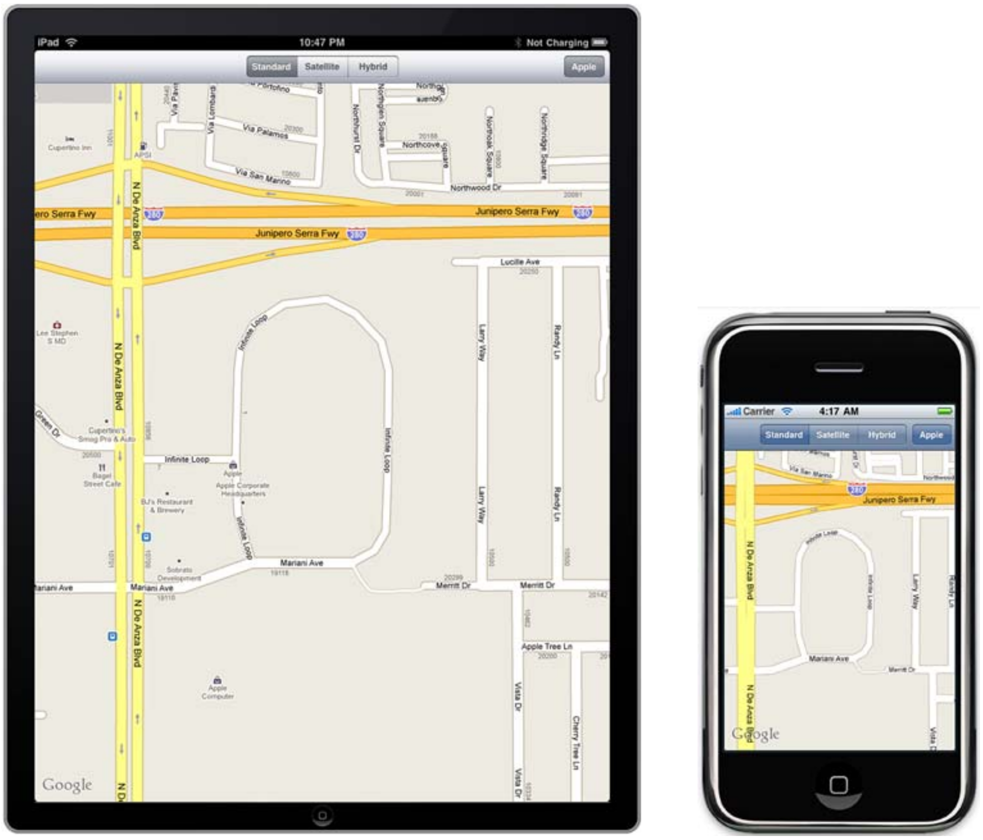


Figure 1.1 The iPad and iPhone side by side. The primary difference between the two—the available screen real estate—is readily apparent.

users to tap anywhere on an iPhone screen and then *flick* in a specific direction to cause scrolling.

Another interesting element of the touchscreen is shown off by the fact that a *finger isn't necessarily singular*. Recall that the iPhone and iPad touchscreens are *multitouch*. This allows users to manipulate the device with multifinger *gestures*. Pinch-zooming is one such example. To zoom into a page, you tap two fingers on the page and then push them apart; to zoom out, you similarly push them together.

Finally, *a finger isn't persistent*. A mouse pointer is always on the display, but the same isn't true for a finger, which can tap here and there without going anywhere in between. As you'll see, this causes issues with some traditional web techniques that depend on a mouse pointer moving across the screen. It also provides limitations that may be seen throughout SDK programs. For example, there's no standard for cut and paste, a ubiquitous feature for any computer produced in the last couple of decades.

In addition to some changes to existing interfaces, the input interface introduces a number of new touches (one-fingered input) and gestures (two-fingered input), as described in table 1.1.

Table 1.1 iPhone and iPad touches and gestures allow you to accept user input in new ways.

Input	Type	Summary
Bubble	Touch	Touch and hold. Pops up an info bubble on clickable elements.
Flick	Touch	Touch and flick. Scrolls the page.
Flick, two-finger	Gesture	Touch and flick with two fingers. Scrolls the scrollable element.
Pinch	Gesture	Move fingers in relation to each other. Zooms in or out.
Tap	Touch	A single tap. Selects an item or engages an action such as a button or link.
Tap, double	Touch	A double tap. Zooms a column.

When you're designing with the SDK, many of the nuances of finger mousing are taken care of for you. Standard controls are optimized for finger use, and you have access only to the events that work on the iPhone or iPad. Chapter 6 explains how to use touches, events, and actions in iOS; as an iOS developer, you'll need to change your way of thinking about input to better support the new devices.

1.3 Getting ready for the SDK

The iOS software development kit (SDK) is a suite of programs available in one gargantuan (at the time of writing over 4 GB) download from Apple. It gives you the tools you need to program (Xcode—Xcode 4 is the version of the IDE used in this book), debug (Instruments), and test (Simulator) your iPhone, iPod Touch, and iPad code.

NOTE You must have an Intel-based Apple Macintosh running Mac OS X 10.6.5 or higher to use the SDK.

1.3.1 Installing the SDK

To obtain the SDK, download it from Apple's iOS Dev Center, which at the time of this writing is accessible at http://developer.apple.com/devcenter/ios/. You'll need to register as an iOS Developer in order to get there, but it's a fairly painless process. Note that you can also use this site to access Apple documentation and sample source code.

NOTE Xcode 4 is a free download for all members of the iOS Developer Program, which costs US$99 per year. If you're not an iOS Developer Program member, you can purchase Xcode 4 from the Mac App Store for US$4.99 or download Xcode 3 for free.

THE APPLE DOCS AND THE SDK

To see the full API documentation as well as sample code, visit http://developer.apple.com/devcenter/ios/. It contains a few introductory papers, of which we think the best are "iOS Overview" and "Learning Objective-C: A Primer," plus the complete class and protocol references for the SDK.

As we'll discuss in the next chapter, you can also access all of these docs from inside Xcode. We usually find Xcode a better interface because it allows you to click through from your source code to your local documents. Nonetheless, the website is a great source of information when you don't have Xcode handy.

Because they tend to be updated relatively frequently, we've been constantly aware of Apple's documents while writing this book, and we've done our best to ensure that what we include complements Apple's information. We'll continue to provide you with introductions to topics and to point you toward the references when there's a need for in-depth information.

After you've downloaded the SDK, you'll find that it leaves a disk image sitting on your hard drive. Double-click it to mount the disk image, and then double-click Xcode and iOS SDK in the folder that pops up to start the installation process (as shown in figure 1.2).

This will bring you through the entire install process, which will probably take 20–40 minutes. You'll also get a few licensing agreements that you need to sign off on, including the iPhone Licensing Agreement, which lists some restrictions on what you'll be able to build for the iOS-based devices.

Warning: installation dangers

The default installation of Xcode and iOS SDK will replace any existing Apple development tools you have. You'll still be able to do regular Apple development, but you'll be working with a slightly more bleeding-edge development environment.

IOS SDK LICENSING RESTRICTIONS

Although Apple is making the iOS SDK widely available for public programming, the company has placed some restrictions on what you can do with it. We expect these restrictions to change as the SDK program evolves, but what follows are some of the limitations at the time of this writing.

Among the most notable technical restrictions: you can't use the code to create plug-ins, nor can you use it to download non-SDK code. It was the latter that apparently spoiled Sun's original plans to port Java over to the iPhone. You also can use only Apple's published APIs. In addition, there are numerous privacy-related restrictions, the most important of which is that you can't log the user's location without permission. Finally, Apple has some specific application restrictions, including restrictions on apps that incorporate pornography or other objectionable content.

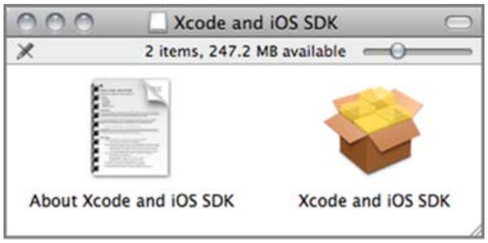


Figure 1.2 **Double-clicking Xcode and iOS SDK starts your installation.**

In order for your program to run on iPhones and iPads, you'll need an Apple certificate, and Apple maintains the right to refuse your cert if it doesn't like what you're doing. If you're planning to write anything that might be questionable, you should probably check whether Apple is likely to approve it first. For example, the most-used third-party software package that isn't available natively is Flash. This book is going to cover how to develop an app on iPhone or iPad with the iOS SDK. But there's another way to deliver an application to iPhone or iPad: using HTML5+JavaScript for web apps. We won't cover web application development in this book.

When the SDK finishes installing, you'll find it in the /Developer directory of your Mac system disk. Most of the programs appear in /Developer/Applications, which we suggest you make accessible using the Add to Sidebar feature in the Finder. The iOS Simulator is located separately at /Developer/Platforms/iPhoneSimulator.platform/Developer/Applications. Because this is off on its own, you may want to add it to your Dock.

You now have everything you need to program for the iOS devices, but you won't be able to release iPhone or iPad programs on your own—that takes a special certificate from Apple. See appendix C for complete information on this process, which is critical for moving your programs from the Simulator onto a real device. The Simulator turns out to be one of several programs you've installed, each of which can be useful in SDK programming.

1.3.2 The anatomy of the SDK

Xcode, Instruments, and Dashcode were all available as part of the development library of Mac OS X before the iPhone came along. Many of these programs are expanded and revised for use on the iPhone, so we've opted to briefly summarize them all, in decreasing order of importance to an SDK developer:

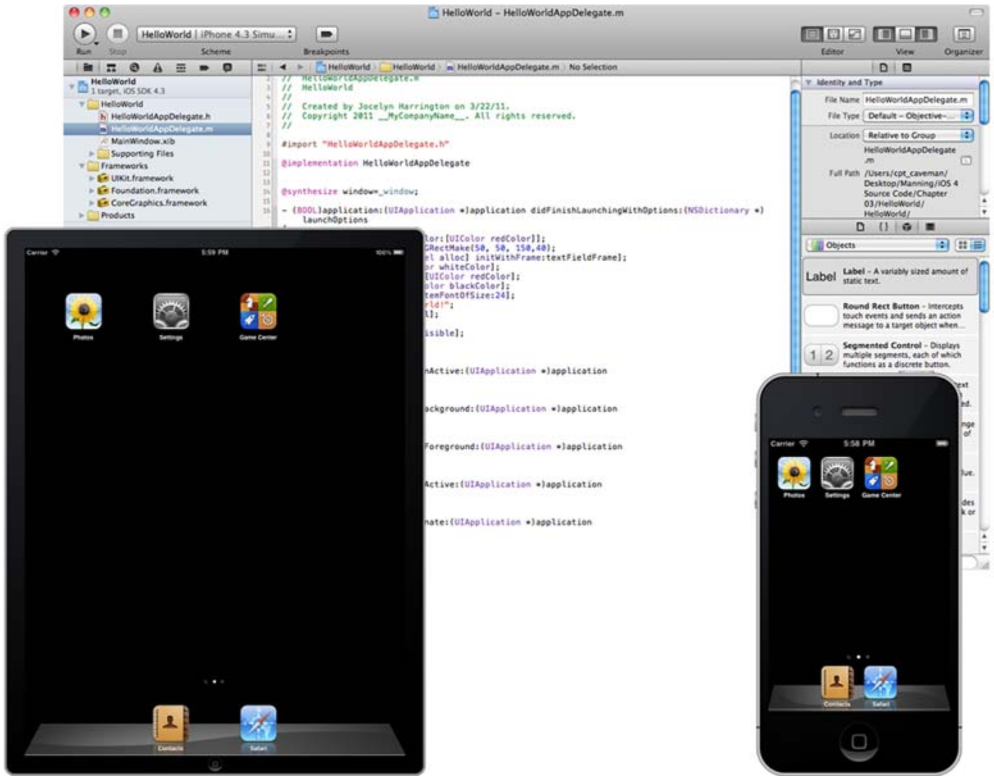
- *Xcode 4* is the core of the SDK's integrated development environment (IDE). It's where you'll set up projects, write code in a text editor, compile code, and generally manage your applications. It supports code written in Objective-C (a superset of C that we'll cover in more depth in the next chapter) and can also parse C++ code. Interface Builder is now a part of Xcode 4, and it allows you to put together the graphical elements of your program, including windows and menus, via a quick, reliable method. You'll learn the specifics of how to use Xcode 4 in chapters 3 and 4.
- *iOS Simulator* allows you to view an iPhone or iPad screen on your desktop. It's a great help for debugging web pages. It's an even bigger help when you're working on native apps, because you don't have to get your code signed by Apple to test it out.
- *Instruments* is a program that allows you to dynamically debug, profile, and trace your program. If you were creating web apps, we would have to point you to a slew of browsers, add-ons, and remote websites to do this sort of work; but for your native apps, that's all incorporated into this one package.

- *Dashcode* is listed here only for the sake of completeness because it's part of the /Developer area. It's a graphical development environment that's used to create web-based programs incorporating HTML, CSS, and JavaScript. Dashcode is used when developing for the web; you won't use it with the iOS SDK.

JUMPING AHEAD

If you'd prefer to immediately dive into your first program, HelloWorld, head to chapter 3. You can then pop back here to see what it all means.

Figure 1.3 shows the most important developer tools. In addition to the visible tools that you've downloaded into /Developer, you've also downloaded the entire set of iOS frameworks: a huge collection of header files and source code—all written in Objective-C—that will greatly simplify your programming experience. In the next chapter, we'll look at Objective-C, the SDK's programming language. Rather than jumping straight into your first program, we instead want to touch on these foundational topics. In the next section, we'll examine some of the basics of iOS.



```
//  HelloWorldAppDelegate.m
//  HelloWorld
//
//  Created by Jocelyn Harrington on 3/22/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.
//

#import "HelloWorldAppDelegate.h"

@implementation HelloWorldAppDelegate


@synthesize window=_window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)
    launchOptions
                    lor:[UIColor redColor]];
                    GRectMake(50, 50, 150,40);
                    el alloc] initWithFrame:textFieldFrame];
                    or whiteColor];
                    [UIColor redColor];
                    olor blackColor];
                    temFontOfSize:24];
                    rld!";
                    l];

                    isible];


                    nActive:(UIApplication *)application


                    ckground:(UIApplication *)application


                    Foreground:(UIApplication *)application


                    Active:(UIApplication *)application


                    nate:(UIApplication *)application
```

```
File Name   HelloWorldAppDelegate.m
File Type   Default – Objective-...
Location    Relative to Group
            HelloWorldAppDelegate.m
Full Path   /Users/cpt_caveman/Desktop/Manning/iOS 4 Source Code/Chapter 03/HelloWorld/HelloWorld/

Label - A variably sized amount of static text.
Round Rect Button - Intercepts touch events and sends an action message to a target object when...
Segmented Control - Displays multiple segments, each of which functions as a discrete button.
```

Figure 1.3 The SDK includes Xcode (top) and two instances of the iOS Simulator, running in iPad mode (bottom left) and iPhone mode (right).

1.4 Introducing iOS

Apple's iOS SDK provides you with a vast library of objects arranged into several frameworks. As a result, you'll spend a lot more time sending messages to objects that are ready-made for your use than creating new ones. Let's begin our look at iOS by exploring several of these objects and how they're arranged. We'll take a tour of the anatomy of iOS, at how the object hierarchy is arranged, and how iOS handles windows and views.

1.4.1 The anatomy of iOS

iOS's frameworks are divided into four major layers, as shown in figure 1.4.

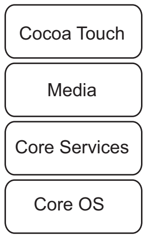


Figure 1.4 Apple provides you with four layers of frameworks to use when writing iOS programs.

Each of these layers contains a variety of frameworks that you can access when writing iOS SDK programs. Generally, you should prefer the higher-level layers when you're coding (those shown toward the top in the diagram).

Cocoa Touch is the framework that you'll become most familiar with. It contains the UIKit framework—which is what we spend most of our time on in this book—and the Address Book UI framework. UIKit includes window support, event support, and user-interface management, and it lets you display both text and web pages. It further acts as your interface to the accelerometers, the camera, the photo library, and device-specific information.

Media is where you can get access to the major audio and video protocols built into the iPhone and iPad. Its four graphical technologies are OpenGL ES, EAGL (which connects OpenGL to your native window objects), Quartz (which is Apple's vector-based drawing engine), and Core Animation (which is also built on Quartz). Other frameworks of note include Core Audio, Open Audio Library, and Media Player.

Core Services offers the frameworks used in all applications. Many of them are data related, such as the internal Address Book framework. Core Services also contains the critical Foundation framework, which includes the core definitions of Apple's object-oriented data types, such as its arrays and sets.

Core OS includes kernel-level software. You can access threading, files, networking, other low-level I/O, and memory functions.

Most of your programming work will be done using the UIKit (UI) or Foundation (NS) framework. These libraries are collectively called Cocoa Touch; they're built on Apple's modern Cocoa framework, which is almost entirely object oriented and, in our opinion, much easier to use than older libraries. The vast majority of code in this book will be built solely using Cocoa Touch.

But you'll sometimes have to fall back on libraries that are instead based on simple C functionality. Examples include Apple's Quartz 2D and Address Book frameworks, as well as third-party libraries like SQLite. Expect object creation, memory management, and even variable creation to work differently for these non-Cocoa libraries.

When you fall back on non-Cocoa libraries, you'll sometimes have to use Apple's Core Foundation framework, which lies below Cocoa. Your first encounter with Core Foundation will be when we discuss the Address Book framework in chapter 9; we'll provide more details about how to use Core Foundation at that point.

Although Core Foundation and Cocoa are distinct classes of frameworks, many of their common variable types are *toll-free bridged*, which means they can be used interchangeably as long as you cast them. For example, `CFStringRef` and `NSString *` are toll-free bridged, as you'll see when we talk about the Address Book. The Apple class references usually point out this toll-free bridging for you.

1.4.2 The object hierarchy of iOS

Within these frameworks, you can access an immense wealth of classes arranged in a huge hierarchy. You'll see many of these used throughout this book, and you'll find a listing of even more in appendix A. Figure 1.5 shows many of the classes that you'll use over the next several chapters, arranged in a hierarchy. They're a fraction of what's available.

THE NS CLASSES

The NS classes come from Core Services' Foundation framework (the Cocoa equivalent of the Core Foundation framework), which contains a huge number of fundamental data types and other objects.

You should use the fundamental Cocoa classes like `NSString` and `NSArray` whenever you can, rather than C fundamentals like `char*` or a plain array. This is because they tend to play nicely with each other and with the UIKit frameworks, and therefore you're less likely to encounter bizarre errors. They also follow the memory-management rules of Objective-C (reference counting). Although it isn't shown, `NSNumber` is another class you should be aware of. Although it shouldn't be used in place of an ordinary number, it serves as a great wrapper when you need a number expressed as an object. This is useful for sending numbers via message passing. `NSNumber` is capable of holding many sorts of numerical values, from floats to integers and more.

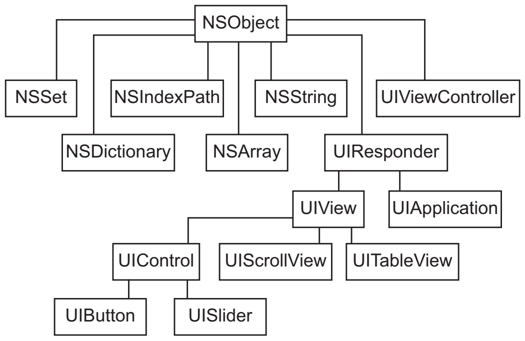


Figure 1.5 This hierarchy graph shows a small selection of the classes available in iOS.

The objects that can hold collections of values like `NSArray` (a numerical array) and `NSDictionary` (an associative array) are picky about your sticking to their NS brethren. You'll need to wrap C variables inside Cocoa classes whenever you hand off objects to these arrays. Finally, though `NSString` can take many sorts of objects when you're formatting a string, you should be aware that Cocoa objects may require a different formatting string than their C equivalents.

In two situations, you'll find that these NS classes can be a deficit. First, if you're using the Core Foundation framework, you'll often have to take advantage of toll-free bridging by casting variables, as you'll see starting in chapter 9, when we look at the Address Book. Second, if you're using external APIs, you may need to convert some classes into their C equivalents. Chapter 9's look at the SQLite API explores this possibility, with `NSString` objects often being converted to their UTF-8 equivalent.

The most important of Cocoa's Foundation objects is `NSObject`, which contains a lot of default behavior, including methods for object creation and memory management; you'll learn about these later in this chapter.

THE UI CLASSES

The second broad category contains the UI classes. These come from Cocoa Touch's UIKit framework, which includes all the graphical objects you'll be using as well as all the functionality for the iOS's event model, much of which appears in `UIResponder`. That's another topic we'll return to soon.

1.4.3 Windows and views

As the UI classes demonstrate, iOS is deeply rooted in the idea of a graphical user interface. Therefore, let's finish our introduction to iOS by looking at some of the main graphical abstractions embedded in the UIKit. There are three major abstractions: windows, views, and view controllers.

A *window* is something that spans the device's entire screen. An application usually has only one, and it's the overall container for everything your application does.

A *view* is the content holder in your application. You may have several of them, each covering different parts of the window or doing different things at different times. They're all derived from the `UIView` class. But don't think of a view as a blank container. Almost any object you use from UIKit will be a subclass of `UIView` that features a lot of behavior of its own. Among the major subclasses of `UIView` are `UIControl`, which gives you buttons, sliders, and other items with which users may manipulate your program, and `UIScrollableView`, which gives users access to more text than can appear at once.

A *view controller* does what its name suggests. It acts as the controller element of the Model-View-Controller triad and in the process manages a view, sometimes called an *application view*. As such, it takes care of events and updating for your view.

In this book, we've divided view controllers into two types. *Basic view controllers* manage a screenful of information (such as the table view controller), whereas *advanced view controllers* let a user move around among several subviews (such as the navigation bar controller and the tab bar controller).

Windows, views, and view controllers are ultimately part of a *view hierarchy*. This is a tree of objects that begins with the window at its root. A simple program may have a window with a view under it. Most programs start with a window and have a view controller under that, perhaps supported by additional view controllers, each of which controls views that may have their own subviews. We'll illustrate this concept more clearly in chapter 5 when we start looking at the basic view controllers that make this sort of hierarchy possible.

1.5 iOS's methods

As you've seen, iOS has a complex and deep structure of classes. In this section, we look at object creation, memory management, event response, and lifecycle management.

Two of the most important classes are `NSObject` and `UIResponder`, which contain many of the methods and properties you'll use throughout your programming. Thanks to inheritance, these important functions (and others) can be used by many different iOS objects. We cover some of these foundational methods here to provide a single reference for their usage, but we'll be sure to point them out again when you encounter them for the first time in future chapters.

1.5.1 Object creation

We talked earlier about how to define classes; but as we said at the time, the specifics of *how* instance objects are created from classes depend on the implementation of your framework. In iOS, the `NSObject` class defines how object creation works.

You'll meet a few different interfaces that are used to support object creation, but they all ultimately fall back to a two-step procedure that uses the `alloc` class method and the `init` instance method. The `alloc` method allocates the memory for your object and returns the object itself. The `init` method then sets some initial variables in that method. They usually occur through a single, nested message:

```
id newObject = [[objectClass alloc] init];
```

The `alloc` method from `NSObject` should always do the right thing for you. But when you write a new subclass, you'll almost always want to write a new `init` method, because that's where you define the variables that make your class what it is. Here's a default setup for an `init`, which would appear as part of your `@implementation`:

```
- (id)init
{
    if (self = [super init]) {
// Instance variables go here
    }
    return self;
}
```

This code shows all the usual requirements of an `init` method. First, it calls its parent to engage in its class's initialization. Then, it sets any instance variables that should be set. Last, it returns the object, usually with `return self;`.

The bare `init` is one of a few major ways you can create objects in iOS.

THE ARGUMENTATIVE ALTERNATIVE

Sometimes you'll want to send an argument with an `init`. You can do so with an initialization function that you name using the format `initWithArgument:(argument)`. Other than the fact that you're sending it an argument, it works exactly like a bare `init`. Here's another example drawn from code you'll see in upcoming chapters:

```
[[UITextView alloc] initWithFrame:textFieldFrame];
```

Initialization methods with arguments allow you to create nonstandard objects set up in ways that you choose. They're common in `UIKit`.

One initialization method with an argument deserves a bit of extra mention. `initWithCoder:` is a special initialization method that's called whenever you create an object with Interface Builder—and important if you want to do setup for such objects. We'll return to Interface Builder in chapter 3.

THE FACTORY METHOD ALTERNATIVE

A final sort of `init` supported through iOS is the factory method (class method). This is a one-step message that takes care of both the memory allocation and initialization for you. All factory methods are named with the format `objecttypeWithArgument:(argument)` Here's another real example:

```
[UIButton buttonWithType:UIButtonTypeRoundedRect];
```

Class (or factory) methods make messaging a little clearer. They also have the advantage of handling some memory management, which is the topic of the next major category of iOS methods.

OBJECT CREATION WRAP-UP

We've summarized the four major ways that iOS supports the creation of objects in table 1.2. As witnessed by the examples, you'll use all these methods as you move through the upcoming chapters.

Table 1.2 iOS supports several methods that you can use to create objects. Different methods are supported by different classes.

Method	Code	Summary
Simple	`[[object alloc] init];`	Plain initialization
Argument	`[[object alloc] initWithArgument:argument];`	An initialization where one or more arguments is passed to the method
Coder	`[[object alloc] initWithCoder:decoder];`	An initialization with an argument used for Interface Builder objects
Factory	`[object objecttypeWithArgument:argument];`	A one-step initialization process with an argument

1.5.2 Memory management

Because of power considerations, iOS doesn't support garbage collection. That means every object that's created must eventually have its memory released by hand—at least, if you don't want to introduce a memory leak into your program.

The fundamental rule of memory management in iOS is this: if you allocate memory for an object, you must release it. This is done via the `release` message (which is once again inherited from `NSObject`):

```
[object release];
```

Send that message when you've finished using an object, and you've done your proper duty as a programmer.

Note that we said you must release the memory only if you *allocated* the memory for it. You are considered to "own" the memory for an object if you created it using a method that contains *alloc*, *new*, *copy*, or *mutableCopy*. You can free memory for an object using the release message as mentioned earlier; however, an easier way in general is making use of the wonders of autorelease. (Factory methods like `UIButton`'s `buttonWithType:` return objects that are already autoreleased, so you don't need to manage their memory unless you explicitly retain it.)

THE AUTORELEASE ALTERNATIVE

If you're responsible for the creation of an object and you're going to pass it off to some other class for usage, you should autorelease the object before you send it off. This is done with the `autorelease` method:

```
[object autorelease];
```

You'll typically send the `autorelease` message just before you return the object at the end of a method. After an object has been autoreleased, it's watched over by a special `NSAutoreleasePool`. The object is kept alive for the scope of the method to which it's been passed, and then the `NSAutoreleasePool` cleans it up.

RETAINING AND COUNTING

What if you want to hold onto an object that has been passed to you and that will be autoreleased? In that case, you send it a `retain` message:

```
[object retain];
```

When you do this, you're saying you want the object to stay around, but now you've become responsible for its memory as well: you must send a `release` message at some point to balance your `retain`.

At this point, we should probably back up and explain the underlying way that iOS manages memory objects. It does so by maintaining a count of object usage. By default, it's set to 1. Each `retain` message increases that count by 1, and each `release` message reduces that count by 1. When the count drops to 0, the memory for the object is freed up.

Therefore, all memory management can be thought of as pairs of messages. If you balance every `alloc` and every `retain` with a `release`, your object will eventually be freed up when you've finished with it.

> **WARNING** Memory management can be the root cause of the bugs. Instruments is a good tool for attempting to diagnose issues with memory leaks.

Whenever you use the keyword `retain` or `alloc`, make sure to release. If the object is already released, don't try to access the released object. A good habit would be to assign `nil` to a released object and check the value isn't `nil` before accessing the object.

MEMORY MANAGEMENT WRAP-UP

Table 1.3 provides a quick summary of the methods we've looked at to manage the memory used by your objects.

Table 1.3 The memory-management methods help you keep track of the memory you're using and clean it up when you're finished.

Method	Summary
`alloc`	Part of the object-creation routine that allocates the memory for an object's usage.
`autorelease`	Request to reduce an object's memory count by 1 when it goes out of scope. This is maintained by an NSAutoreleasePool.
`release`	Reduces the object's memory count by 1.
`retain`	Increases the object's memory count by 1.

For more information on memory management, including a look at the `copy` method and how this all interacts with properties, look at Manning's *Objective-C Fundamentals* (Christopher Fairbairn, Collin Ruffenach, and Johannes Fahrenkrug, 2011). A good description of memory-management rules is also found in the "Memory Management Programming Guide" on the Mac Developer Library website.

1.5.3 *Event response*

The next-to-last category of methods that we examine for iOS is event response. Unlike object creation and memory management, we tackle this issue only briefly, because it's much better documented in chapter 6. The topic is important enough that we want to offer a quick overview of it now.

Events can appear on the iPhone or iPad in three main ways: through bare events (or actions), through delegated events, and through notification. Whereas the methods of our earlier topics all derived from `NSObject`, event response instead comes from the `UIResponder` object, whereas notification comes from the `NSNotification-Center`. You won't have to worry about accessing responder methods and properties because `UIResponder` is the parent of most UIKit objects, but the `NSNotification-Center` requires special access.

EVENTS AND ACTIONS

Most user input results in an *event* being placed into a *responder chain*. This is a linked set of objects that, for the most part, goes backward up through the view hierarchy. Any input is captured by the *first responder*, which tends to be the object the user is directly interacting with. If that object can't resolve the input, it sends it up to its *superview* (for example, a label might send it up to its full-screen view), then to its superview, all the way up the chain (up through the views, then up through the view controllers). If input gets all the way up the view hierarchy to the window object, it's next sent to the application itself, which tends to pass it off to an *application delegate* as a last resort.

Any of these objects can choose to handle an event, which stops its movement up the responder chain. Following the standard MVC model, you'll often build event response into `UIViewController` objects, which are pretty far up the responder chain.

For any `UIControl` objects, such as buttons, sliders, and toggles, events are often turned into *actions*. Whereas events report touches to the screen, actions instead report manipulations of the controls and are easier to read. Actions follow a slightly different hierarchy of response.

DELEGATES AND DATA SOURCES

Events can be sent to an object in a way other than via a first responder: through a *delegate*. This is an object (usually a view controller) that says it will take care of events for another object (usually a view). It's close kin to a data source, which is an object (again, usually a view controller) that promises to do the data setup and control for another object (again, usually a view).

Delegation and data sourcing are each controlled by a *protocol*, which is a set of methods the delegate or data source agrees to respond to. For example, a table's delegate might have to respond to a method that alerts it when a row in the table has been selected. Similarly, a table's data source might describe what all the rows of the table look like.

Delegates and data sources fit cleanly into the MVC model used by Objective-C, because they allow a view to hand off its work to its controller without having to worry about where each of those objects is in the responder chain.

NOTIFICATIONS

Standard event response and delegation represent two ways that objects can be alerted to standard events, such as fingers touching the screen. A third method can also be used to program many different sorts of activities, such as the device's orientation changing or a network connection closing: the notification.

Objects register to receive a certain type of notification with the `NSNotification-Center` and afterward may process those notifications accordingly. Again, we'll discuss this topic in chapter 6.

1.5.4 Lifecycle management

In this discussion, we've neglected a topic: how to recognize when objects are being created and destroyed—starting with your application. With multitasking enabled in iOS 4, you can create custom behavior before or after your application enters background mode. We'll cover more details on this topic in chapter 21.

Table 1.4 summarizes some of the important messages that will be sent as part of the lifecycle of your program. To respond to them, you fill in the contents of the appropriate methods in either an object or its delegate—which requires writing a subclass and is one of the prime reasons to do so.

Table 1.4 Several important methods let you respond to the lifecycle of your application or its individual objects.

Method	Object	Summary
`application:DidFinishLaunchingWithOptions:`	`UIApplicationDelegate`	Application has loaded. You should create initial windows and otherwise start your program.
`applicationDidReceiveMemoryWarning:`	`UIApplicationDelegate`	Application received a low-memory warning. You should free up memory.
`applicationWillTerminate:`	`UIApplicationDelegate`	Application is about to end. You should free up memory and save state.
`init:`	`NSObject`	Object is being created. You should initialize it here.
`dealloc:`	`NSObject`	Object is freeing up its memory. You should release any objects that haven't been autoreleased.

Note that we've included `init:` here, because it forms a natural part of the object lifecycle. You should look at the individual Apple class references, particularly `UIApplicationDelegate`, for other methods you may want to respond to when writing programs.

With that, we've completed our look at the big-picture methods of iOS. You've not yet seen them in real use, so bookmark these pages—we'll refer to them when you begin programming in chapter 3.

1.6 How to make an application from an idea

At the beginning of this chapter we talked about turning great idea in to a killer application. How do you do it? Let's walk through the general steps to help make your dreams come true.

1.6.1 *The checklist*

There are several ways to build a universal application running on both the iPhone and iPad. Let's start with a handy checklist. (If you've already installed the Xcode and iOS SDK as demonstrated earlier in this chapter, you've finished half of the task.)

1. Join Apple's iOS Developer Program (US$ 99/year will give you access to submit applications to the App Store).
2. Have access to an Intel-based Mac computer with Mac OS X 10.6 or above.
3. Get a good book for beginners (this book, for example).
4. Get a test device: iPhone, iPod touch, or iPad. It will be mainly for testing during application development. If you're on a tight budget and don't already own one of these devices, you don't have to purchase one. There are test device services that provide rental equipment.
5. Get a sketchbook for the UI design, or make use of UI mocking software such as that from Balsamiq.

That's it! The last step is to learn Objective-C by reading this book and build your application with the iOS SDK. With this goal in mind, let's move on to the application concept.

1.6.2 *What's the category for your application?*

Knowing the category your application fits into will help you make a better estimate of how difficult it will be to build, which will help you plan your release date. Let's review the most common app categories.

GAMES AND ENTERTAINMENT

This category is hot and crowded. It's super competitive to create a successful application under games or entertainment. A game or entertainment application generally is heavy on media. As a developer, you should consider working with a UI designer in order to take advantage of the awesome graphic display quality of the iPhone and iPad.

Generally speaking, a game application may combine the use of the accelerometer, drawing and animations, audio, and Game Kit. Once you're familiar with the iOS SDK basics, you can jump to later chapters that will cover the iOS frameworks in detail. Chapter 15 provides complete coverage of game application development.

The difficulty level for a developer of game applications is higher compared to other categories. Fortunately, the rich UI tools on iOS provide a decent, fast prototyping environment for game developers.

RICH CONTENT APPLICATIONS

Rich content applications are commonly data oriented—for instance, a Twitter application. To allow user access data from the cloud, rich content applications provide organized data on the client side.

On the iPhone, the challenge of this rich content application is the limited screen real estate. The key to success, therefore, is in presenting the user with a good amount of well-organized data.

On the iPad, the focus is to provide rich content on one screen. The iPad screen can be compared to a book. With a detail-oriented UI design, you can add plenty of realism to your application. For example, the page-flipping animation in an iBook application allows users to flip through a digital book as if it were a traditional paper book.

This type of application needs to download data from the server and then store the data locally. Chapters 8 and 9 provide a great introduction on how to store data locally on the iPhone and iPad. Chapter 14 demos how to fetch data through Internet protocols on iOS.

NAVIGATION AND TRAVEL APPLICATIONS

This category makes more sense on the iPhone compared to the iPad. With its built-in GPS and compass, the iPhone can be used to provide the user's current location on the fly.

If you're thinking about presenting Map View on the screen, don't miss out on chapter 18, which covers the details on iOS's Map Kit framework.

With the combination of an accelerometer, GPS chip, and camera, you can build an augmented reality navigation application with iOS's hardware framework access. iOS frameworks for these hardware accesses can be found in this book.

UTILITY APPLICATIONS

The key to a successful utility application is to keep it simple. Make sure your application will focus on one major task and stick with it. The calculator application on iPhone is a good example. For a beginner, this is a relatively easy category. Moreover, it fits the needs for the creative idea or the niche market. With the knowledge acquired in the first seven chapters of this book, you'll be able to create a decent utility application.

Next, we'll cover the business model for the iOS platform.

1.6.3 Making money with your application

Generally speaking, there are three ways to generate profit on the iOS platform:

- *Submit a paid application.* You can price your application at the level you're happy with; the current price tier allowed on the App Store is from US$0.99 to US$999.99 (there are similar pricing tiers for each App Store market in local currencies). It should be mentioned that you have a 30% profit share with Apple.
- *Submit a free application with in-app purchase.* You can use Store Kit on iOS to generate profit through an in-app purchase. For details, please refer to chapter 19. You share 30% of all revenue through in-app purchases with Apple.
- *Submit a free application supported by advertisements.* This is a common business model for free applications. Follow the step-by-step instructions on iAd from chapter 20; you'll be able to make money with your application in no time.

You don't have to build an application for profit. Simply creating a cool application and learning a new programming technique is fun and rewarding in itself. The bottom line is if you're planning to distribute your application through the App Store, make sure you read and follow the application guidelines from the iOS Developer Center.

Finally, stay focused and don't give up! Learning a new programming language isn't easy. Try listing the key features for your application, and focus on the most important ones throughout the development process. Unless you have unlimited resources, it will be hard to put all the features you want inside one application. And even if you manage to, it may be too hard for users to figure out how to use your app. Remember: sometimes less is more.

1.7 Summary

In this chapter, we first explored iOS on the iPhone and iPad, and then we explained how to install the iOS SDK on your Mac. We also covered the anatomy of iOS, including objects, classes, and methods, providing the backdrop for coding in Objective-C, which follows in the next chapter. With the program environment ready, you can start the journey with iOS development.

iPHONE/iPAD DEVELOPMENT

iOS 4 IN ACTION

Harrington • Trebitowski • Allen • Appelcline

Version 4 of the iOS SDK adds powerful new features like multitasking, GCD, blocks, and iAds. With the release of Xcode 4, it's easier than ever to get programming, even if you're new to Objective-C.

iOS 4 in Action, written for Xcode 4, is a detailed, hands-on guide that goes from setting up your development environment, through your first coding steps, all the way to creating a polished, commercial iOS 4 application. You'll run through examples from a variety of areas including a chat client, a video game, an interactive map, and background audio. You'll also learn how the new iOS 4 features apply to your existing iOS 3 based apps. This book will help you become a confident, well-rounded iOS 4 developer.

What's Inside

- Full coverage of iOS SDK 4.3
- Mastering Xcode 4
- Multitasking for iPhone and iPad
- Game Center, iAd, and AirPrint
- Local and push notification
- Implementing in-app purchasing

No previous iPhone or iPad know-how needed. Familiarity with C, Cocoa , or Objective-C helps but is not required.

Jocelyn Harrington is an experienced full-time mobile developer with a dozen entries in the App Store. **Brandon Trebitowski, Christopher Allen,** and **Shannon Appelcline** are the authors of the previous edition, *iPhone and iPad in Action*.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/iOS4inAction

Free ebook
SEE INSERT

"The best iOS development book—and the first to feature Xcode 4!"
—Alex Curylo, Trollwerks Inc.

"All the information you need to write a large variety of iOS applications."
—Glenn Stokol
Oracle Corporation

"The ideal quick-start to programming Apple's mobile devices."
—Jonas Bandi, TechTalk

"A no-nonsense approach to writing apps for iOS 4."
—David Sinclair
Digital Innovators

"Take an Apple, byte, and start coding."
—Jeroen Benckhuijsen
Salves Development

ISBN 13: 978-1-61729-001-5
ISBN 10: 1-61729-001-7
54499
9 781617 290015

MANNING

$44.99 / Can $51.99 [INCLUDING eBOOK]