Chapter 1

Entering Mobile Application Development

In This Chapter

- Identifying the market
- Following the design process
- Entering the world of object-oriented development

Mobile devices are everywhere. These smartphones and tablets run powerful applications and are making a difference in how people live, work, and play.

Many folks already use these devices as they do computers: to create and edit documents; to interact with others via e-mail, telephone, and chat; to play highly entertaining games; and to shop and manage money. Even schools, which used to ban cellphones in the classroom, are considering delivering educational materials to students via smartphones. Because they're common and robust, tablets and smartphones are now the primary computing and communication devices for many people.

A mobile device, in particular a smartphone, is more than a computing and communication device, however. Because it goes everywhere with you, you can be constantly connected to work and with other users. Also, because a smartphone can retain information about people you talk to, where you've been, and how much you spend, it in a sense "knows" you intimately. Mobile applications can take advantage of this device-user relationship to provide personalized and targeted services that users will depend upon and love.

Apps for a Mobile Platform

This book assumes that you've written applications for other platforms, such as desktop or laptop computers or the web. You can transfer a lot of this experience to writing applications for mobile devices like cellphones and tablets, including iOS devices.

However, when writing applications for iOS, you need to consider these differences:

- **Tiny keyboards:** iOS device keyboards make data entry *very* difficult. Data entry is no easy task to begin with, and *touchscreen* virtual keyboards, which you press with your thumbs, are prone to data-entry errors (for example, your app should provide smart spell-checking or allow the user to simply select from a set of options rather than making him type text).

  Some applications are created primarily to enter data (think Twitter or e-mail apps). However, try to limit data entry by doing things such as prefilling commonly used default values and providing drop-down lists that users can select from.

- **Small display area:** Displays on iOS devices come in these three shapes and sizes (see Figure 1-1):
  - 4-inch iPhone and iPod Touch
  - 7.9-inch iPad mini
  - 9.7-inch iPad

Figure 1-1: Here are the three iOS device sizes.

Compare these sizes to laptop screens, which are usually 15 inches or larger, and you'll see what I mean by limited screen space.

In order to be usable on small screens, an application must be designed so as to allow users to

- Move intuitively in the program (without getting confused by a maze of screens).
- Use controls (buttons, for example) that are large enough to press easily and place them in a way that helps to prevent click errors.

- **Universal applications needed:** In order for an iOS application to be popular, it must run on a range of devices with varied capabilities — that is, the iPhone, the iPad mini, and the 9.7-inch iPad (refer to Figure 1-1).

  Applications need to function well on the smallest and largest iOS displays.

  Note that previous generations of iOS devices had even smaller screens (iPhones prior to iOS 5 and iPod Touches prior to the 5th generation all had 3.5-inch displays). Also, Apple TV runs iOS. If Apple opens these platforms for app development with the latest iOS versions, the problem of creating universal apps will become even more complicated.

- **Limited storage:** iOS devices can store only about one-tenth of the information that PCs can, in both memory and persistent storage (flash or disk).

  Don't store too many images, music, or (especially) video on the device because it can run out of space pretty darn quickly.

- **Unreliable networks:** It's a fact of life: Mobile devices periodically lose network connectivity. Even when a device has a stable connection, the amount of data that can be sent or received varies based on the strength of the connection. So make your app

  - Buffer incoming data when the network connectivity is good.
  - Save outgoing data locally.
  - Receive and transmit data on a separate background thread.

- **Device unavailability:** A mobile device can be turned on and off depending on a user's situation (for example, when boarding a plane). A device can also be damaged (say, by being dropped), its computing speed can slowly degrade, and it can even shut down as its battery is consumed.

  Your application must deal with all these situations. For example, it could periodically check-point its state and have low-power modes of operation (for instance, a video-playing app might switch to playing only audio when the battery is low).

- **A range of uses:** Mobile devices are used in a variety of locations: rooms with low ambient lighting or sports stadiums with high levels of background noise, for example.

Your applications must be able to adapt to these types of situations. For example, your app may lower the brightness of the screen when the ambient light is low or increase its audio volume when background noise is high.

- **Coding in Objective-C:** Apple made an early and highly innovative decision to base its development platform on Objective-C, well before standard object-oriented (sometimes referred to as OO) programming languages (such as Java, C++, and C#) came on the scene. Objective-C has an unusual syntax (as I explain in Chapter 3). It also has object-oriented semantics that are more like the early object-oriented languages like Smalltalk, but it's different from the later and now standard object-oriented languages like C++ and Java that most programmers are used to.

  Apple has provided a robust, highly reliable framework and excellent documentation to help build up strong skills in iOS app development.

iOS Benefits

Although many types of smartphones and mobile devices are still on the market today, the battle for market share is now pretty much between iOS and Android.

The lure of Apple and its wonderful set of innovative devices are what make the iOS platform so popular, and developing on the iOS platform offers you several benefits:

- **Wide acceptance:** iOS has legs — it's inside millions of devices and is a major platform for application developers. So your app has a readymade market.
- **Powerful, built-in, reusable capabilities:** The iOS framework has lots of existing capabilities and services. It has built-in support for rich graphics, location finding, and data handling. In other words, you don't have to write all the code for your application from scratch.
- **Framework-based guidance for developers:** Because iOS is a framework — not just a toolkit composed of a set of libraries — it imposes a structure on applications by using an application model. In return for this imposition, you receive a lot of benefits. You get to follow a systematic path in designing a robust application, which frees you to focus on providing rich capability rather than on figuring out the application's structure and high-level design or on nonfunctional tasks, such as managing your application's lifecycle. (You know what I mean — the starting-it-up stuff and the restoring-its-state-after-shutdown stuff, for example.)

Doing the sample application thing

This book uses a simple Tic-Tac-Toe game as an example. Each player claims a symbol, usually an O or an X. Players alternately place their symbol in empty locations on a 3 x 3 grid, with the goal of placing the same symbol in three grid spaces in a straight line, either in a horizontal row, a vertical column, or on a diagonal. The figure shows a sample sequence of plays. This Tic-Tac-Toe application allows two players to play against each other or for one player to play against the device.

You want the application to offer the following game-related functionality (in these examples, a user is playing against a computer):

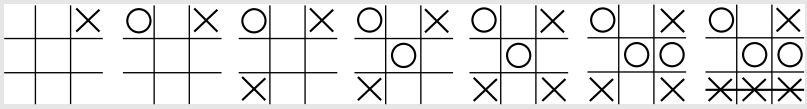
- Allow the user to create a profile, consisting of a playing name and who goes first in the game — the user or the computer (see Chapter 6).
- Allow the user to start and play the game (see Chapter 7).
- Allow the user to exit the game at any time (see Chapter 10).
- Identify when the game progresses to a draw, a victory for the user, or a victory for the computer, and show the results (see Chapter 7).
- Record and save the results of a completed game (see Chapter 6).

In addition to the basic gameplay features, an application intended for the Android market needs to be robust: *reliable and secure*. Here I show you how to give the app these additional benefits (for more on these topics, see Chapter 11):

- Make the user's game data private by creating player accounts.
- Keep a history of game play by having the program log to a file.
- Make the game crash-resistant so that it retains its preferences after a forced shutdown.

The Tic-Tac-Toe game also illustrates how to use iOS built-in capabilities with features such as these:

- Invoking external services — such as location services (see Chapter 12)
- Sending the results of a game by e-mail to an address book contact (see Chapter 13)
- Playing music from an audio file and recording music from the built-in microphone (see Chapter 13)



iOS App Development Essentials

Writing an application program would require a lot of work if you had only the device to work with. The good news is that the iOS framework uses a piece of software known as the *operating system (OS),* which provides device-independent interfaces to everything on the device (such as the file system, sensors, and so on). The OS also provides a standard interface to computing capabilities (such as for starting and stopping programs).

As a result, operating systems make writing and running applications easier, and they're especially helpful — in fact, essential — on mobile devices. Apple developed and owns iOS, the operating system for its mobile products. Originally called the iPhone OS, iOS was unveiled in 2007 for the iPhone and was later extended to support the other Apple devices, as well as the Apple TV device.

Unlike, say, Linux, which powers Android, iOS is a single-user operating system. That said, this and other limitations are artificial. At its core, iOS can do nearly everything that Apple's desktop operating system (OS X) can. For a variety of reasons, including secrecy and a genuine desire for tight quality control, Apple closely guards iOS, and only developers with special privileges are given access to its internals.

Devices

Every iOS device (like every mobile device) is a computer, composed of a set of hardware components: processor, memory, input/output (I/O) devices (such as a keyboard, touchpad, and screen), and storage (discs and flash, for example).

Unlike Android devices, the hardware configuration is controlled completely by Apple, so there are just four main variations of devices to consider when developing iOS apps:

- iPhone (of course)
- iPod Touch
- iPad
- iPad mini

Like other smart devices, iOS devices also come with several built-in hardware components, such as the following:

- Cameras (front and back facing)
- Audio inputs and outputs

- GPS
- Accelerometer
- Light sensor

Apple has yet to come out with a near-field communications-enabled device (or NFC-enabled device) but was recently awarded a patent for NFC-enabled data synching technology. For the inside story from Apple, check the link labeled NFC at `www.dummies.com/go/iosprogramminglinks`.

Unless you really and truly want to, you'll never see iOS, the operating system, nor will your program. However, you must recognize that it's there — the iOS framework does certain things in certain ways because it runs on iOS. For example, every running program is assigned a process. When an iOS app starts, an iOS process becomes active. This process takes over an area of the screen on the device and allows the user to interact with the application. If another application starts, it pushes the first application to the background. At this point, the process assigned to the first application may be (arbitrarily) terminated by the operating system to save device resources. Before this happens, the iOS runtime notifies the application to save its state.

This iOS operating system is the OS that manages the device *on which your apps run*. A different operating system manages the personal computer on which you develop apps (the Macintosh OS or OS X).

Application development technologies

Layered on the operating system are the iOS application development technologies. These are the technologies that you'll use to build iOS apps. These technologies are structured as a set of layers, as shown in Figure 1-2.

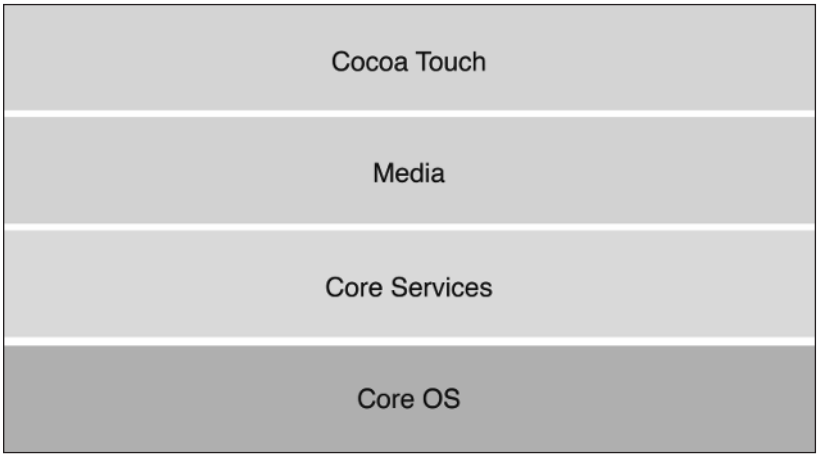


Figure 1-2: Architecture of the iOS Technologies.

Unix ho!

If you're a Unix lover, you'll be pleased to see Core OS reveal its Unix roots.

For example, Core OS includes many of the typical libraries found on Unix systems (in the `/usr/lib` directory of the system, with header files in the `/usr/include` directory).

Dynamic shared libraries are identified by their `.dylib` extension.

I start with the bottom layer so that you see how the technologies are built from the hardware up.

Each layer exposes a set of components that Apple calls *frameworks*. As I describe each layer, I'll list and briefly describe each layer's capabilities. For Apple's introduction to these layers, check out the link labeled iOS Frameworks at `www.dummies.com/go/iosprogramminglinks`.

Core OS layer

The Core OS layer contains the operating system and the services upon which the other technologies are built. These services include the following:

- Image and digital signal processing
- Linear algebra (the math of matrix operations — primarily used for vector drawing)
- Bluetooth access
- Third-party device connections by serial port
- Generic security services
- System and networking services

You won't often use Core OS directly in your applications, except when you need to deal with communication or security capabilities at the operating system level or control an external hardware accessory (like a device connected to a serial port). However, you will use its functionality via the other layers.

For more information on Core OS, check the link labeled CoreOS Layer at `www.dummies.com/go/iosprogramminglinks`.

Core Services layer

The Core Services layer provides access to several more system services that most applications use. These services include

- **iCloud:** iCloud is a cloud-based storage service that gives you iOS devices to share documents and applications and to share small bits of data (such as preferences) across your multiple iOS devices.
- **Automatic reference counting (ARC):** ARC is the name of the new Objective-C compiler as well as a runtime feature that enables memory management within your program without you having to explicitly free memory. ARC automatically keeps track of all references to an object and then deletes the object when no references point to it. If you're a Java programmer, you'll recognize that ARC is essentially the iOS version of automatic memory management and garbage collections.

  Apple's development environment (Xcode) provides tools that help you migrate from an older application that doesn't use ARC to one that does.
- **Block objects:** Block objects are inline code along with associated data that's treated as a function.

  Block objects are particularly useful as callbacks — such as to user interface events or to thread events.
- **Data protection:** This is the capability to encrypt, lock, and unlock files that an application needs to keep secret.
- **File-sharing support:** This enables applications to share files via iTunes (version 9.0 and higher).
- **Grand Central Dispatch:** This is a concurrency-enabling mechanism that enables programmers to define concurrent tasks, rather than create threads directly, and then lets the system perform the tasks.
- **In-App Purchase:** This is the ability to purchase from vendors such as iTunes directly from an app. In-App Purchase is implemented by a framework known as the Store Kit.
- **Core Data:** Core Data is a framework for managing the lifecycle of persistent objects. Core Data works well with SQLite, which is probably the most widely used database on mobile devices. Core Data and its use of SQLite are discussed in Chapter 6..
- **JSON support:** This service provides support for parsing and creating JSON documents. You find more on this topic in Chapter 6.

The Core Services layer also provides a collection of frameworks for the following:

- Managing the address book
- Supporting ads
- Providing high-performance access to networks
- Manipulating strings, bundles, and raw blocks
- Making use of location, media, motion, and telephony
- Managing documents
- Downloading newsstand content
- Managing coupons and passes
- Presenting thumbnail views of files
- Accessing social media accounts
- Purchasing from the iTunes store
- Programmatically determining the network configuration and access of a device

The Core Services layer provides the object-oriented Foundation framework that does the following:

- Defines the basic behavior of object.
- Provides management mechanisms.
- Provides object-oriented ways of handling primitive data types, such as integers, strings and floating-point numbers, collections, and operating-system services.

The Cocoa Touch framework (see the section, "Cocoa Touch layer," later in this chapter) and the Foundation framework make up the two key iOS development components used by developers. Use all the other components on an as-needed basis.

For more information on Core Services, check the link labeled Core Services in `www.dummies.com/go/iosprogramminglinks`.

Media layer

The Media layer contains support for graphics, audio, and video technologies. This layer has the following components:

- **Core Graphics (also known as Quartz):** Natively handles 2D vector- and image-based rendering.
- **Core Animation:** Provides support for animating views and other content. This is also a part of Quartz.

- **Core Image:** Provides support for manipulating video and still images.
- **OpenGL ES and GLKit components:** Provide support for 2D and 3D rendering using hardware-accelerated interfaces.
- **Core Text:** Provides a text layout and rendering engine.
- **Image I/O:** Provides interfaces for reading and writing most image formats.
- **Assets Library:** Provides access to the photos and videos in the user's photo library.

This layer also allows you to manage images, audio, video, and audio and video assets (music and movie files, and so on), along with their metadata. A MIDI interface is provided for connection with musical instruments.

Integrated record and playback of audio is provided as follows:

- Through a media player that allows you to manipulate iTunes playlists
- Via lower-level components for
  - Managing audio playback and recording
  - Managing positional audio playback (such as surround sound)
  - Playing system alert sounds
  - Vibrating a device
  - Buffering streamed audio content
  - Airplay streaming

Video services provided include playing movie files from your application or streaming them from the network and capturing video and incorporating it into your application. Once again, this functionality is provided in several ways: from a high-level media player to lower-level components that give you fine-grained control.

Image handling operations include creation, display and storage of pictures, and filters and feature detection.

Also, this layer is the one that provides support for text and font handling — such as layout and rendering.

For more information on the Media layer, check the link labeled Media Layer at `www.dummies.com/go/iosprogramminglinks`.

Cocoa Touch layer

The Cocoa Touch layer contains most of the object-oriented developer-facing frameworks for building iOS applications. It's your single point of entry to app development.

The Apple guides encourage you to investigate the technologies in this layer to see whether they meet your needs, before looking at the other layers. In other words, Apple intends for Cocoa Touch to be your single point of entry into iOS app development.

Cocoa Touch is where you build your app's user interface, handle touch-based and gesture-based interactions, connect the user interface to the app's data, deal with multitasking, and integrate everything from state preservation to push notification to printing.

Cocoa Touch provides object-oriented access for managing your address book and events, building games, and dealing with ads, maps, messages, social media, and sensors. So, most of the time, you'll work through Cocoa Touch; it gives you access to the other layers of the technology. In particular, you'll work with the UIKit framework, which packages most of the functionality just described.

At times, you may need direct access to the lower layers. Although showing you how to achieve this kind of direct access isn't the focus of this book, I cover such access in the appropriate chapters in the book.

For a complete list of the iOS frameworks, check the link labeled iOS Frameworks at `www.dummies.com/go/iosprogramminglinks`.

Xcode

Xcode is two things. It's the kernel (the *engine* according to Apple) of Apple's integrated development environment (IDE) for OS X and iOS. It's also the name of the IDE application itself.

With Xcode, you can do the following:

- Create and manage projects.
- Manage project dependencies, such as specifying platforms, target requirements, dependencies, and building configurations.
- Build the app from the project.
- Write source code using intelligent editors that auto-check syntax and automatically format your code.
- Navigate and search through a project, program files, and developer documentation.
- Debug the app in an iOS Simulator, or on the device.
- Analyze the performance of your app.

Figure 1-3 shows the Xcode startup screen.

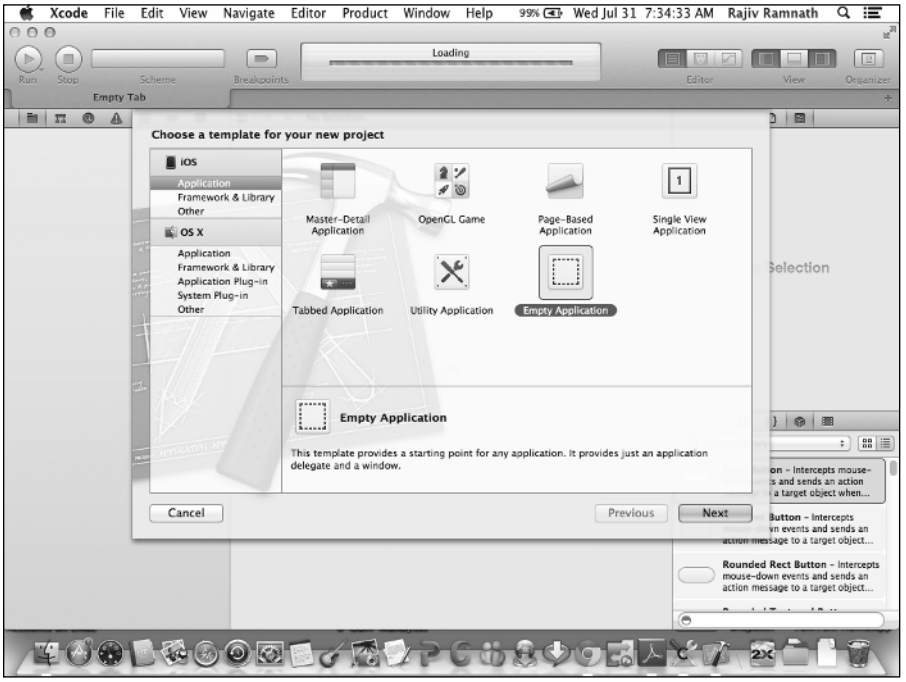


Figure 1-3: The Xcode IDE.

If you've used another IDE, such as Eclipse, NetBeans, or BlueJ, you'll find Xcode easy to use.

The Application Model

To begin with, note that the operating system on your iOS device starts a set of system programs when the device boots. This set of programs, which you can think of as the iOS runtime system, runs constantly in the background and manages every app that is run.

Technically, your app is nothing more than an executable program (like an `.exe` on Windows) that runs on the device and interacts with the iOS runtime system. The home screen on the iOS device simply shows icons for all such executable programs. When an icon is clicked, the operating system launches the executable corresponding to the icon and causes the program to run on the iOS device.

In other words, an iOS app is just a program that runs on the device — a pretty straightforward beast.

An Android app, on the other hand, consists of a set of Java classes that are loaded by and encapsulated inside the Android runtime system. This Android runtime system is a Java program that runs on the Java virtual machine.

When the app is built, it's linked with a standard main program along with an app-specific component generated by the Xcode IDE known as the *app delegate*. The main program and the app delegate together serve as the interface between your app and the iOS runtime. These components deal with user interface events, such as touches, and system events such when your app goes into the background — for example, because of a user's action or maybe an e-mail comes in (for more on this topic, see Chapter 6).

Understanding the lifecycle of an iOS app

An iOS app follows a typical lifecycle (see Figure 1-4). At the beginning, the app is simply an executable; it's not running, lying patiently in wait for a user to click its icon. When the app starts, it goes through numerous initialization steps. During this transitory period, the app is in the inactive state. The app *is* indeed running (and in the foreground) but will not receive events, so it can't interact with anything during this time. The app then transitions to the active state. Now, the app is making merry, and you and the app are making sweet music together. This active state is the app's useful state.

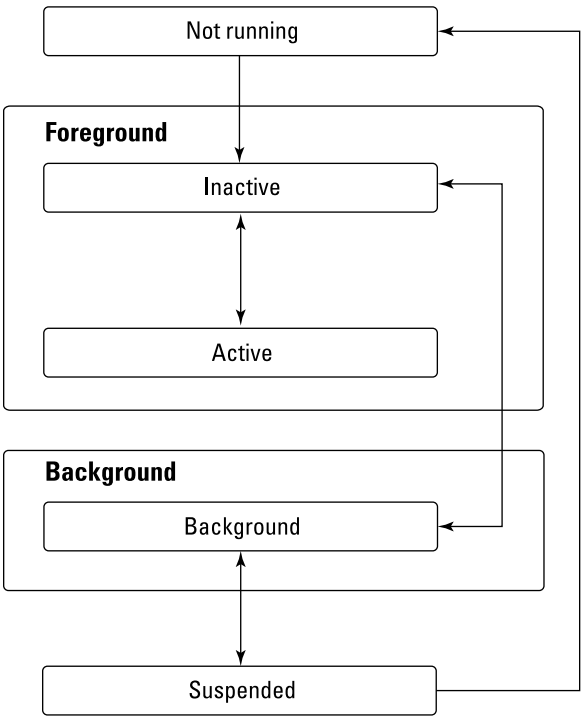


Figure 1-4: The lifecycle of an iOS app.

At some point — mostly when another app starts, say, a phone that's triggered by an incoming call — the iOS runtime will put your app in the background. At this point, the app is in the background state. Most apps stay in this state for a short time before being suspended. However, an app could request extra time to complete some processing (such as saving its state into a file for use the next time it starts). In addition, an app meant to run in the background will enter and stay in this state. Note that apps in the background can and do receive events, even though they don't have a visible user interface.

An app in the suspended state isn't running code; however, it is using power and the processor. The system moves an app to this state whenever it needs to further conserve resources, and does so without notifying the app. If memory runs low, the system may purge the app to create more space.

As the app transitions through its states, specific methods of the app (that is, code that you wrote) are invoked as explained here (and shown in Figure 1-5).

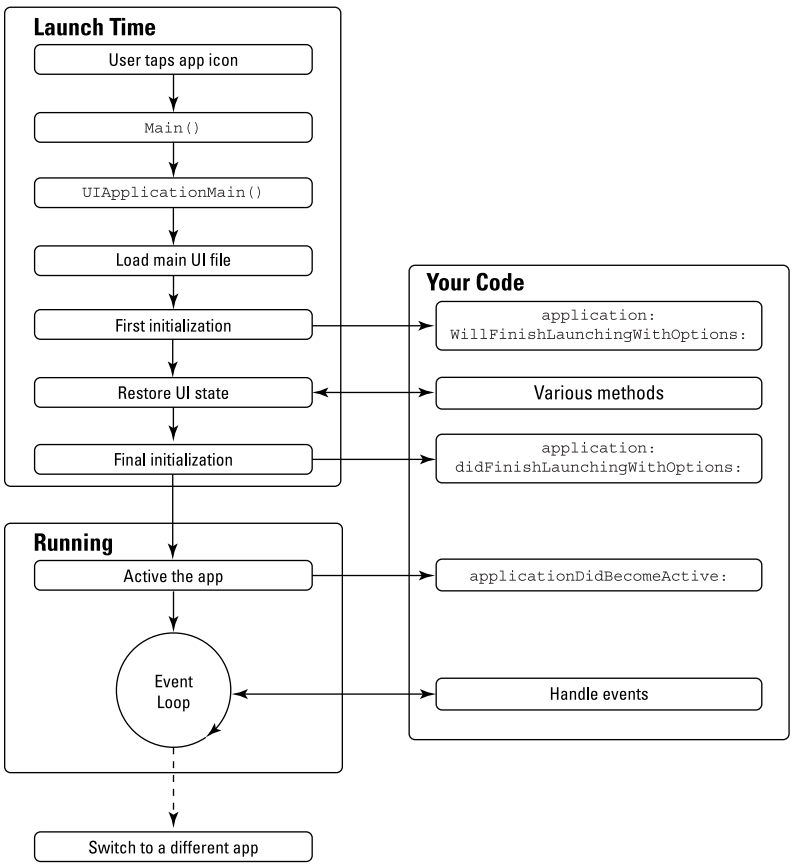


Figure 1-5: How the iOS runtime interacts with an app's lifecycle.

1. After the first initialization of the app, `appDidFinishLaunchingWithOptions` is called, which in turn invokes the portion of the app's code that sets up its user interface.

   The user then sees the app. The app now sits in an event loop, where it waits for user interactions.

2. When a user interacts with the app, an event is triggered, and a callback method tied to the event is invoked. Most often, the callback method consists of code written by the app's developer, although it could be reusable code provided as part of the iOS framework.

3. Once the callback method is done, the app goes back to its event loop. This sequence of actions (of events triggering callback methods) proceeds until the app receives an event that causes it to either shut down or go into the background state.

Understanding the structure of an iOS app

Every iOS app follows a standard structure known as a Model-View-Controller (MVC) pattern. (I begin discussing patterns in Chapter 2 and expand on patterns in Chapter 4.) For now, it's enough to know that a pattern is a standard way of writing software for a particular goal.

Specifically, the Model-View-Controller pattern splits the code of an app into

- The data it manages (known as the Model)
- The user-interface elements (known as the View)
- The Controller, which is the component that sits in between the Model and the View (or views) and translates user actions into updates to the Model and the View

You can see this structure in Figure 1-6. The dashed lines indicate linkage. Therefore, the model is linked to the view, and the views are linked to the controller. The solid lines indicate actions. So, the view updates portions of the model while the controller updates the views (or more correctly, causes the views to update themselves). The controller also updates models as needed. iOS extends this pattern so that each app is really a hierarchy of controllers, each managing a set of views and potentially a model.

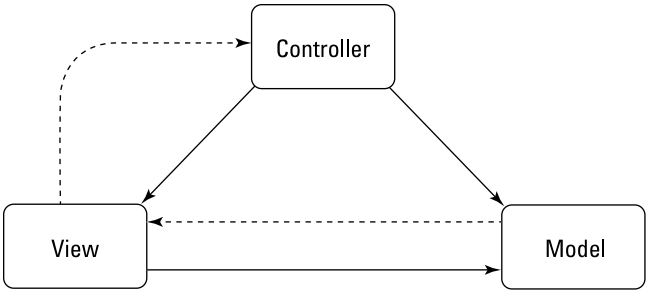


Figure 1-6: The Model-View-Controller design pattern.

Object-Orientation Concepts

Object-orientation applies to iOS development a couple of ways:

- iOS apps are (mostly) written in Objective-C, an object-oriented programming language that implements object-oriented concepts.
- iOS apps are built around a core design pattern known as the *MVC design pattern* and follow several other design patterns as well.

  Design patterns are nothing more than standard templates for designing the classes and objects that make up your system. In other words, design patterns are higher-level concepts built on object-oriented building blocks.

This book guides you through iOS from an object-oriented perspective:

- Chapter 2 explains in depth what object-orientation means, its basic building blocks, and the higher-level concepts of patterns and frameworks.
- Chapter 3 introduces you to Objective-C.
- Chapter 6 takes you deep into object-oriented development using the patterns in the iOS framework. Chapter 6 also deconstructs the iOS framework in object-oriented terms.
- A complete example of object-oriented software development of an iOS app is worked out in Chapter 7.
- Other chapters, which focus on the extensive capabilities of iOS, are presented in object-oriented terms.