The App Life Cycle

Apps are a sophisticated interplay between your custom code and the system frameworks. The system frameworks provide the basic infrastructure that all apps need to run, and you provide the code required to customize that infrastructure and give the app the look and feel you want. To do that effectively, it helps to understand a little bit about the iOS infrastructure and how it works.

iOS frameworks rely on design patterns such as model-view-controller and delegation in their implementation. Understanding those design patterns is crucial to the successful creation of an app. It also helps to be familiar with the Objective-C language and its features. If you are new to iOS programming, read *Start Developing iOS Apps (Swift)* for an introduction to iOS apps and the Objective-C language.

The Main Function

The entry point for every C-based app is the `main` function and iOS apps are no different. What is different is that for iOS apps you do not write the `main` function yourself. Instead, Xcode creates this function as part of your basic project. Listing 2-1 shows an example of this function. With few exceptions, you should never change the implementation of the `main` function that Xcode provides.

Listing 2-1 The `main` function of an iOS app

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


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

The only thing to mention about the `main` function is that its job is to hand control off to the UIKit framework. The `UIApplicationMain` function handles this process by creating the core objects of your app, loading your app's user interface from the available storyboard files, calling your custom code so that you have a chance to do some initial setup, and putting the app's run loop in motion. The only pieces that you have to provide are the storyboard files and the custom initialization code.

The Structure of an App

During startup, the `UIApplicationMain` function sets up several key objects and starts the app running. At the heart of every iOS app is the `UIApplication` object, whose job is to facilitate the interactions between the system and other objects in the app. Figure 2-1 shows the objects commonly found in most apps, while Table 2-1 lists the roles each of those objects plays. The first thing to notice is that iOS apps use a model-view-controller architecture. This pattern separates the app's data and business logic from the visual presentation of that data. This architecture is crucial to creating apps that can run on different devices with different screen sizes.

Figure 2-1 Key objects in an iOS app

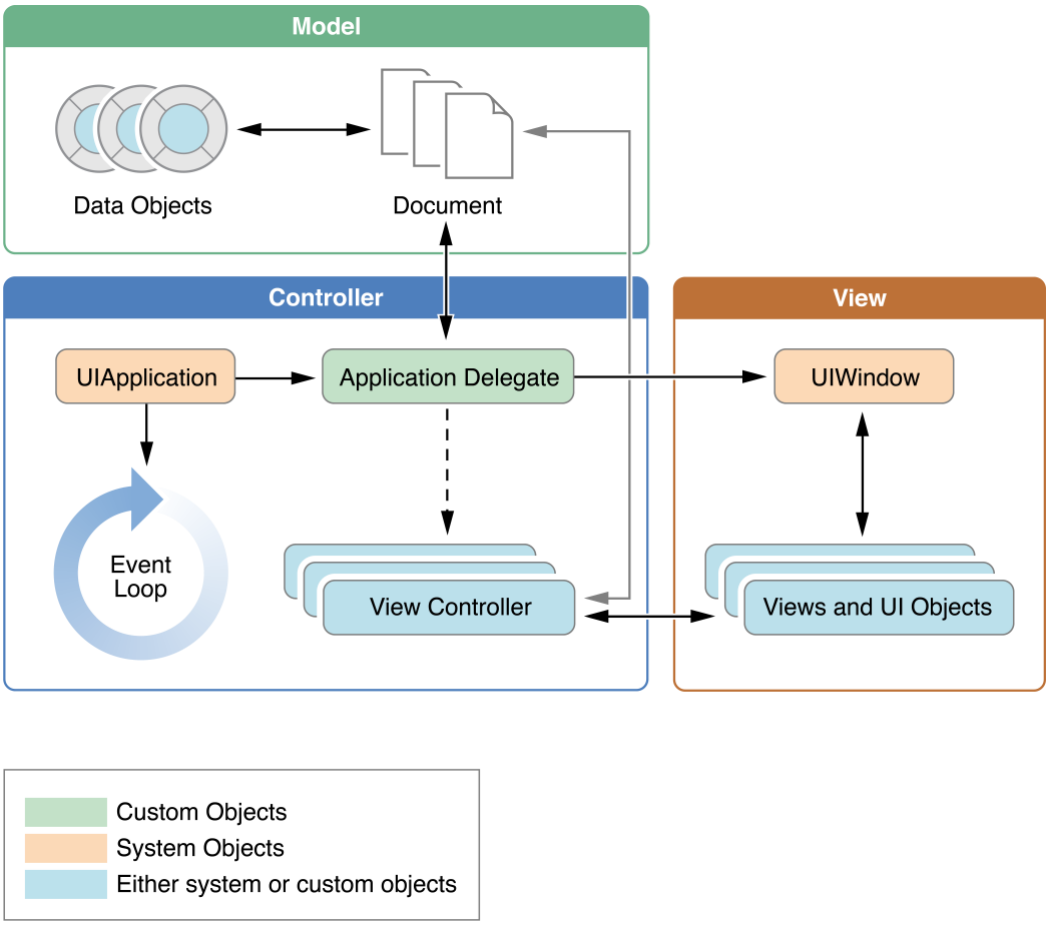


Table 2-1 The role of objects in an iOS app

Object	Description
`UIApplication` object	The `UIApplication` object manages the event loop and other high-level app behaviors. It also reports key app transitions and some special events (such as incoming push notifications) to its delegate, which is a custom object you define. Use the `UIApplication` object as is—that is, without subclassing.
App delegate object	The app delegate is the heart of your custom code. This object works in tandem with the `UIApplication` object to handle app initialization, state transitions, and many high-level app events. This object is also the only one guaranteed to be present in every app, so it is often used to set up the app's initial data structures.
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 *Document-Based App Programming Guide for iOS*.
	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

View controller objects	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*.
`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 objects, control objects, 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.

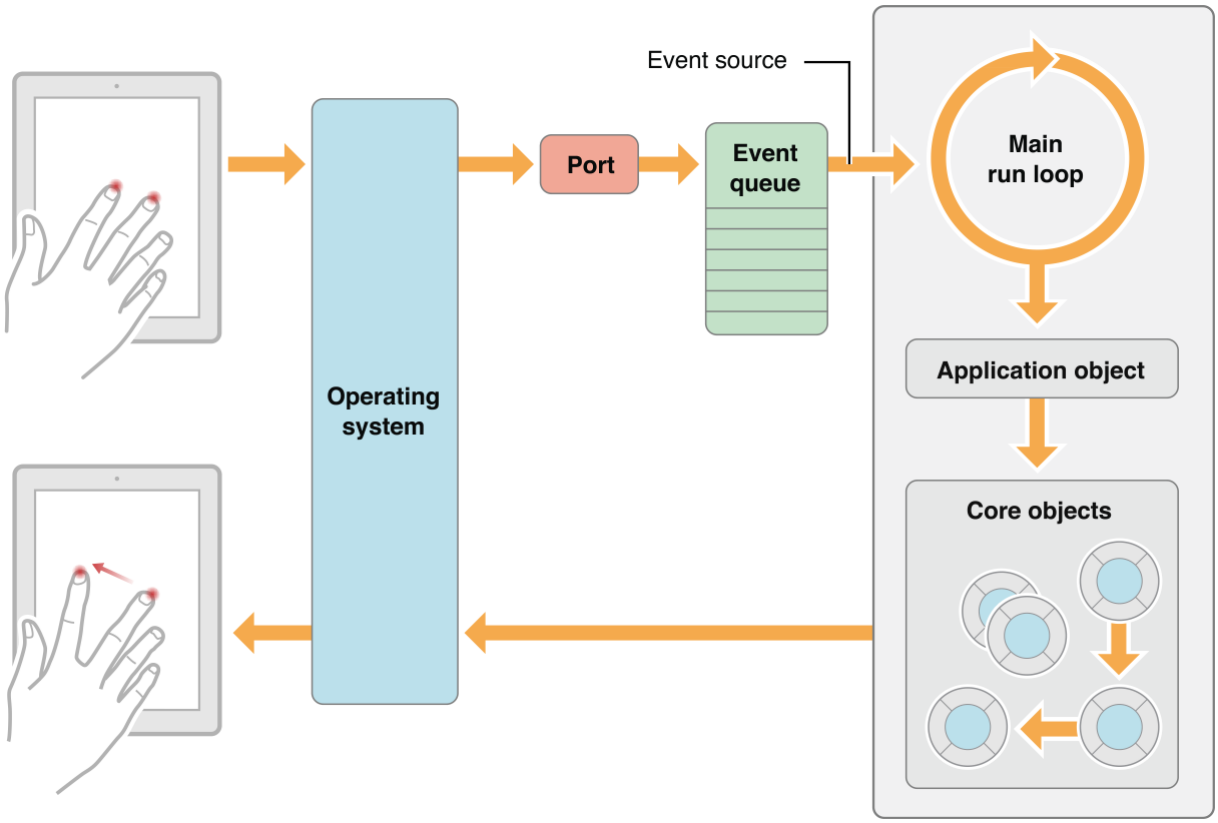
What distinguishes one iOS app from another is the data it manages (and the corresponding business logic) and how it presents that data to the user. Most interactions with UIKit objects do not define your app but help you to refine its behavior. For example, the methods of your app delegate let you know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

The Main Run Loop

An app's *main run loop* processes 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 2-2 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 2-2 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 2-2. Many of these event types are delivered using the main run loop of your app, but some are not. Some events are sent to a delegate object or are passed to a block that you provide. 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 2-2 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 Shake motion events	First responder object	Remote control events are for controlling media playback and are generated by headphones and other accessories.
Accelerometer Magnetometer Gyroscope	The object you designate	Events related to the accelerometer, magnetometer, and gyroscope hardware are delivered to the object you designate.
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 and Maps Programming Guide*.
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*.

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.

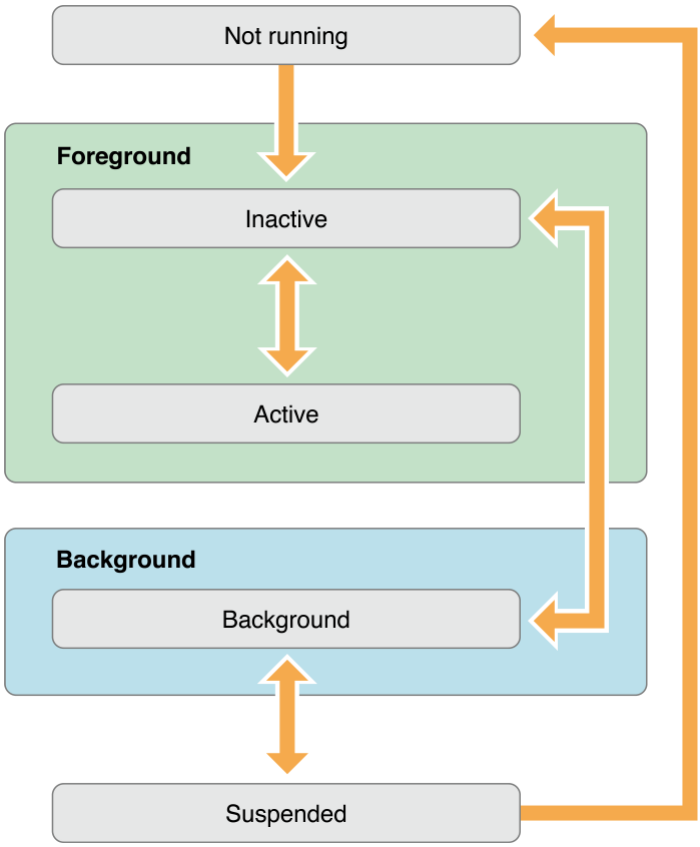
Execution States for Apps

At any given moment, your app is in one of the states listed in Table 2-3. 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 2-3 shows the paths that an app takes when moving from state to state.

Table 2-3 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.
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 2-3 State changes in an iOS app



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.

App Termination

Apps must be prepared for termination to happen at any time and should not wait to save user data or perform other critical tasks. System-initiated termination is a normal part of an app's life cycle. The system usually terminates apps so that it can reclaim memory and make room for other apps being launched by the user, but the system may also terminate apps that are misbehaving or not responding to events in a timely manner.

Suspended apps receive no notification when they are terminated; the system kills the process and reclaims the corresponding memory. If an app is currently running in the background and not suspended, the system calls the `applicationWillTerminate:` of its app delegate prior to

termination. The system does not call this method when the device reboots.

In addition to the system terminating your app, the user can terminate your app explicitly using the multitasking UI. User-initiated termination has the same effect as terminating a suspended app. The app's process is killed and no notification is sent to the app.

Threads and Concurrency

The system creates your app's main thread and you can create additional threads, as needed, to perform other tasks. For iOS apps, the preferred technique is to use Grand Central Dispatch (GCD), operation objects, and other asynchronous programming interfaces rather than creating and managing threads yourself. Technologies such as GCD let you define the work you want to do and the order you want to do it in, but let the system decide how best to execute that work on the available CPUs. Letting the system handle the thread management simplifies the code you must write, makes it easier to ensure the correctness of that code, and offers better overall performance.

When thinking about threads and concurrency, consider the following:

- Work involving views, Core Animation, and many other UIKit classes usually must occur on the app's main thread. There are some exceptions to this rule—for example, image-based manipulations can often occur on background threads—but when in doubt, assume that work needs to happen on the main thread.
- Lengthy tasks (or potentially length tasks) should always be performed on a background thread. Any tasks involving network access, file access, or large amounts of data processing should all be performed asynchronously using GCD or operation objects.
- At launch time, move tasks off the main thread whenever possible. At launch time, your app should use the available time to set up its user interface as quickly as possible. Only tasks that contribute to setting up the user interface should be performed on the main thread. All other tasks should be executed asynchronously, with the results displayed to the user as soon as they are ready.

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