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.

App은 여러분이 짠 코드와 시스템 프레임워크간에 정교한 상호작용을 통하여 구성이 됩니다. 시스템 프레임워크는 app이 구동되는 데에 필요한 기본적인 인프라를 제공해 주고, 여러분은 자신이 원하는 모양과 느낌을 앱으로 표현하는 데에 필요한 인프라를 커스터마이징하여 코드를 구현하게 될 것입니다. 이러한 일을 효과적으로 하기 위해, iOS의 인프라와 어떻게 작동하는 지를 이해하는 것이 매우 중요합니다.

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.

iOS 프레임워크는 MVC(Model - View - Controller), 델리게이션과 같은 디자인 패턴 등을 사용합니다. 이 디자인 패턴을 이해하는 것은 여러분이 앱을 만드는 데에 있어 매우 필수적입니다. 또한 Objective-C와 그 기능에 대해서 익숙해지는 데에 도움을 줄 것입니다.

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.

모든 C가 기본으로 된 App에서는 `main`이 시작지점이고 iOS App또한 같습니다. 다른 점이 있다면 iOS App들은 `main`를 만들 필요가 없다는 것 입니다. 대신 Xcode가 여러분의 프로젝트에 이 기능을 자동적으로 만들어 줍니다. 아래의 예제 코드를 보세요. 몇몇 특수한 경우를 제외한다면 여러분은 Xcode가 제공하는 `main`기능을 바꾸지 말아야 할 것입니다.

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 storyboardfiles, 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.

`main`이 하는 유일한 기능은 UIKit 프레임워크에 제어 권한을 넘겨주는 일 입니다. `UIApplicationMain`은 여러분 앱의 코어 객체를 생성하고, 사용가능한 파일로부터 인터페이스를 로딩하고, 여러분이 초기 설정을 할 수 있도록 만들어진 코드를 호출하고, 앱의 run loop를 동작시켜 제어 권한을 넘기는 일을 수행합니다. 여러분이 해야할 유일한 것은 스토리보드 파일과 여러분이 커스텀한 초기화 코드를 제공하는 것 뿐입니다.

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.

`UIApplicationMain`은 여러 필수 객체들을 설정하고 앱을 구동시킵니다. 모든 iOS App의 필수요소는 시스템과 객체들 간의 상호작용을 담당하는 `UIApplication`객체입니다. 아래의 예제(Figure 2-1)는 대부분의 앱에서 볼 수 있는 객체들과 그 객체들이 하는 역할에 대해서 설명해 놓았습니다. 첫번 째로 알아야 할 것은 iOS 앱은 MVC 디자인패턴 구조를 사용한다는 것입니다. 이 패턴은 앱의 데이터와 로직을 View와 같은 화면에 보여지는 것과 분리를 시킨 구조입니다. 이 구조는 각기 다른 사이즈를 가진 장치 (아이폰, 아이패드 등)들에서 사용할 수 있도록 앱을 만드는 데에 필수적입니다.

Figure 2-1 Key objects in an iOS app

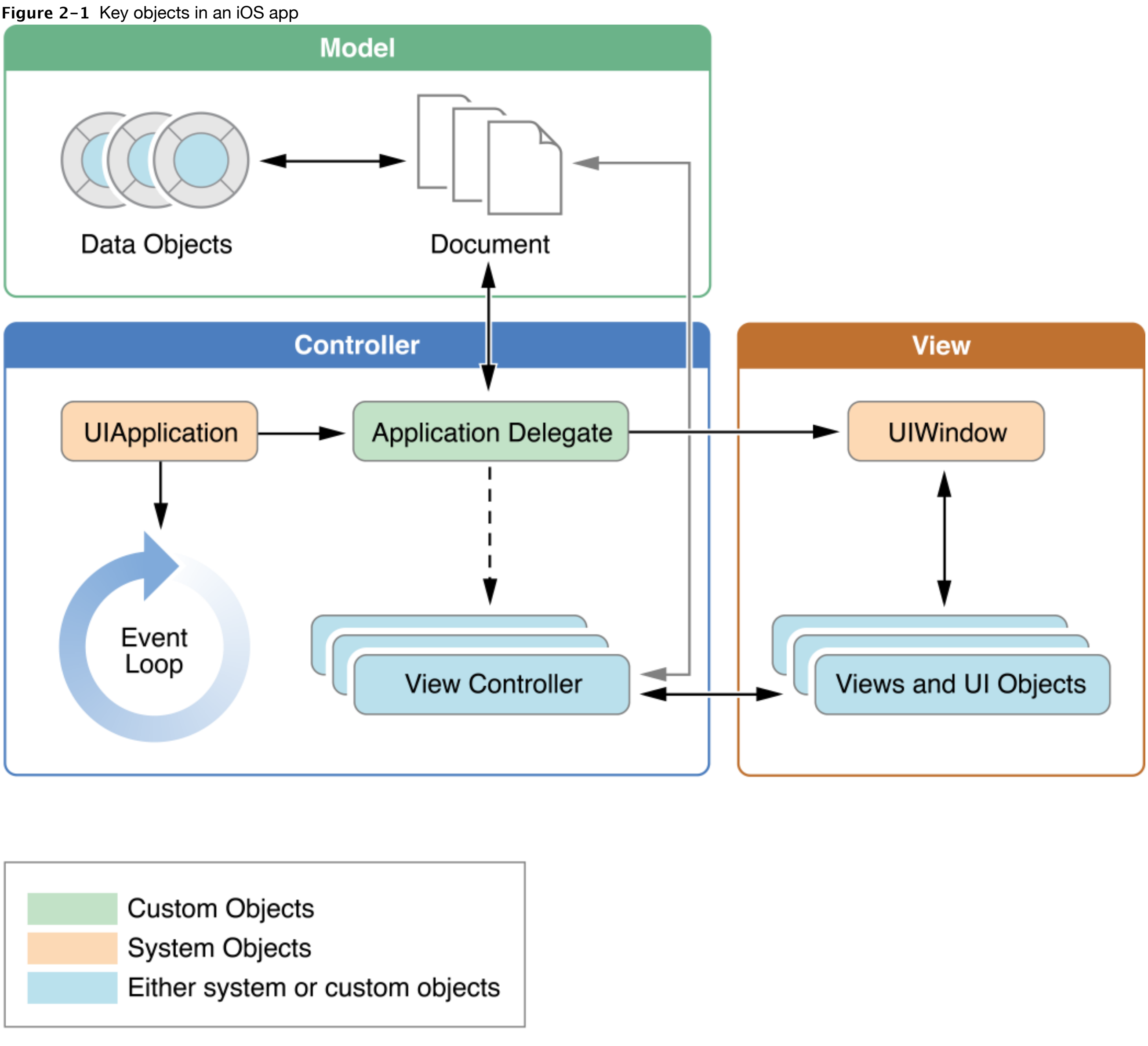


Table 2-1 The role of objects in an iOS app

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. `UIApplication`객체는 이벤트 루프와 다른 high-level 앱의 기능을 관리합니다. 또한 앱의 상태변화(background, foreground등)와 몇몇 특별한 이벤트(예를들어 푸시 알람)를 여러분이 정의한 객체인 앱델리게이트에 전달하게 됩니다. `UIApplication`객체는 서브클래싱(상속)하여 사용하지 말고 `UIApplication`객체 그대로 사용하셔야 합니다.
App delegateobject	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. 앱 델리게이트는 여러분이 작성한 코드에 중요한 역할을 합니다. 이 객체는 `UIApplication`객체와 함께 앱의 초기화, 상태변화, 그리고 많은 hige-level의 이벤트를 처리하게 됩니다. 또한 이 객체는 모든 앱에 오직 한개만 존재하기 때문에 앱의 초기 데이터 구조를 정의하는데에 많이 사용됩니다.
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.) 데이터 모델은 앱의 컨텐츠(내용)을 저장하고 여러분의 앱이 무슨 앱인지 정의 해줍니다. 예를들어 은행앱은 금융거래정보가 포함된 데이터베이스를 저장하지만 그림 그리기 앱은 이미지 객체 또는 해당 이미지를 생성할 수 있는 일련의 그리기 명령을 저장할 수도 있습니다. (후자의 경우, 이미지 데이터를 갖고 있기 위한 객체 이기 때문에 이미지 객체를 데이터 객체로 생각해야 할 것입니다.) 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*. 또한 앱은 모든 또는 일부분의 데이터 모델 객체를 관리를 위해 document객체를 사용할 수 있습니다. document객체는 필수적이지는 않지만 하나의 파일 또는 파일 패키지(묶음)에 포함된 데이터 그룹을 관리하는데에 편리한 기능을 제공합니다.
View controller objects	*View controller objects* manage the presentation of your app's content on screen. A view controller manages a single view and its collection of subviews. When presented, the view controller makes its views visible by installing them in the app's window. 뷰 컨트롤러 객체는 앱의 컨텐츠(내용)을 화면에 보여주는 기능을 관리합니다. 하나의 뷰컨트롤러 객체는 하나의 뷰와 그 뷰의 서브뷰(자식뷰)들을 관리하게 됩니다. 뷰 컨트롤러는 뷰를 앱의 윈도우에 올림(설치)하여 화면상에 보여질 수 있도록 합니다. The `UIViewController` class is the base class for all view controller objects. It provides default functionality for loading views, presenting them, rotating them in response to device rotations, and several other standard system behaviors. UIKit and other frameworks define additional view controller classes to implement standard system interfaces such as the image picker, tab bar interface, and navigation interface. `UIViewController`클래스는 모든 뷰 컨트롤러 객체의 베이스가 되는 클래스입니다. `UIViewController`클래스는 뷰들을 로딩하고, 그 뷰들을 보여주고, 장치(아이폰, 아이패드)의 rotations(가로모드, 세로모드)에 반응하여 뷰들을 회전시키고 다른 여러 시스템 기본동작을 위한 기본적인 기능들을 제공합니다. UIKit 그리고 다른 프레임워크들은 이미지 피커, 탭바 인터페이스, 네비게이션 인터페이스와 같은 기본 시스템 인터페이스를 구현하기 위한 뷰 컨트롤러들을 제공하고 있습니다. For detailed information about how to use view controllers, see *View Controller Programming Guide for iOS*. 뷰 컨트롤러에 대한 자세한 내용은 *View Controller Programming Guide for iOS*.을 참고해주세요.
`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. `UIWindow`객체는 화면에 보여질 하나 또는 그 이상의 뷰들을 조정합니다. 대부분의 앱은 메인화면에 컨텐츠를 표현할 오직 한개의 윈도우만 갖게 됩니다. 그러나 앱은 하나 이상의 외부 디스플레이에 올려질 컨텐츠를 위한 추가적인 윈도우를 가질 수도 있습니다. 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. 여러분의 앱의 컨텐츠를 변경하기 위해선, 뷰 컨트롤러를 이용하여 해당 윈도우에 올려진 뷰 들을 변경할 수 있습니다. 절대로 윈도우 그 자체를 변경하거나 다른 것으로 대체하여선 안됩니다. In addition to hosting views, windows work with the `UIApplication` object to deliver events to your views and view controllers. 또한 윈도우는 `UIApplication`객체와 더불어 뷰들과 뷰 컨트롤러에게 이벤트를 전달하는 기능을 가지고 있습니다.
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. 뷰 들과 control은 앱 컨텐츠의 시각적인 표현을 담당합니다. 뷰는 지정된 직사각형 영역안의 컨텐츠(내용)을 그려주고 그 영역에서 발생한 이벤트에 응답하는 객체입니다. 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. UIKit 프레임워크는 여러 다른 타입의 컨텐츠를 보여주기 위한 기본뷰를 제공합니다. 또한 를 `UIView`를 서브클래싱(상속)함으로써 여러분만의 커스텀 뷰들을 만들 수 있습니다. 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. 앱은 뷰 들과 control을 관리하는 것 외에도 Core Animation(코어애니메이션)객체를 뷰와 control계층에 통합하여 관리할 수 있습니다. 레이어 객체는 실제로 시각적인 내용(화면상 보이는 부분)을 나타내는 데이터 객체입니다. 뷰객체들은 뷰의 컨텐츠를 렌더링 하기 위해 레이어 객체들을 사용합니다. 여러분의 인터페이스에 커스텀한 레이어 객체들을 추가하여 복잡한 애니메이션과 여러 다른 타입의 시각적효과들을 구현할 수 있습니다.

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.

하나의 iOS앱이 다른 앱과 구별할 수 있는 점은 그 앱이 관리하는 데이터와 그 데이터를 어떻게 사용자에게 표현하는 것 입니다. 대부분의 UIKit 객체들과의 상호작용만으로 여러분의 앱을 만들 수 없지만 해당 여러기능을 수정하여 목적에 맞게 수정하는데에 도움을 줄 수 있습니다. 예를 들어, 앱이 상태변화(백그라운드, 예상치 못한 강제종료 등)를 여러분이 짠 커스텀 코드가 적절하게 대응할 수 있도록 앱 델리게이트가 알려줄 것입니다.

The Main Run Loop

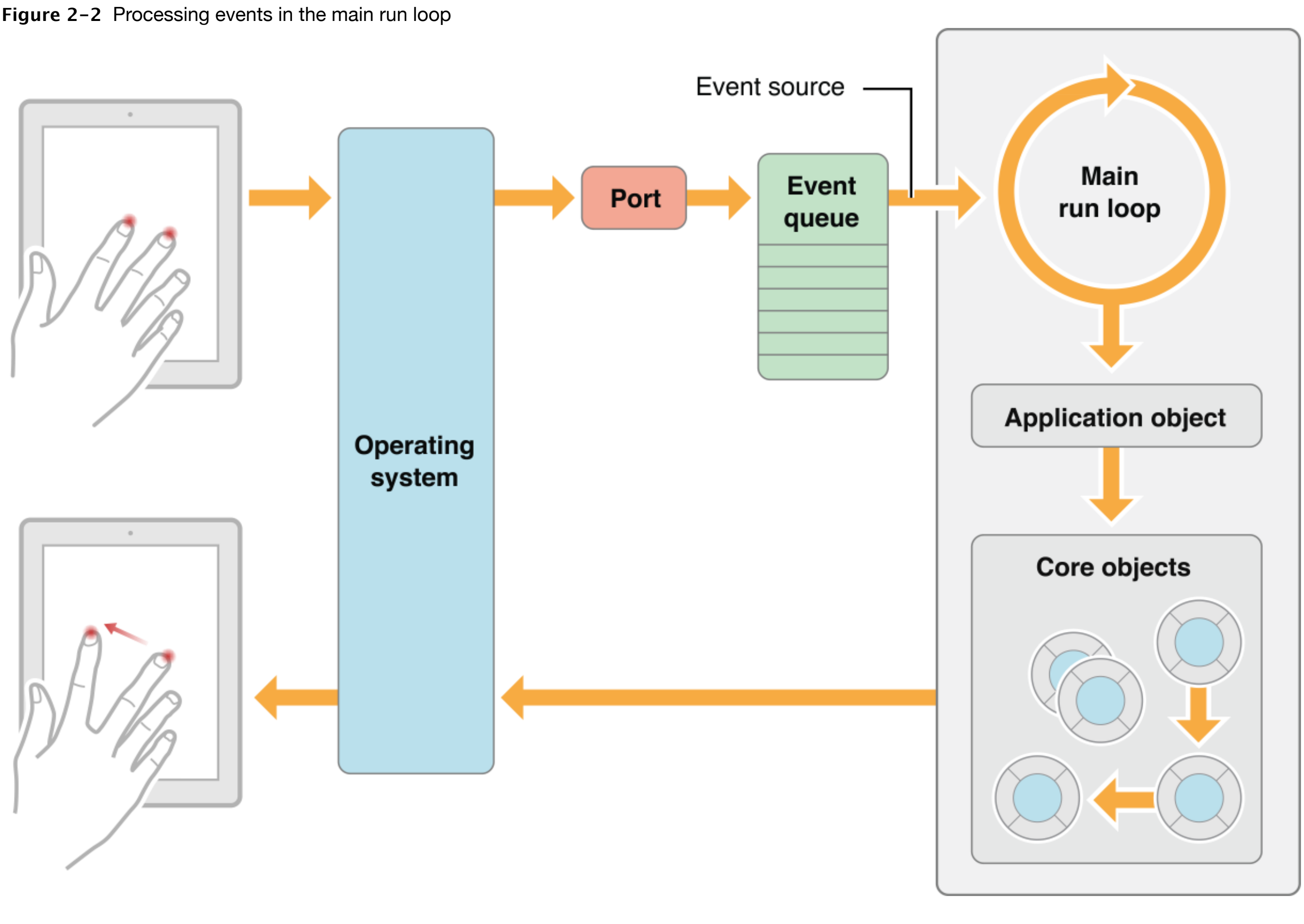
An app's *main run loop* processes all user-related events. The `UIApplication` object sets up the main run loop at launch time and uses it to process events and handle updates to view-based interfaces. As the name suggests, the main run loop executes on the app's main thread. This behavior ensures that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

앱의 메인 run loop는 유저와 관련된 모든 이벤트를 처리합니다. `UIApplication`객체는 앱이 구동될 때 메인 run loop를 설정하고 run loop를 이용하여 이벤트를 처리하고 뷰와 관련된 인터페이스의 업데이트를 처리합니다. 명칭을 보면 알 수 있듯이, 메인 run loop는 메인 스레드에서 실행됩니다. 이는 사용자와 관련된 이벤트를 수신된 순서대로 처리하게끔 해줍니다.

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.

아래의 그림은 메인 run loop의 구조와 사용자의 이벤트(터치, 스와프 등)가 여러분의 앱에 어떻게 영향을 미치는지를 보여줍니다. 사용자가 장치를 상호작용(터치, 스와프)할 때, 그와 관련된 이벤트가 시스템에 의하여 생성되고 UIKit에서 설정된 특별한 통로를 통해 앱에게 전달됩니다. 이벤트들은 앱에 의해 내부적으로 큐에 저장되고 이를 처리하기 위해 메인 run loop에 하나씩 전달되게 됩니다. `UIApplication`객체는 이벤트를 전달받는 첫 번째 겍체이며 그 이벤트를 처리하기 위해 어떠한 것이 필요한지 결정합니다. 터치 이벤트를 보통 메인 윈도우 객체로 전달되고 윈도우 객체는 터치가 발생한 뷰에 그 이벤트를 전달합니다. 그 외의 다른 이벤트는 다양한 객체를 통해 조금 다른 경로로 전달될 수 있습니다.

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*.

iOS앱에선 많은 종류의 이벤트가 다뤄지고 전달됩니다. 그 중 많이 사용되는 이벤트가 아래 Table 2-2에 정리되어 있습니다. 많은 종류의 이벤트는 메인 run loop를 통해 전달되지만 몇몇의 경우는 그렇지 않습니다. 몇몇 이벤트는 델리게이트 객체를 통해 전달되거나 여러분이 생성한 block을 통해 전해지기도 합니다. touch, remote control, motion, accelerometer, and gyroscopic 를 포함한 여러 이벤트를 어떻게 처리하는지 자세히 알고 싶으면 *Event Handling Guide for iOS*.를 참고하세요

Table 2-2 Common types of events for iOS apps

Event type	Delivered to… 누구에게 전달되는가	Notes
Touch	The view object in which the event occurred 이벤트가 발생된 뷰 객체	Views are responder objects. Any touch events not handled by the view are forwarded down the responder chain for processing. 뷰는 객체입니다. 해당 이벤트를 받지 않는 뷰는 그 이벤트를 responder chain을 통해 상위 뷰로 전달합니다.
Remote control Shake motion events	First responder object	Remote control events are for controlling media playback and are generated by headphones and other accessories. Remote control은 미디어 재생을 위한 것으로 헤드폰이나 기타 다른 장치에서 발생됩니다.
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 다시 그려질 필요가 있는 뷰	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*. 이벤트는 이벤트 객체에 관여하지 않지만 단순히 그 뷰 자체를 다시 그리기 위해 뷰를 호출합니다.

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.

터치 그리고 remote control 같은 이벤트들은 앱의 responder objects에 의해 처리됩니다. responder objects는 앱 어디에나 찾아볼 수 있습니다(객체, 뷰 객체들, 뷰 컨트롤러 객체 모두 responder objects 객체입니다). 대부분의 이벤트들은 responder객체를 타겟으로 하지만 해당 이벤트를 처리할 수 있는 다른 responder 객체에게 이벤트를 넘겨줄 수 있습니다(responder 체인을 통해서). 예를 들어, 특정 이벤트를 받지 않는 뷰는 해당 이벤트를 자신의 superview(부모뷰) 또는 뷰 컨트롤러 객체에 넘겨줄 수 있습니다.

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.

버튼과 같은 controls안에서 일어나는 터치 이벤트는 다른 종류의 뷰에서 일어나는 이벤트와는 조금 다른 처리과정을 가집니다. 이러한 이벤트들은 액션 메시지로 다시 패키징되어 적절한 타겟 객체에 전달되어집니다. 이러한 target-action 디자인 패턴은 여러분의 코드가 적절히 실행될 수 있도록 합니다.

Execution States for Apps

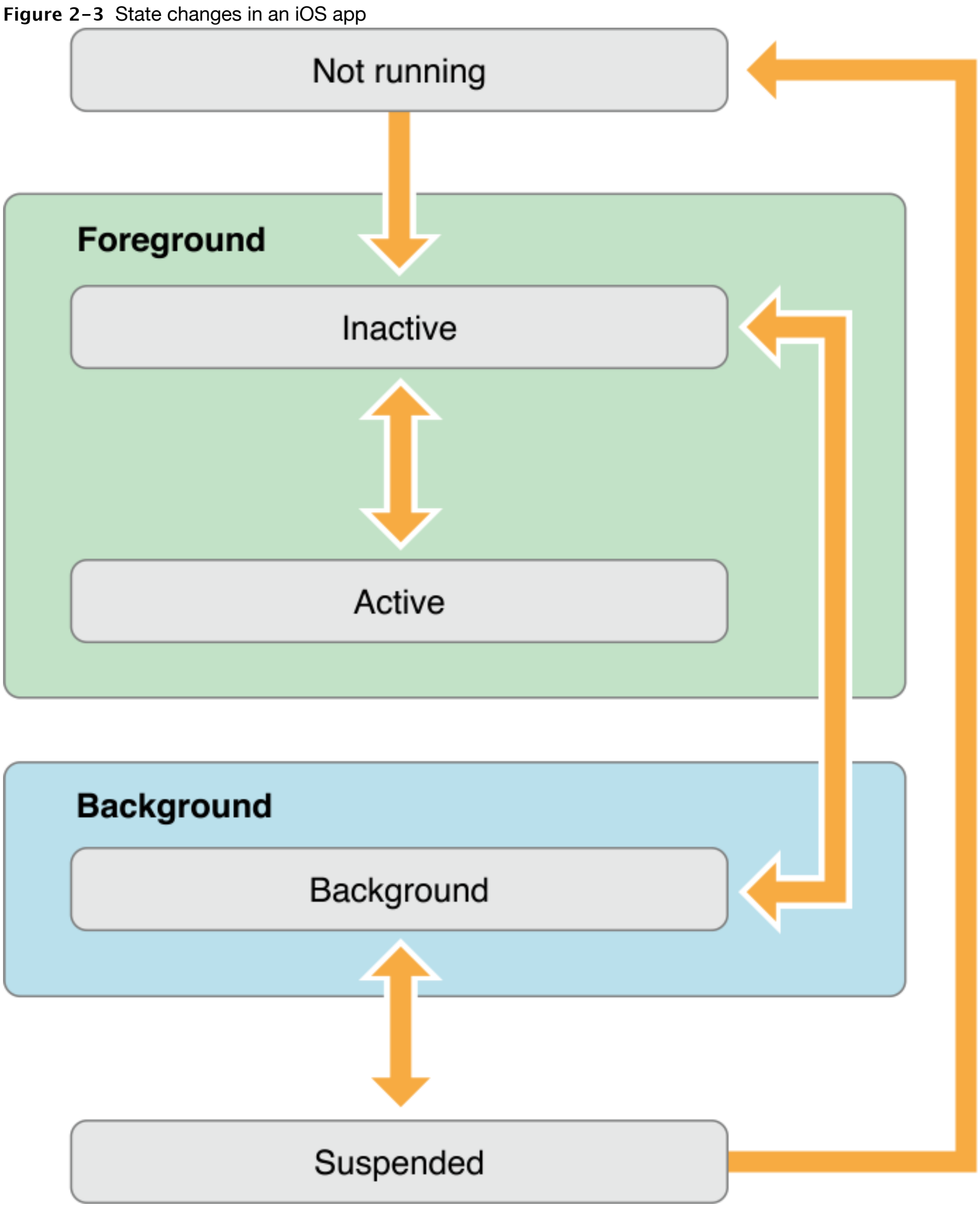
At any given moment, your app is in one of the states listed in Table 2-3. The system moves your app from state to state in response to actions happening throughout the system. For example, when the user presses the Home button, a phone call comes in, or any of several other interruptions occurs, the currently running apps change state in response. Figure 2-3 shows the paths that an app takes when moving from state to state.

여러분의 앱이 가질 수 있는 모든 상태를 아래 Table 2-3에 정의해 놓았습니다. 시스템은 시스템에서 일어나는 액션에 대응하여 앱의 상태를 바꿉니다. 예를 들어 사용자가 홈 버튼을 눌렀을 시, 전화가 왔을 시 또는 여러 상황에서 현재 동작하고 있는 앱이 이에 대응하여 상태를 변화시키게 됩니다. Figure 2-3 아래의 예는 특성 상태로 변화되었을 때 앱이 수행하는 과정을 설명하였습니다.

Table 2-3 App states

State	Description
Not running	The app has not been launched or was running but was terminated by the system. 앱이 아직 실행되지 않았거나 시스템에 의하여 강제종료된 경우
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. 앱이 foreground에서 동작하고 있으나 이벤트를 받지 못하는 상태입니다. 앱은 종종 상태가 변화될 때 잠깐동안 Inactive가 될 수 있습니다.
Active	The app is running in the foreground and is receiving events. This is the normal mode for foreground apps. 앱이 foreground에서 동작하고 있으며 이벤트를 받을 수 있는 상태입니다.
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. 앱이 백그라운드에서 동작하며 코드를 실행시키고 있는 상태입니다. 대부분의 앱들은 잠시 suspended(중지)될 때 백그라운드 상태가 될 수 있습니다. 하지만 백그라운드에서 바로 동작된 앱은 inactive 상태 대신 백그라운드상태로 유지됩니다. 백그라운드로 동작하는 동안 어떻게 코드를 실행시킬지에 대해선 Background Execution.을 참고해주세요.
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. 앱이 백그라운드 상태에 있지만 아무런 코드도 수행하지 않는 상태입니다. 시스템은 자동으로 앱을 Suspended상태로 변화시킬 수 있으며 미리 앱에게 알려주지 않습니다. Suspended되는 동안 앱은 메모리에 남아있으나 아무런 코드를 실행시키지 못합니다. When a low-memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app. 시스템은 메모리가 부족하게 될 시, foreground에서 동작하고 있는 앱을 위해 아무런 사전통지없이 Suspended(일시중지)된 앱을 제거할 수 있습니다.

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.

대부분의 앱 상태변화는 앱 델리게이트 객체의 메소드 호출과 함께 일어나게 됩니다. 이 메소드들은 상태 변화에 대응하여 적절한 코드를 수행할 수 있도록 해줍니다. 아래에 앱 델리게이트 메소드들을 나열하였으며 어떻게 사용해야할지 요약하였습니다.

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

  이 메소드는 앱이 실행되고 첫번째로 코드를 실행시킬 수 있는 델리게이트 메소드입니다.

- `application:didFinishLaunchingWithOptions:`—This method allows you to perform any final initialization before your app is displayed to the user.

  이 메소드를 이용하여 여러분의 앱이 화면에 나타나기 전에 최종적으로 초기화 작업을 해줄 수 있습니다.

- `applicationDidBecomeActive:`—Lets your app know that it is about to become the foreground app. Use this method for any last minute preparation.

  이 메소드는 여러분의 앱이 foreground상태로 넘어가는 것을 알려줍니다.

- `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.

  앱이 foreground로 상태변화 되는 중 임을 알려줍니다.

- `applicationDidEnterBackground:`—Lets you know that your app is now running in the background and may be suspended at any time.

  앱이 백그라운드상태이며 언제라도 suspended(일시정지)될 수 있음을 알립니다.

- `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.

  앱이 백그라운드에서 foreground로 상태변화 중 임을 알려줍니다. 하지만 아직 active상태는 아닙니다.

- `applicationWillTerminate:`—Lets you know that your app is being terminated. This method is not called if your app is suspended.

  앱이 곧 종료 됨을 알립니다. 앱이 suspended(일시정지)상태일 땐 이 메소드가 호출되지 않습니다.

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.

앱은 언제라도 종료될 수 있기 때문에 항상 준비된 상태여야 하며 사용자가 데이터를 저장하거나 다른 중요한 작업을 수행할 때까지 기다려선 안됩니다(즉 사용자가 대비하는 것을 기대하지말고 대비해야 합니다). 시스템에 의해 앱이 종료되는 것은 보통 앱에서 일어날 수 있는 일입니다. 시스템은 종종 메모리 관리차원 그리고 유저에 의해 방금 실행된 앱을 위한 메모리공간을 위해 앱을 강제종료 시킬 수 있습니다. 또한 시스템이 오작동하거나 이벤트에 적절히 대응하지 못하는 앱들을 종료하게 할 수 있습니다.

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.

Suspended(일시정지)된 앱은 강제종료 될 시 아무린 notification(통지)를 받지 못합니다. 만약 앱이 백그라운드에서 동작 중이고 suspended상태가 아니면 시스템은 종료되기 전에 앱 델리게이트의 `applicationWillTerminate:` 메소드를 호출합니다. 장치가 재부팅될 때엔 시스템은 이 메소드를 호출하지 않습니다.

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.

시스템이 앱을 강제종료 시키는 것 외에도 사용자가 multitasking을 이용하여 앱을 종료시킬 수 있습니다. 유저에 의해 종료되는 것 또한 suspended상태인 앱을 종료시키는 것과 똑같은 결과를 냅니다.

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.

시스템은 메인 스레드를 생성하고 다른 작업을 위해 필요할 시 추가 스레드를 생성할 수 있습니다. iOS에선 여러분이 스스로 스레드를 생성하거나 관리하기보단 Grand Central Dispatch (GCD), operation objects, 다른 비동기 기능을 사용하는 것이 효과적입니다.

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*.