The App Life Cycle

* iOS Apps acts like intermediate between the user defined code and the frameworks available in the system.
* Frameworks from the system provide the basic things that all apps need to run, and it also allows us to change the system frameworks to write code according to our requirements.
* iOS frameworks follows the Model View Controller (MVC) architecture to manage the frameworks.
* To create one successful App understanding of the MVC architecture is mandatory.
* By using MVC architecture it helps us to better understanding of Objective-c and its feature.

**Example: 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]));

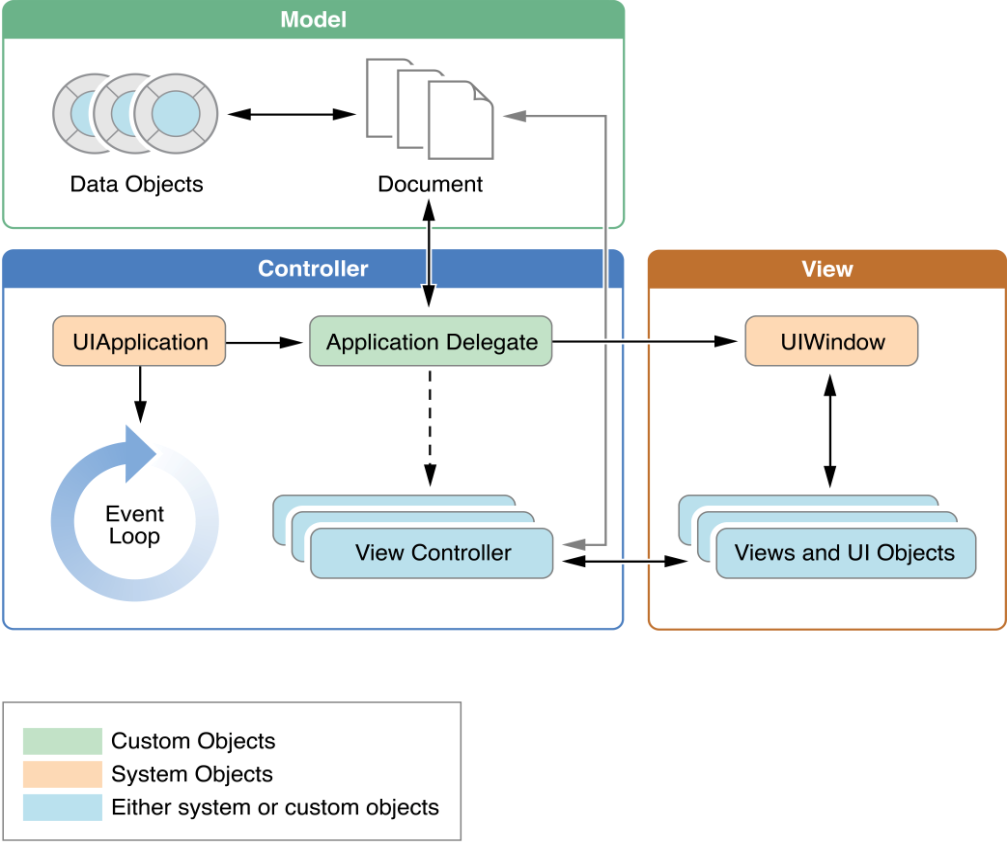
}

}

* The important  function of main  is that its job is to hand control off to the UIKit framework
* The [UIApplicationMain](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKitFunctionReference/index.html#//apple_ref/c/func/UIApplicationMain) function do the following things:
* It handles the process by creating the core objects of the app
* It loads the app’s user interface from the available storyboard files
* It calls the custom code so that we have a chance to do some initial setup, and putting the app’s run loop in motion.
* The only pieces that the developer has to provide are the storyboard files and the custom initialization code.

## The Structure of an App

* When App startup, the [UIApplicationMain](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKitFunctionReference/index.html#//apple_ref/c/func/UIApplicationMain) function sets the several key objects and starts the app running.
* The [UIApplication](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html#//apple_ref/occ/cl/UIApplication) is known as the heart of every iOS App and its job is to facilitate the interactions between the system objects and other objects in the App.
* The below Figure 1 shows the most commonly used objects in all Apps
* The important thing is that iOS apps use a Model-View-Controller architecture (MVC).
* This used here to separate the Apps data and business logic that the developer can understand the code better after seeing it.
* The Model-View-Controller (MVC) architecture is mandatory to create App because the App is going to run in different devices with different screen sizes.



**Figure 1** key Objects in an iOS app

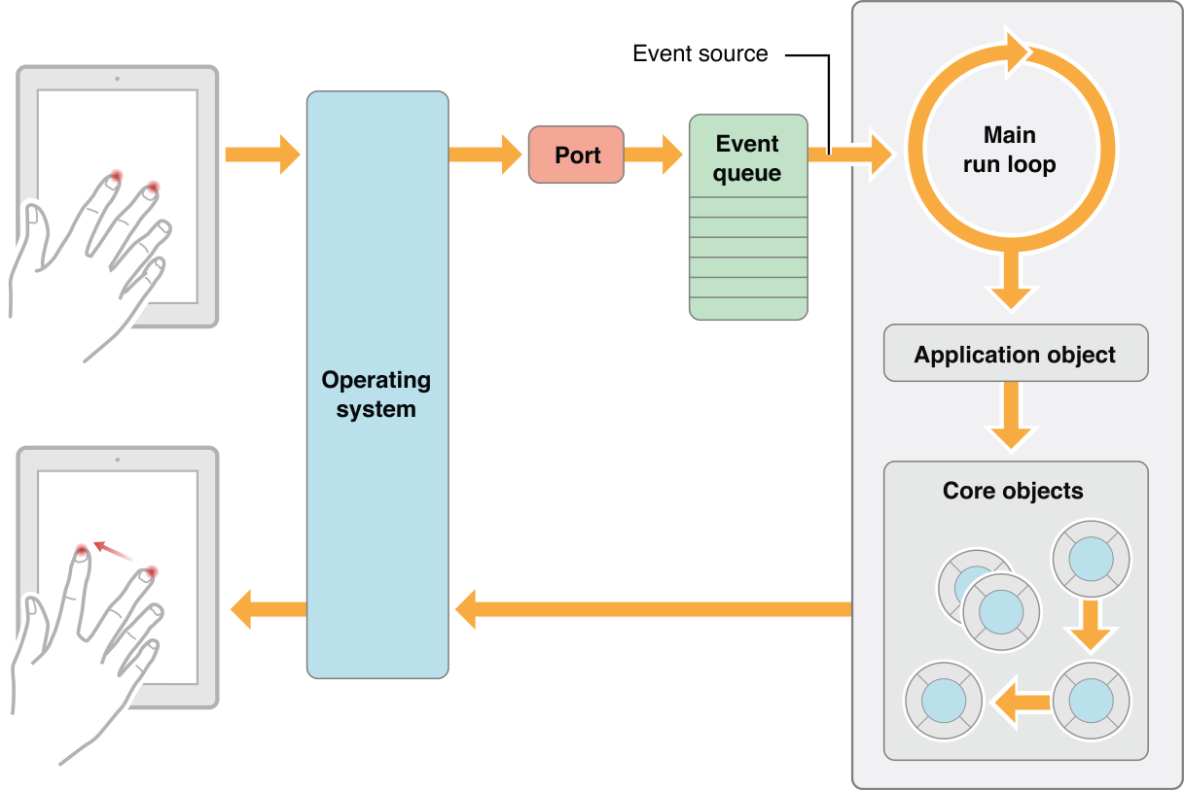
**Courtesy: Apple documentation**

**Table 1** The role of objects in an iOS app

|  |  |
| --- | --- |
| **Object** | **Description** |
| [UIApplication](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html#//apple_ref/occ/cl/UIApplication) object | * The UIApplication object has the ability to manage the event loop and other high-level behaviors of an App. * The UIApplication object also reports key app transitions and some special events (such as incoming push notifications) to its delegate, which is a custom written by us. |
| App delegate object | * The app delegate is known as the heart of the custom code. * This object works along with the UIApplication object to handle app initialization, state transitions, and many high-level app events * This object is present in every App because it is used to set up the App’s initial data structures |
| Documents and data model objects | * Data model objects store the app’s content and is specific to the 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. * The Apps can also use document objects these are subclasses of [UIDocument](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIDocument_Class/index.html" \l "//apple_ref/occ/cl/UIDocument" \t "_self) to manage some or all of their data model objects. * Document objects are not necessary but it helps to make a convenient way to group data that belongs in a single file or file package |
| View controller objects | * The View controller objects are used to manage the presentation of the app’s content on screen. * The view controller are used manage a single view and its collection of sub views. * When presented, the view controller makes its views visible by installing them in the app’s window. * The [UIViewController](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/index.html#//apple_ref/occ/cl/UIViewController) class is the base class for all view controller objects. * The [UIViewController](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/index.html#//apple_ref/occ/cl/UIViewController) provides default functionality like loading views, presenting them, rotating them in response to device rotations, and several other standard system behaviors. * We can use UIKit and other frameworks to define additional view controller classes to implement standard system interfaces such as the image picker, tab bar interface, and navigation interface. |
| [UIWindow](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWindow_Class/index.html#//apple_ref/occ/cl/UIWindow) object | * The [UIWindow](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWindow_Class/index.html#//apple_ref/occ/cl/UIWindow) object coordinates or manages 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 the app, then use a view controller to change the views displayed in the corresponding window and you are not suppose to replace the window itself. |
| View objects, control objects, and layer objects | * The Views and controls are used to provide the visual representation of the 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. * By subclassing [UIView](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/index.html#//apple_ref/occ/cl/UIView)  it is possible to define a custom view. * 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 or to create their content. * We can also add custom layer objects to our interface to implement complex animations and other types of sophisticated visual effects. |

## The Main Run Loop

* The app is having main run loop it is responsible for processing all user-related events.
* The [**UIApplication**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html#//apple_ref/occ/cl/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 shows the architecture of the main run loop and how the user events result in actions taken by the 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



**Figure 2** Processing events in the main run loop

**Courtesy: Apple documentation**

* 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 as shown in fig1.
* All touch events are usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred.
* Whatever Other events happens might take slightly different paths through various app objects.
* Many types of events can be sending into an iOS app but many of these event types are delivered using the main run loop of the app, but some are not.
* Some events are sent to a delegate object or are passed to a block that the developers provide

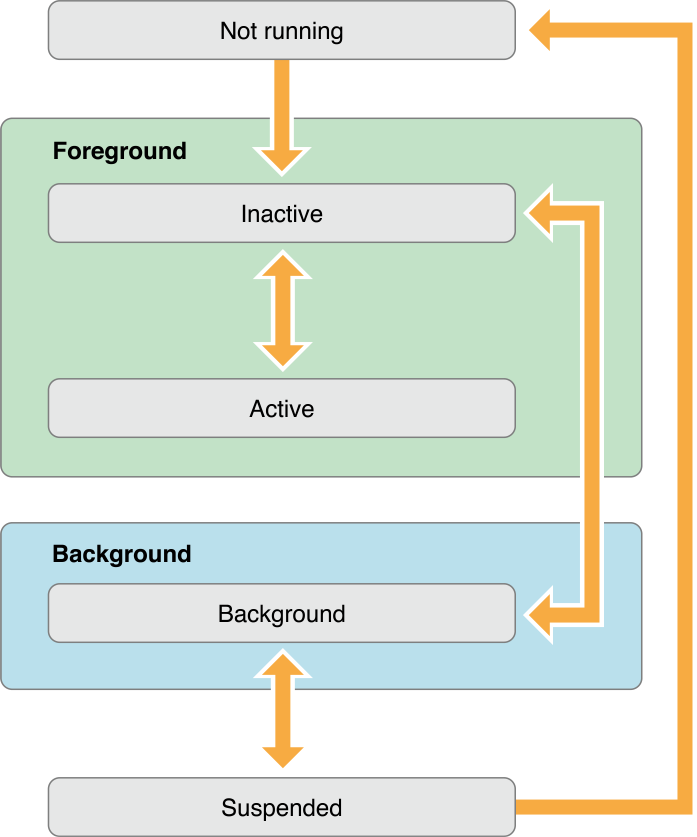
**Table 2** Common types of events for iOS apps

|  |  |  |
| --- | --- | --- |
| **Event type** | **Delivered to…** | **Notes** |
| Touch | The view object in which the event occurred | * The Views are responder objects. * Any touch events which are not handled by the view are forwarded down the responder chain for processing. |
| Remote control  Shake motion events | First responder object | The Remote control events are like controlling media playback which are generated by headphones and other accessories. |
| Accelerometer  Magnetometer  Gyroscope | The object we define | Events related to the accelerometer, magnetometer, and gyroscope hardware are delivered to the object we define. |
| Location | The object we define | We register to receive location events using the Core Location framework. |
| Redraw | The view that needs the update | The Redraw events do not involve an event object but are simply calls to the view to draw itself. |

* Some events like touch and remote control events are handled by the app’s responder objects.
* Responder objects are everywhere in your app. (The [UIApplication](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html#//apple_ref/occ/cl/UIApplication) object, the view objects, and the view controller objects are all examples of responder objects.)
* Most events targets a specific responder object but if not then 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 then it can be passed to the event superview or to a view controller.
* The Touch events occurring in controls like buttons are handled differently than touch events occurring in many other types of views.
* There are typically only a limited number of interactions are possible with a control, and so those interactions are repackaged into action messages and delivered to an appropriate target object.

## Execution States for Apps

* At any given moment, the app is in one of the following states listed in Table 3.
* The system moves the app from one state to another state with response to the actions that are 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.
* The [Figure 3](https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html#//apple_ref/doc/uid/TP40007072-CH2-SW6) shows the paths that an app can takes when moving from one state to another state



**Figure 3** State changes in an iOS app

**Courtesy: Apple documentation**

**Table 3** App states

|  |  |
| --- | --- |
| **State** | **Description** |
| Not running | The app is in **Not running** state only when:   * The App not been launched * It was running but it was terminated by the system. |
| Inactive | The app is in **Inactive** state only when:   * The app is running in the foreground but it 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 in **Active** state only when:   * The app is running in the foreground and is receiving events. This is the normal mode for foreground apps. |
| Background | The app is in **Background** state only when:   * The app is in the background but it still alive and executing code. * Most apps enter this state briefly on their way to being suspended. * However, an app that requests extra execution time and may remain in this state for a period of time. * In addition, an app being enters directly into the background state instead of entering into the inactive state. |
| Suspended | The app is in **Suspended** state only when:   * When the app is in the background but is not executing code then the system moves the apps into 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 remove the suspended apps without notice to make more space for the Apps which are running in foreground app. |

* The life cycle methods can be used by calling the corresponding methods to the app delegate object.
* These methods help to give respond in an appropriate way when the App changes its state.
* Thy lifecycle methods are listed below, along with how we might use them.
* **application:willFinishLaunchingWithOptions**: This method is for app’s first chance to execute code at launch time.
* **application:didFinishLaunchingWithOptions:** This method allows for Apps to perform any final initialization before the app is displayed to the user.
* **applicationDidBecomeActive:** It lets the app to know that it is about to become the foreground app. Use this method for any last minute preparation.
* **applicationWillResignActive:** Lets us to know that the app is transitioning away from being the foreground app. Use this method to put your app into a passive state.
* **applicationDidEnterBackground:** Lets us to know that the app is now running in the background and may be suspended at any time.
* **applicationWillEnterForeground:** Lets us to know that your app is moving out of the background and back into the foreground, but that it is not yet active.
* **applicationWillTerminate:** Lets us to know that the app is being terminated. This method is not called if your app is suspended.

## App Termination

* Apps can be terminated at any time so Apps must be prepared for termination to happen at any time and it should not wait to save user data or perform any 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 back and make space 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.
* The suspended apps will not receive any notification when they are terminated and the system kills the process and reclaims back the corresponding memory.
* If an app is currently running in the background and not suspended, then the system calls the applicationWillTerminate: of its app delegate before to termination. The system does not call this method when the device reboots.
* In addition to the system terminating the app, the user can terminate the app explicitly by using the multitasking UI.
* The user-initiated termination has the same effect as terminating a suspended app. Then the app’s process is killed and no notification is sent to the app.

## Threads and Concurrency

* Thread is nothing but running the different part of the program simultaneously.
* In App’s the system creates the app’s main thread and we 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 by ourselves.
* Technologies such as GCD let us to define the work we want to do and the order that we want to do it in, but let the system decide how best way to execute that work on the available CPUs.
* By letting to the system to handle the thread management it simplifies the code that we must write and makes it easier to ensure the correctness of that code, and offers better overall performance.

**While thinking about threads and concurrency, consider the following:**

* The works 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.

References:

* https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html