



iOS

Succinctly

by Ryan Hodson

Chapter 1 Hello, iOS!

In this chapter, we'll introduce the three main design patterns underlying all iOS app development: model-view-controller, delegate objects, and target-action. The model-view-controller pattern is used to separate the user interface from its underlying data and logic. The delegate object pattern makes it easy to react to important events by abstracting the handling code into a separate object. Finally, the target-action pattern encapsulates a behavior, which provides a very flexible way to perform actions based on user input.

We'll talk about all of these patterns in more detail while we're building up a simple example application. This will also give us some experience with basic user interface components like buttons, labels, and text fields. By the end of this chapter, you should be able to configure basic layouts and capture user input on your own.

Creating a New Project

First, we need to create a new Xcode project. Open Xcode and navigate to **File > New > Project**, or press **Cmd+Shift+N** to open the template selection screen. In this chapter, we'll be creating the simplest possible program: a *Single View Application*. Select the template, and then click **Next**.

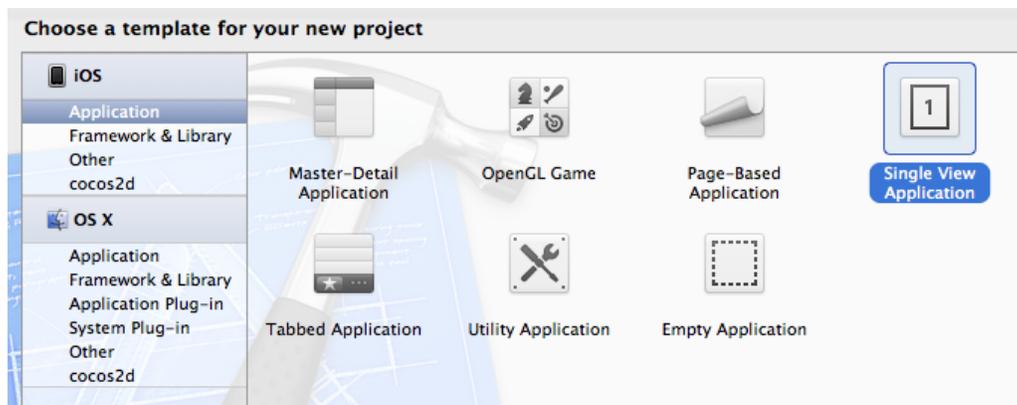


Figure 2: Selecting the Single View Application template

Use *HelloWorld* for the **Product Name**, anything you like for **Organization Name**, and *edu.self* for the **Company Identifier**. Make sure that **Devices** is set to **iPhone** and that the **Use Storyboards** and **Use Automatic Reference Counting** options are selected:

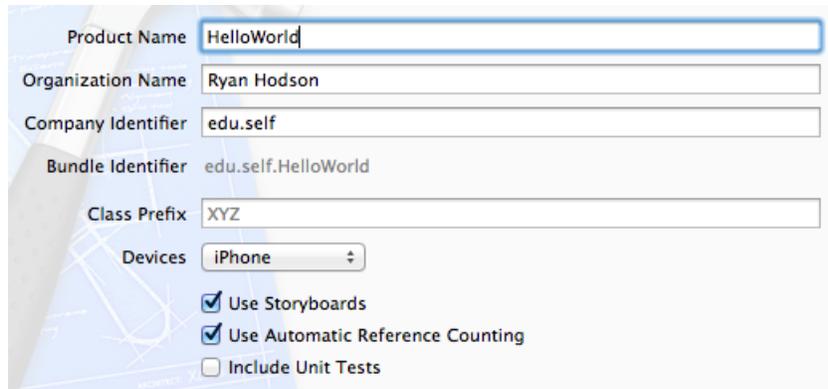


Figure 3: Configuration for our HelloWorld app

Then, choose a location to save the file, and you'll have your very first iOS app to experiment with.

Compiling the App

As with the command-line application from *Objective-C Succinctly*, you can compile the project by clicking the **Run** button in the upper-left corner of Xcode or using the Cmd+R keyboard shortcut. But, unlike *Objective-C Succinctly*, our application is a graphical program that is destined for an iPhone. Instead of simply compiling the code and executing it, Xcode launches it using the **iOS Simulator** application. This allows us to see what our app will look like on the iPhone without having to upload it to an actual device every time we make the slightest change. The template we used is a blank project, so you'll just see a white screen when you run it:



Figure 4: Running the HelloWorld project in the iOS Simulator

While we can't really tell with our current app, the simulator is a pretty detailed replica of the actual iPhone environment. You can click the home button, which will display all the apps that we've launched in the simulator, along with a few built-in ones. As we'll see in a moment, this lets us test the various states of our application.

App Structure Overview

Before we start writing any code, let's take a brief tour of the files provided by the template. This section introduces the most important aspects of our HelloWorld project.

main.m

As with any Objective-C program, an application starts in the `main()` function of `main.m`. The `main.m` file for our HelloWorld project can be found in the **Supporting Files** folder in Xcode's **Project Navigator** panel. The default code provided by your template should look like the following:

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

#import "AppDelegate.h"
```

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

This launches your application by calling the `UIApplicationMain()` function, and passing `[AppDelegate class]` as the last argument tells the application to transfer control over to our custom `AppDelegate` class. We'll discuss this more in the next section.

For most applications, you'll never have to change the default `main.m`—any custom setup can be deferred to the `AppDelegate` or `ViewController` classes.

AppDelegate.h and AppDelegate.m

The iOS architecture relies heavily on the **delegate design pattern**. This pattern lets an object transfer control over some of its tasks to another object. For example, every iOS application is internally represented as a [UIApplication](#) object, but developers rarely create a `UIApplication` instance directly. Instead, the `UIApplicationMain()` function in `main.m` creates one for you and points it to a delegate object, which then serves as the root of the application. In the case of our HelloWorld project, an instance of the custom `AppDelegate` class acts as the delegate object.

This creates a convenient separation of concerns: the `UIApplication` object deals with the nitty-gritty details that happen behind the scenes, and it simply informs our custom `AppDelegate` class when important things happen. This gives you as a developer the opportunity to react to important events in the app's life cycle without worrying about how those events are detected or processed. The relationship between the built-in `UIApplication` instance and our `AppDelegate` class can be visualized as follows:

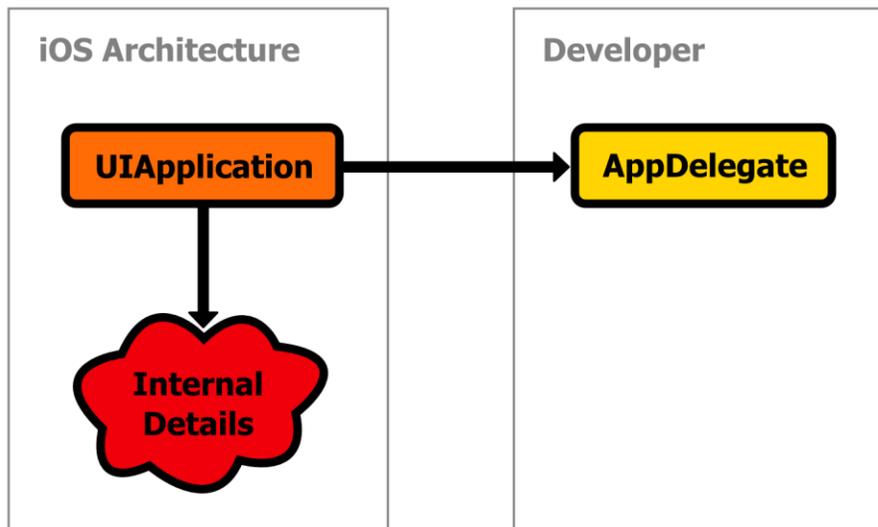


Figure 5: Using AppDelegate as the delegate object for UIApplication

Recall from *Objective-C Succinctly* that a protocol declares an arbitrary group of methods or properties that any class can implement. Since a delegate is designed to take control over an arbitrary set of tasks, this makes protocols the logical choice for representing delegates. The [UIApplicationDelegate](#) protocol declares the methods that a delegate for **UIApplication** should define, and we can see that our **AppDelegate** class adopts it in **AppDelegate.h**:

```
@interface AppDelegate : UIResponder <UIApplicationDelegate>
```

This is what formally turns our **AppDelegate** class into the delegate for the main **UIApplication** instance. If you open **AppDelegate.m**, you'll also see implementation stubs for the following methods:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions;
- (void)applicationWillResignActive:(UIApplication *)application;
- (void)applicationDidEnterBackground:(UIApplication *)application;
- (void)applicationWillEnterForeground:(UIApplication *)application;
- (void)applicationDidBecomeActive:(UIApplication *)application;
- (void)applicationWillTerminate:(UIApplication *)application;
```

These methods are called by **UIApplication** when certain events occur internally. For example, the **application:didFinishLaunchingWithOptions:** method is called immediately after the application launches. Let's take a look at how this works by adding an **NSLog()** call to some of these methods:

```
- (BOOL)application:(UIApplication *)application
```

```

    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
        NSLog(@"Application has been launched");
        return YES;
    }
- (void)applicationDidEnterBackground:(UIApplication *)application {
    NSLog(@"Entering background");
}
- (void)applicationWillEnterForeground:(UIApplication *)application {
    NSLog(@"Entering foreground");
}
}

```

Now, when you compile the project and run it in the iOS Simulator, you should see the **Application has been launched** message as soon as it opens. You can click the simulator's home button to move the application to the background, and click the application icon on the home screen to move it back to the foreground. Internally, clicking the home button makes the **UIApplication** instance call **applicationDidEnterBackground::**

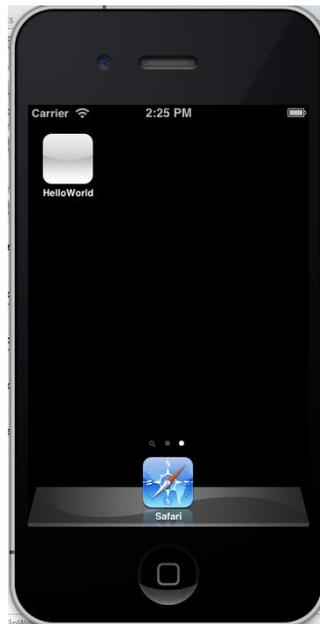


Figure 6: Moving the HelloWorld application to the background

This should display the following messages in Xcode's output panel:

```

All Output ⌵ Clear [ ] [ ] [ ]
2012-10-28 14:25:18.854 HelloWorld[19141:11303] Application
has been launched
2012-10-28 14:25:21.290 HelloWorld[19141:11303] Entering
background

```

Figure 7: Xcode output after clicking the home button in the iOS Simulator

These **NSLog()** messages show us the basic mechanics behind an application delegate, but in

the real world, you would write custom setup and cleanup code to these methods. For example, if you were creating a 3-D application with OpenGL, you would need to stop rendering content and free up any associated resources in the `applicationDidEnterBackground:` method. This makes sure that your application isn't hogging memory after the user closes it.

To summarize, our `AppDelegate` class serves as the practical entry point into our application. Its job is to define what happens when an application opens, closes, or goes into a number of other states. It accomplishes this by acting as a delegate for the `UIApplication` instance, which is the internal representation of the entire application.

ViewController.h and ViewController.m

Outside of the application delegate, iOS applications follow a model-view-controller (MVC) design pattern. The model encapsulates the application data, the view is the graphical representation of that data, and the controller manages the model/view components and processes user input.

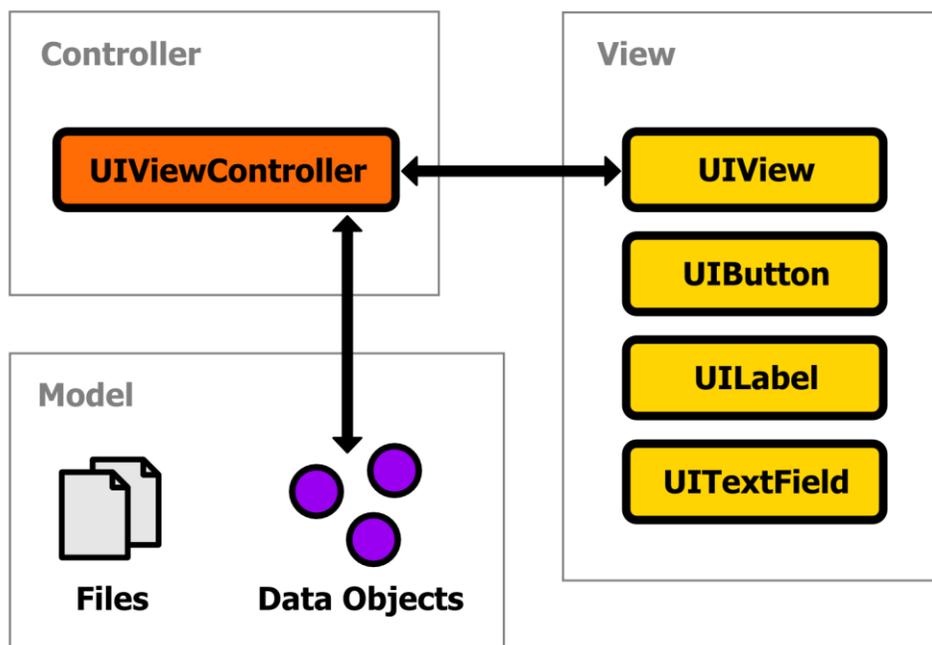


Figure 8: The model-view-controller pattern used by iOS applications

Model data is typically represented as files, objects from the [CoreData](#) framework, or custom objects. The application we're building in this chapter doesn't need a dedicated model component; we'll be focusing on the view and controller aspects of the MVC pattern until the next chapter.

View components are represented by the [UIView](#) class. Its `UIButton`, `UILabel`, `UITextField` and other subclasses represent specific types of user interface components, and `UIView` itself can act as a generic container for all of these objects. This means that assembling a user