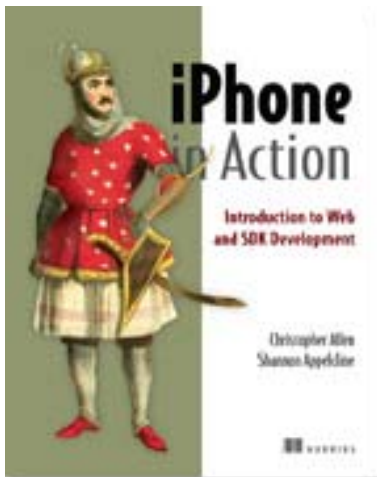


Creating a first project in Xcode: Hello, World!

Excerpted from



[iPhone in Action](#)

EARLY ACCESS EDITION

*Introduction to Web and SDK
Development*
**Christopher Allen and Shannon
Appelcline**

MEAP Release: May 2008

Softbound print: January 2009 (est.) | 350
pages

ISBN: 193398886X

This article is excerpted from the upcoming book [iPhone in Action: Introduction to Web and SDK Development](#) by Christopher Allen and Shannon Appelcline and published by Manning Publications. It takes a look at the parts of a standard SDK program via a Hello, World! application using Xcode, the SDK's main development environment. For the table of contents, the Author Forum, and other resources, go to <http://manning.com/callen/>.

Every project should begin by running "File > New Project," choosing a template, and naming your file. For our first sample project, we selected the Window-Based Application template and the name "helloworldxc."

Before we start actually writing new code, we want to offer a basic understanding of what's there already, which we're going to do by examining the contents of the three most important files that were created by our basic template: main.m, helloworldxcAppDelegate.h, and helloworldxcAppDelegate.m.

Understanding main.m

The first file created by Xcode is main.m, which contains your main function, as shown in listing 1.

Listing 1 main.m comes with some standard code preinstalled for you

```
#import <UIKit/UIKit.h>                                1

int main(int argc, char *argv[]) {

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];    2
    int retVal = UIApplicationMain(argc, argv, nil, nil);          3
    [pool release];                                               4
    return retVal;
}

1 Import standard header
2 Create autorelease pool
3 Run program
4 Release autorelease pool
```

The creation of this main routine is automatic, and you generally shouldn't have to fool with it *at all*. However, it's worth understanding what's going on. You start off with an `#import` directive #1, which you'll recall is Objective-C's substitute for `#include`. More specifically, you've included the UIKit framework, the most important framework in Objective-C. This actually isn't needed, because it's also in the Prefix.pch file, but at least at the time of this writing, it's part of the default main.m file.

You next create an `NSAutoreleasePool` #2. You'll recall that we mentioned this in our discussion of memory management in the previous chapter. It's what supports the `NSObject`'s `autorelease` method. Also note that you release the pool itself after you've run your application's main routine, following the standard rule that if you `alloc` an object, you must also release it.

The `UIApplicationMain` line #3 is what creates your application and kicks off your event cycle. It works like this:

```
int UIApplicationMain (
    int argc,
    char *argv[],
    NSString *principalClassName,
    NSString *delegateClassName
);
```

As with the rest of the main.m file, you should never have to change this, but we're nevertheless going to briefly touch upon what the latter two arguments mean—though they'll usually be set to default thanks to the `nil` arguments.

The `principalClassName` defines the application's main class, `UIApplication` by default. This is the class that does a lot of the action- and event-controlling for your program, topics that we're going to return to in chapter 14.

The `UIApplication` object is created as part of this startup routine, but you'll note that no link to the object is provided. If you ever need to access it (and you will), you can use a `UIApplication` class method to do so:

```
[UIApplication sharedApplication];
```

This will return the application object. It will typically be sent as part of a nested message to a UIApplication method, as we'll see in future chapters.

The `delegateClassName` defines the application object's delegate, an idea we met in the last chapter. As we noted there, this is the object that responds to some of the application's messages, as defined by the UIApplicationDelegate protocol. Among other things, the application delegate must respond to life-cycle messages, most importantly the `applicationDidFinishLaunching:` message which is what runs your program's actual content.

In Xcode's templates your delegate class files will always have the name `projectAppDelegate`. Your program finds them, thanks to a `delegate` link that's built into Interface Builder.

You *could* change the arguments sent to `UIApplicationMain` and you *could* add other commands to the `main.m` file, but generally you don't want to. The defaults should work fine for any program that you're likely to program in the near future. Thus, you should now put `main.m` away and look at the file where any programming actually starts: your application delegate.

Understanding the application delegate

As we've already seen, the application delegate is responsible for answering many of the application's messages. You can refer to the previous chapter for a list of some of the more important ones or to Apple's UIApplicationDelegate protocol reference for a complete listing.

More specifically, an application delegate should do the following:

- At launch time, it must create an application's windows, and display them to the user.
- It must initialize your data.
- It must respond to "quit" requests.
- It must handle low-memory warnings.

Of these topics, it's the first that's of importance to you now. Your application delegate files, `helloworldxcAppDelegate.h` and `helloworldxcAppDelegate.m`, are what will get your program started.

The Header File

Now that you've moved past `main.m`, you're actually using classes, which is the sort of coding that makes up the vast majority of Objective-C code. Listing 2 shows the contents of your first class' header file, `helloworldxcAppDelegate.h`.

Listing 2 The Application Delegate header contains some basic definitions

```
@interface helloworldxcAppDelegate : NSObject <UIApplicationDelegate> { 1
    IBOutlet UIWindow *window; 2
}

@property (nonatomic, retain) UIWindow *window; 3
```

@end

- 1 Begin header definition**
- 2 Incorporate Interface Builder object**
- 3 Define variable as property**

Again, there's nothing you're going to change here, but we want to examine the contents, both to make concrete some of the lessons learned in the last chapter and to have a good foundation for work you're going to do in the future.

First, you'll see an interface line #1 that subclasses your delegate off of NSObject (which is appropriate, since the app delegate is a non-displaying class) and includes a promise to follow the UIApplicationDelegate protocol.

Next, you have the declaration of an instance variable, window #2. It's preceded by an IBOutlet statement, which tells you that the object was actually created in Interface Builder. We'll examine this concept in more depth in the next chapter, but for now you just need to know that you have a window object already prepared for your usage.

Finally, you declare that window as a property #3. You'll note this statement includes some of those property attributes that we mentioned, here nonatomic and retain.

Though you won't modify the header file in this example, you will in the future, and you'll generally be repeating the patterns you see here: creating more instance variables, including IBOutlets, and defining more properties. You may also declare methods in this header file, something that this first header file doesn't contain.

The Source Code File

Listing 3 displays the application delegate's source code file, helloworldxcAppDelegate.m, and it's here that you're going to end up placing your new code.

Listing 3 The Application Delegate object contains your startup code

```
#import "helloworldxcAppDelegate.h" 1
@implementation helloworldxcAppDelegate 2
@synthesize window; 3
- (void)applicationDidFinishLaunching:(UIApplication *)application { 4
    [window makeKeyAndVisible]; 5
}
- (void)dealloc {
    [window release];
    [super dealloc];
}
@end
```

The source begins with an inclusion of the class' header file #1 and an @implementation statement #2. Your window property is also synthesized #3.

It's the content of the applicationDidFinishingLaunching method #4 that's really of interest to you. As you'll recall, that's one of the iPhone OS life-cycle messages that we

touched upon in the last chapter. Whenever an iPhone application gets entirely loaded into memory, it'll send an `applicationDidFinishLaunching:` message to your application delegate, running that method. You'll note there's already some code to display that Interface Builder created window #5.

For this basic project, you'll add all your new code to this same routine—such as an object that says Hello, World!

Writing Hello, World!

We've been promised for a while that you're going to be amazed by how simple it is to write things using the SDK. Granted, your Hello, World! program may not be as easy as a single `printf` statement, but nonetheless it's pretty simple considering that you're dealing with a complex, windowed UI environment.

As promised, you'll be writing everything inside the `applicationDidFinishLaunching` method, as shown in listing 4.

Listing 4 The iPhone Presents ... Hello, World!

```

- (void)applicationDidFinishLaunching:(UIApplication *)application {
    [window setBackgroundColor:[UIColor redColor]];           1
    CGRect textFieldFrame = CGRectMake(50, 50, 150,40);      2
    UILabel *label = [[UILabel alloc] initWithFrame:textFieldFrame]; 3
    label.textColor = [UIColor whiteColor];                 4
    label.backgroundColor = [UIColor redColor];              4
    label.shadowColor = [UIColor blackColor];                4
    label.font = [UIFont systemFontOfSize:24];               4
    label.text = @"Hello, World!";                           4
    [window addSubview:label];                               5
    [window makeKeyAndVisible];                              6
    [label release];                                        7
}
1 Set background color
2 Define rectangular frame
3 Create label in frame
4 Change label properties
5 Add label to window
6 Make window visible
7 Free up label's memory

```

Since this is your first look at real live Objective-C code, we're going to examine everything in some depth.

About the Window

You start off by sending a message to your window object, telling it to set your background to red #1. You'll recall from our discussion of the header file, that Interface Builder was what actually created the window. The IBOutlet that was defined in the header is what allows you to do manipulations of this sort.

Note that this line also makes use of a nested message, which we promised we'd see with some frequency. Here, you make a call to the UIColor class object and ask it to send you the red color. You then pass that on to your window.

Over the course of this book, we're going to hit a lot of UIKit classes without explaining them in depth. That's because the simpler objects all have pretty standard interfaces; the only complexity is in which particular messages they accept. If you ever feel like you need more information about a class, you should look at appendix 1, which contains short descriptions of many objects, or in the complete class references available online at developer.apple.com (or in Xcode).

About Frames

You're next going to need to define where your text label is placed. You start that process off by using CGRectMake to define a rectangle #2. Much as with Canvas, the SDK uses a grid with the origin (0,0) set at the top left. Your rectangle's starting point is thus 50 to the right and 50 down (50,50) from the origin. The rest of this line of code sets the rectangle to be 150 pixels wide and 40 tall, which is enough room for your text.

You're going to be using this rectangle as a "frame," which is one of the methods that you can use to define a view's location.

Choosing a View Location

Where your view goes is one of the most important parts of your view's definition. Many classes use an initWithFrame: init method, inherited from UIView, which defines location as part of the object's setup.

The frame is simply a rectangle that you've defined with a method like CGRectMake. Another common way to create a rectangular frame is to set it to take up your full screen:

```
[[UIScreen mainScreen] bounds];
```

Sometimes you'll opt not to use the initWithFrame: method to create an object. UIButton is an example of a UIKit class that instead suggests you use a class factory method that lets you define a button shape.

In a situation like that you must set your view's location by hand. Fortunately this is easy to do because UIView also offers a number of properties that you can set to determine where your view goes, even after it's been initialized.

UIView's `frame` property can be passed a rectangle, just like the `initWithFrame:` method. Alternatively you can use its `center` property to designate where the middle of the object goes and the `bounds` property to designate its size internal to its own coordinate system.

All three of these properties are further explained in the UIView class reference, while an example of button placement appears in chapter 14.

Note that `CGRectMake` is a function, not a method. It takes arguments using the old, unlabeled style of C, rather than Objective-C's more intuitive manner of using labeled arguments. Once you get outside of Cocoa Touch, you'll find that many frameworks use this older paradigm.



About The Label

The label is a simple class that allows you to print text on the screen. We included figure 1 nearby to really show off what your label (and the rest of your program) looks like.

Figure 1 Hello, World! on the iPhone, using the SDK.

As you'd expect, your label work begins with the actual creation of a label object #3. Note that you follow the standard methodology of nested object creation that we introduced in the previous chapter. First you use a class method to allocate the object, and then you use an instance function to initialize it.

Afterward you send a number of messages to your object #4, this time using the dot shorthand. This was offered as a purposeful variation from the way you set the window's background color. If you'd preferred, you could have used the dot shorthand of `window.backgroundColor` there, too. The two ways to access properties are totally equivalent.

The most important of your messages sets the label's text. However you also set a font size and some colors. You even can give the text an

attractive back shadow, to really show off how easy it is to do cool stuff using the iPhone OS's objects.

Every object that you use from a framework is going to be full of properties, methods, and notifications that you can take advantage of. The best place to look all these up in is the class references that we continue to allude to.

Finishing Up Our World

The final steps in your program are all pretty simple and standard.

First, you connect up your label and your window by using the window's `addSubview` method #5. This is a standard (and important!) method for adding views or view controllers to your window. You'll see it again and again.

Second, you actually create your window on the screen, using the line of code that was here when we started #6. Making the window "key" means that it's now the prime recipient of user input (for what that's worth in this simple example), while making it "visible" means that the user can see it.

Third, you remember the standard rule that you must release anything you allocated. Here, that's just the label #7.

And that's a simple Hello, World! program, completely programmed and working, with some neat graphical nuances.