Master Cocoa Touch frameworks
with Xcode and Objective-C

Learn Cocoa Touch for iOS

Jeff Kelley

DETROIT LABS

Apress®

Learn Cocoa Touch for iOS

Jeff Kelley

Apress®

Learn Cocoa Touch for iOS

ISBN 978-1-4302-4269-7

ISBN 978-1-4302-4270-3 (eBook)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code`.

To my wife, Amanda. I love you.

Contents at a Glance

Contents

About the Author

■ **Jeff Kelley** started programming for the iPhone with iPhone OS 2 and has seen it evolve into the iOS we know and love today. Jeff has developed dozens of apps for clients both large and small in a wide variety of industries, as well as several apps for his own use. He's been programming since using BASIC in grade school, with his professional start coming in the Mac IT world in education. Today he does iOS programming full-time, as well as speaking engagements at conferences and the local chapter of CocoaHeads.

About the Technical Reviewer

■ **Scott Gardner** is an Apple technology evangelist, consultant, and developer. He combines insight gained from the field and continuous study of iOS to develop apps that are beneficial and intuitive. Scott resides in the Midwest with his wife and daughter.

Acknowledgments

I'd like to thank everyone at Apress for their hard work on this book—specifically, Scott Gardner as my technical reviewer, who made a number of suggestions that improved the quality of the work, as well as keeping me honest with providing top-tier content. Douglas Pudnick, my development editor, also did a great job guiding the direction of the content, and the other editors, Michelle Lowman, Jennifer Blackwell, Kelly Moritz, and Jill Balzano, were all a great help with this book. I'd also like to thank the Detroit Labs crew, specifically cofounders Paul Glomski, Henry Balanon, Nathan Hughes, and Dan Ward, for being understanding with the challenges of writing apps by day and a book by night, as well as for giving me an awesome place to work. At home, a great deal of thanks goes to my wife, Amanda, who put up with me writing this book while also pregnant with our first child, something she deserves endless credit for. Finally, I'd like to thank my parents for helping me get my start in programming; my mom would copy BASIC programs from a book when I was in kindergarten so that I could mess with them when I got home, and later in grade school my dad bought me a copy of Visual Studio—the first app I ever wrote for someone else was a retirement calculator he could keep on his screen at work to count down the seconds until retirement. I've had a lot of help along the way in getting this book published, so to everyone, mentioned here or not, thank you.

Introduction

With every successive release of iOS and its related hardware products, Apple and journalists the world over spout hyperbolic statements about "revolutionary" features, "insanely great" devices, and "unbelievable" sales. The numbers don't disappoint, with hundreds of millions of iOS devices having been sold and billions of dollars sent to developers in revenue. As we enter the post-PC era, we do so using our smartphones and tablets. Apple's iOS is consistently the most user-friendly, powerful platform for these new devices, and developers the world over benefit from offering their products on the App Store. That being said, it is a market that continues to grow every day, especially when customers can obtain an iPhone for next to nothing up front with a two-year contract. As the barrier to entry to the smartphone market declines and the user base goes up, opportunity skyrockets. This book will allow you to take advantage of that opportunity. We'll get up and running using Xcode on Mac OS X, we'll create applications as we learn Objective-C (the language in which you'll be developing your apps), and we'll tour the frameworks that make Cocoa Touch one of the best development environments in the world.

As you should get used to when programming for an Apple environment, there are rules. As such, there are some things you'll need to go through this book: a Mac with an Intel processor running Mac OS X 10.7 (Lion) or newer, with Xcode 4.3 or newer (available from the Mac App Store), and ideally an iOS device running iOS 5.1 or newer. While older versions of Mac OS X, Xcode, and iOS may still be in use, screenshots and step-by-step instructions in this book may not work for other versions.

Who This Book Is For

This book assumes a basic level of programming knowledge. You don't have to be an expert, but any experience you have with C, C++, or even Java will be useful to help frame concepts explained in the early stages of the book. You should also be familiar with the basics of Apple's Mac OS X and iOS operating systems, enough to get around the filesystem in Mac OS X and launch Xcode and enough to launch apps and understand typical app behavior on iOS.

How This Book Is Structured

In general, chapters in this book will begin with more abstract concepts. Where there has been evolution in the development frameworks and libraries, we'll start with the older, more complicated ways and lead in to the newer way of doing things in order to better understand why things have developed the way they have. As each chapter progresses, we'll switch from the abstract to the concrete, with sample projects and example code. We'll develop two apps in multiple chapters, with other smaller examples in addition.

Chapter 1 gets you up and running with Xcode and creating a "Hello, World!" app.

Chapter 2 covers the Objective-C language in detail, including memory management, best practices, and the latest additions to the language.

Chapter 3 discusses working with view controllers, one of the most important types of objects you'll use in iOS development.

Chapter 4 covers handling your data, from moving it around inside the app to saving and loading from disk.

Chapter 5 details handling user touches and basic app flow.

Chapter 6 covers networking and web services, including parsing JSON and XML.

Chapter 7 introduces blocks, Apple's new addition to the C language that encapsulates code.

Chapter 8 explains more about the message dispatch process in iOS, leading to a discussion of multithreaded code.

Chapter 9 covers user interface design in your app.

Chapter 10 details the multitude of hardware APIs available on iOS devices, including the accelerometer, gyroscope, and magnetometer, as well as location services using GPS.

Chapter 11 outlines using media in your app, both audio and video.

Chapter 12 covers the internationalization and localization processes, which help give your app a broader reach.

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, `www.apress.com`. You can find a link on the book's information page under the Source Code/Downloads tab. This tab is located underneath the Related Titles section of the page.

Contacting the Author

Send your questions, comments, criticisms, and lame puns (*especially* lame puns) to me on Twitter as `@SlaunchaMan` or by e-mail at `SlaunchaMan@gmail.com`. Read my blog at `http://blog.slaunchaman.com`, and check out my professional work at `www.detroitlabs.com`.

Chapter 1

Getting Started

While apps for your iPhone are a relatively new phenomenon, they're based on decades-old technologies present also on your Mac. Mac OS X introduced a new set of APIs and frameworks collectively known as Cocoa. While iOS shares many lower-level system frameworks and APIs with Mac OS X, the APIs relating to its touch-based user interface, telephone capabilities, and iOS-only functionality reside in the Cocoa Touch layer, an analog to Cocoa for mobile devices. One of the similarities Cocoa Touch has with its desktop counterpart is the tools used for development, including the same IDE, Xcode. In fact, SDKs for iOS and Mac OS X development are included when you download the developer tools. In this chapter, we'll take a closer look at these tools and get started using them.

Installing Xcode

Before you get started writing your applications, you'll need to install Apple's developer tools. While there are many individual applications, libraries, and utilities you'll use over the course of app development, the main one you'll use is Apple's IDE, Xcode.

NOTE: Unlike the iPhone and other Apple products, the leading X in Xcode is capitalized.

There are two ways to install Xcode. The easiest, best-supported, and most up-to-date way is to download Xcode from the Mac App Store. When the download finishes, Xcode will be in your `/Applications` directory, with no further installation required.

NOTE: By default, the Xcode installer installs developer tools to the `/Applications` folder on your hard drive. It is possible to install Xcode to a different location, but recent versions of the installer have not exposed that option to users. I recommend installing the App Store version of Xcode to `/Applications` and installing any beta versions you may use to other folders.

The second way to install Xcode is by downloading an installer from Apple's developer site. While Apple doesn't always release each final shipping version of Xcode this way, this is how you'll install prerelease versions of the tool set. Once you log in with your developer credentials, you'll download a disk image containing an Installer package for the developer tools. Run that package to install Xcode. As of this writing, the latest version of Xcode is 4.3; while older versions may work on your Mac, versions older than 4.0 are significantly different, enough so that it may be difficult to follow along with the tutorials in this book.

Either way, you should know going in that Apple's tool set is a large download, usually more than several gigabytes. There has been some progress on separating individual components into something that Xcode can update without redownloading the whole set of tools, but the initial download is something you probably can't do at your local coffee shop.

The Developer Tools

The developer tools you've installed center around Xcode, but there are some other components that you'll use a lot over the course of this book:

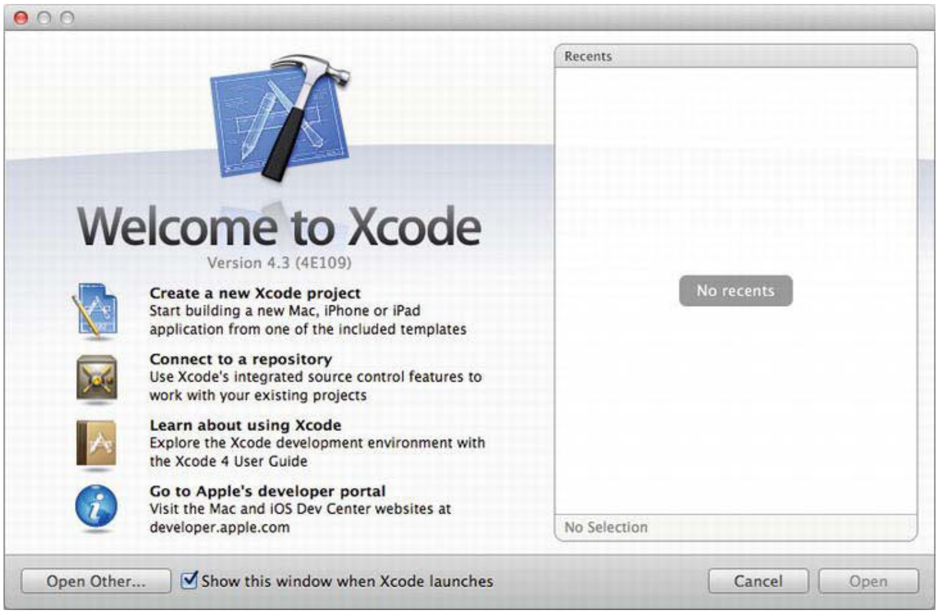
- Instruments allows you to inspect the performance of your application, finding memory leaks, discovering computational bottlenecks, and even breaking down the 3D rendering of games with ease.

- The iOS Simulator runs your iOS applications in a simulated environment. It's important to note the difference between a *simulator* and an *emulator*. In a simulator, your code is compiled for the platform the *simulator* is running on. In the case of an iOS app, the code is compiled for your Mac and runs in a fake, iPhone-*like* environment. In an emulator, the code is compiled the same for the emulator and the platform you're writing *for*. There is no iOS emulator available, but if there were, code compiled for the emulator would be the same as code compiled for the device. This is important in testing because the processor architectures are different on different platforms; your Mac has an Intel processor, but an iPhone has an ARM processor. For this reason, you should always test on the device before releasing an app to ensure that there aren't any device-specific bugs.
- Xcode allows you to download local copies of the entire documentation set usually available at `http://developer.apple.com`; this documentation allows you to see help inline in Xcode while you write.
- Finally, the tools include compilers, linkers, and other tools needed to turn your code into an actual, functioning application. If you're comfortable with the command line, you can now use `gcc` and related tools to compile applications. Xcode 4 replaced GCC with Clang running on the LLVM infrastructure, a more modern compiler and the new default. For most cases, LLVM can replace GCC with no loss in functionality—in fact, the `gcc` command-line utility is really just a symlink to LLVM in recent tool set distributions.

To get started, launch Xcode. By default, the path will be `/Applications/Xcode.app`. With Xcode installed and launched, let's make our first application.

Hello, World!

When you first start Xcode, you'll see a welcome screen (Figure 1-1). From here, you can open recent projects, launch Apple's developer web site, open the Xcode user guide (which you should definitely read at some point), download source code from a revision control system, and create a new project. Since we haven't created one yet, click "Create a new Xcode project."



Create a new Xcode project
Start building a new Mac, iPhone or iPad application from one of the included templates

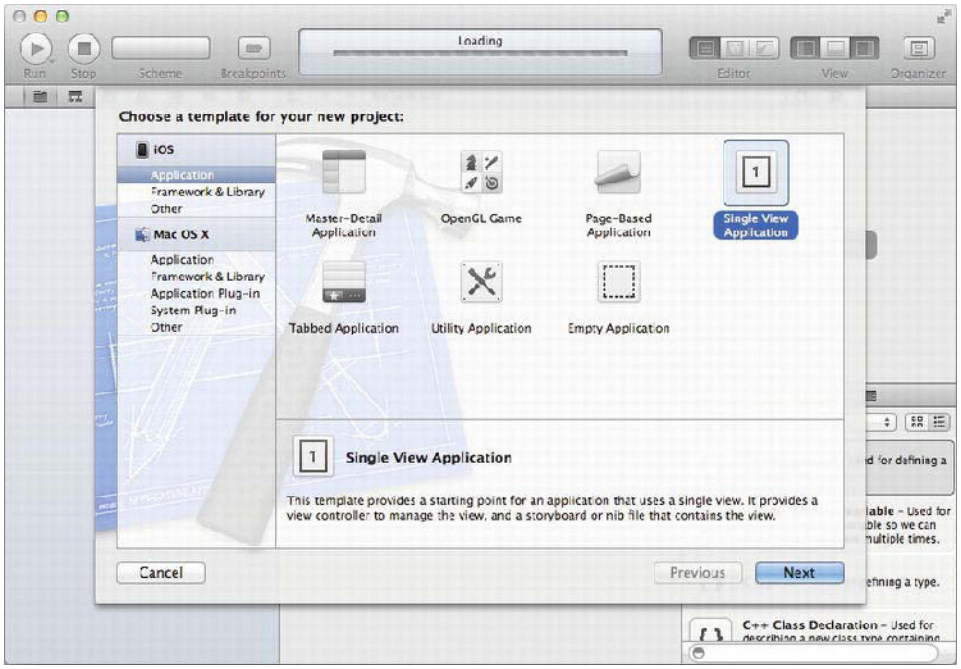
Connect to a repository
Use Xcode's integrated source control features to work with your existing projects

Learn about using Xcode
Explore the Xcode development environment with the Xcode 4 User Guide

Go to Apple's developer portal
Visit the Mac and iOS Dev Center websites at developer.apple.com

Figure 1-1. *The Xcode welcome screen*

When you create a new project, Xcode presents a wizard, seen in Figure 1-2, that starts with a list of the types of projects it can make. Xcode uses templates to speed the development of common types of applications. On the left, you can see the categories of templates that are currently installed. If it isn't already selected, select Application under iOS on the left to display all of the iOS templates. Our simple application will have only one screen, so select Single View Application and click Next.



Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

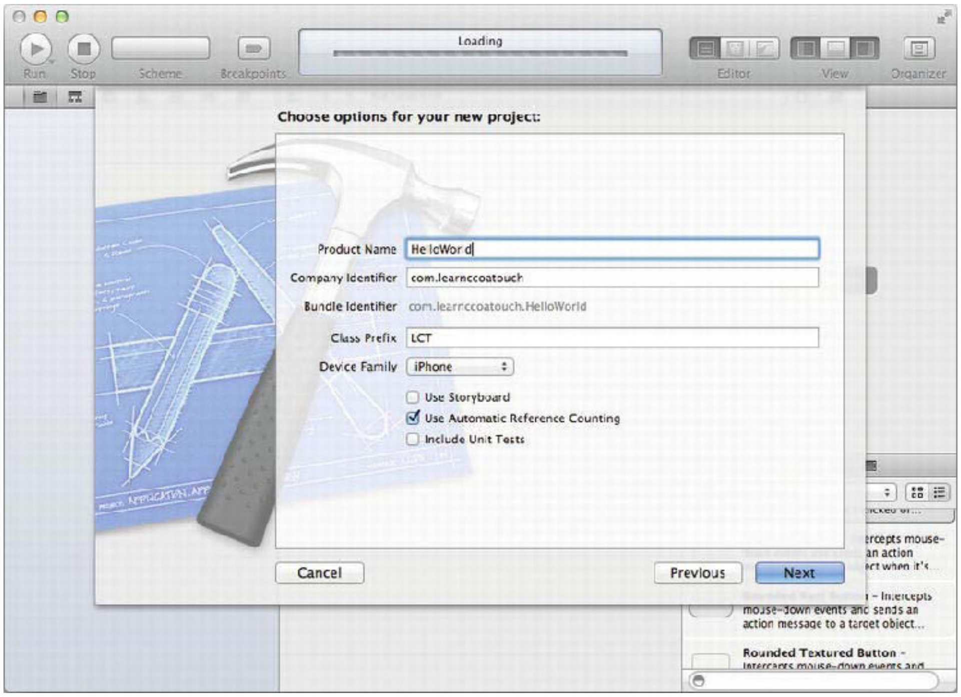
Figure 1-2. *Selecting a template from the Xcode New Project Wizard*

The next screen gives you some options to set the metadata for the project and to further refine which template Xcode uses. Since this is our first project, we'll create a "Hello, World!" iOS application. "Hello, World!" is a tradition nearly as old as programming itself wherein the first thing you do in a new language or on a new platform is make a program that displays the words "Hello, World!" to the user. Enter **HelloWorld** for Product Name. The Company Identifier value should be a reverse-DNS label for your company name (if you have one). If you don't have one, your personal web site will do. If you don't have one, consider getting one before releasing any apps to the App Store.) Since my web site is at `http://learncocoatouch.com`, I use `com.learncocoatouch` as my company identifier. This reverse-DNS style listing is used often in iOS to differentiate between applications and other identifiable things, typically with your application ID affixed to the end. For me, the HelloWorld project has the identifier of `com.learncocoatouch.HelloWorld`. App IDs must be unique in the App Store, and installing an app on a device with the same ID as another app will overwrite the existing one.

The class prefix is used to identify code that you create and differentiate it from code that others write. Typically you'll use your initials. This is important to ensure that two developers don’t create things with the same name. If your initials happen to be the same as another developer’s or what a system framework uses for a prefix, you can use three letters, letters from your company name, or any combination of letters you like. For *Learn Cocoa Touch*, I’ll use LCT.

> **NOTE:** You can find an unofficial list of “claimed” prefixes at `www.cocoadev.com/index.pl?ChooseYourOwnPrefix`. Claim yours now!

The next options affect the template that the project will use. Leave Device Family set to iPhone for now. If you’re creating an app for iPad or a Universal app that supports both devices, this is where you set it. Uncheck Use Storyboard and Include Unit Tests, but check Automatic Reference Counting. We’ll go over what those mean in more detail later. Once those are set, we’re finally ready to create our application. Your screen should look like Figure 1-3. Click Next.



Choose options for your new project:

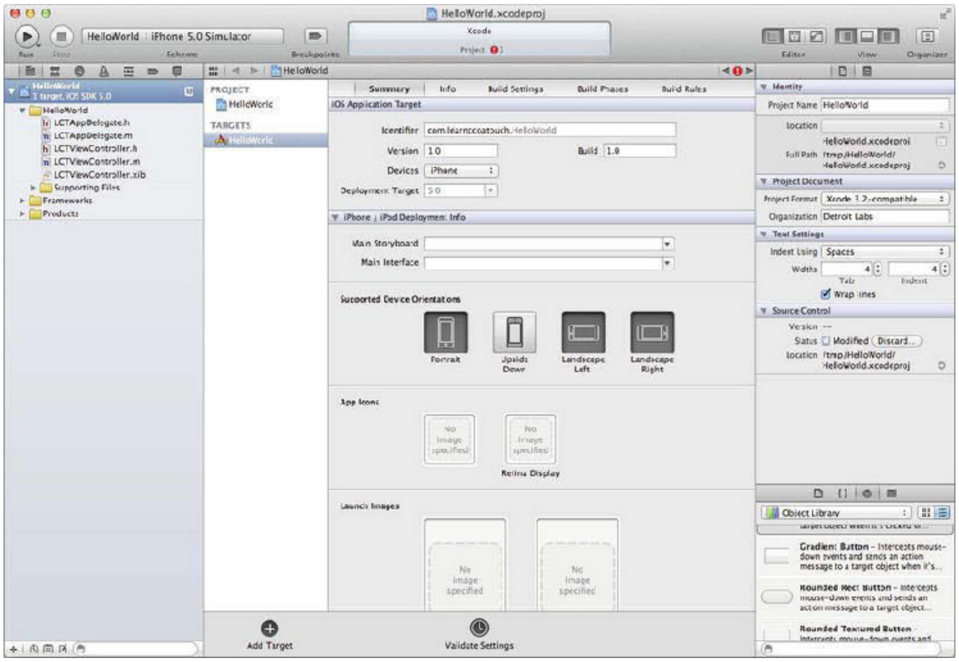
Product Name: HelloWorld
Company Identifier: com.learncocoatouch
Bundle Identifier: com.learncocoatouch.HelloWorld
Class Prefix: LCT
Device Family: iPhone

☐ Use Storyboard
☑ Use Automatic Reference Counting
☐ Include Unit Tests

Figure 1-3. *Choosing options for the new project*

Xcode will prompt you to select a location for the project on your hard disk, as well as give you the option to create a local Git repository while it creates the project. If you know and use Git, feel free to select that option; otherwise, it's unneeded for this project. While going through this book, you may find it useful to create a separate directory somewhere in your Home folder for the various apps we'll be writing, such as `~/Projects/Learn Cocoa Touch/`.

Once you select a location, Xcode creates your project. The initial screen, shown in Figure 1-4, shows you your project settings. Here we can modify project metadata such as supported resolutions, which iOS version(s) the project will run on, the version number of the application, which device orientations it supports, the icons to use, and so on. We'll leave these alone for now.



iOS Application Target

Identifier: com.learncocoatouch.HelloWorld
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.0

iPhone / iPod Deployment Info

Main Storyboard:
Main Interface:
Supported Device Orientations
App Icons
Launch Images

Identity

Project Name: HelloWorld
Location: HelloWorld.xcodeproj
Full Path: /tmp/HelloWorld/HelloWorld.xcodeproj

Project Document

Project Format: Xcode 3.2-compatible
Organization: Detroit Labs

Text Settings

Indent Using: Spaces
Widths: 4 (Tab) 4 (Indent)
✓ Wrap lines

Source Control

Version: --
Status: Modified
Location: /tmp/HelloWorld/HelloWorld.xcodeproj

Gradient Button – Intercepts mouse-down events and sends an action message to a target object when it's...
Rounded Rect Button – Intercepts mouse-down events and sends an action message to a target object...
Rounded Textured Button – Intercepts mouse-down events and

Figure 1-4. *This is the initial layout of the Xcode window once you've created a project.*

To run your application in the iOS Simulator, click the Run button at the upper-left corner of the Xcode window (the one that looks like the iTunes Play button). Since we haven't modified the code at all, it won't look like much. Figure 1-5 shows what you should see at this point when you run your app.

NOTE: If the text to the right of the Run button says iOS Device, change the selection to the iPhone Simulator.

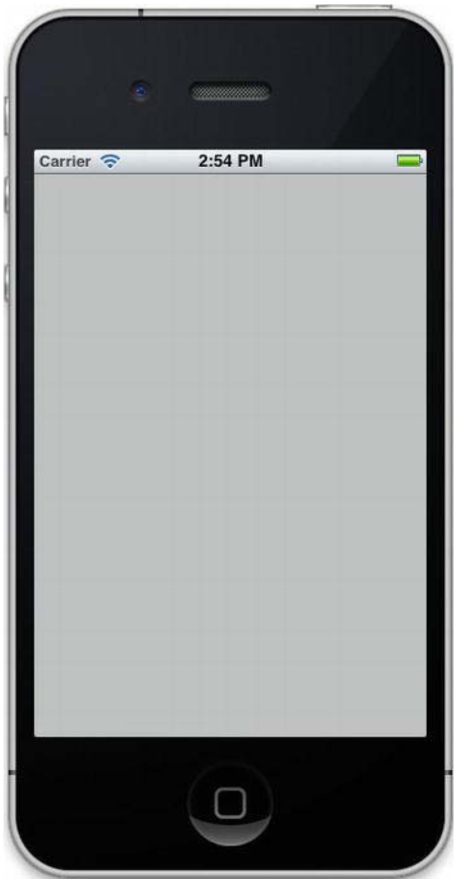


Figure 1-5. *Our first iOS app running in the simulator*

Now that we have the application set up and ready to modify, let's take a look at our goal for this application:

Goal: Build an app that says "Hello, world!" to the user.

Ready to modify the app? Good. Quit the iOS Simulator and head back to Xcode. Press Command+1 to open the File browser on the left pane. Find the file under `HelloWorld` that ends in `ViewController.xib` and select it. Note that it will start with your class prefix—in my case, it's called `LCTViewController.xib` by default. The file will open in an Interface Builder view: a visual layout of your application's interface. Right now, it's the same gray view that you saw in the iOS Simulator. Let's change that. The bottom-right corner of the screen contains the Object Library, a collection of user interface elements that you can add to the view. You can switch to its search field by pressing Control+Option+Command+3. Figure 1-6 shows what your screen should look like with the Object Library visible.

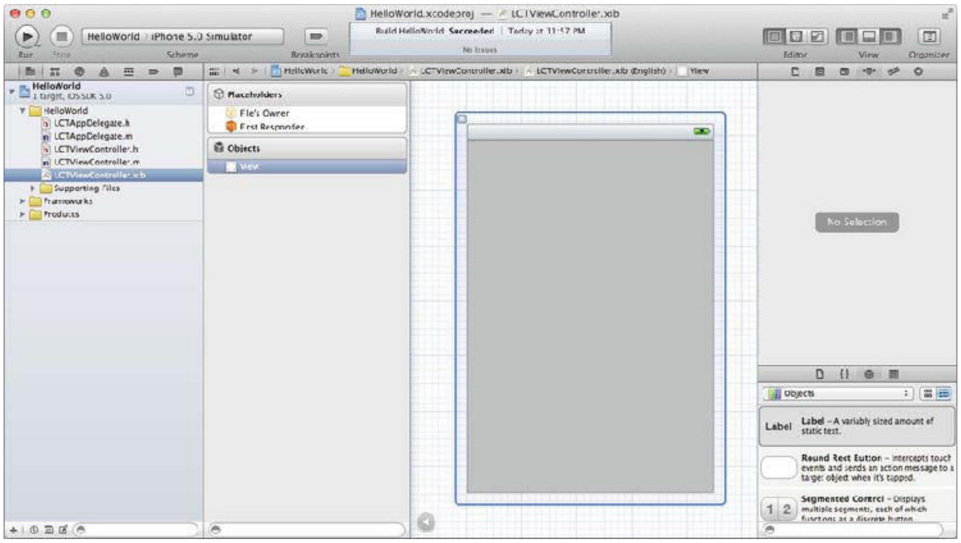

HelloWorld.xcodeproj — LCTViewController.xib
Run
Stop
HelloWorld › iPhone 5.0 Simulator
Scheme
Breakpoints
Build HelloWorld: Succeeded | Today at 11:17 PM
No Issues
Editor
View
Organizer
HelloWorld
1 target, iOS SDK 5.0
HelloWorld
LCTAppDelegate.h
LCTAppDelegate.m
LCTViewController.h
LCTViewController.m
LCTViewController.xib
Supporting Files
Frameworks
Products
Placeholders
File's Owner
First Responder
Objects
View
No Selection
Objects
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.

Figure 1-6. *The Xcode window using Interface Builder with the object library visible.*

To add an object to your view, either drag it from the Object Library to your view or double-click it. Drag two objects to your view: a Label and a Round Rect Button. Double-click the button to add a title; let's make this one read "Say Hello." Notice that the button resizes itself when you add the title. You can get labels and buttons to resize themselves to their content by pressing Command+=. Double-click the label and remove the text, and then make it stretch across the view. Once you remove the text, the label will appear to be invisible; if you can't find it, click Editor ➤ Canvas ➤ Show Bounds Rectangles, which will outline the label for you. When you're done, it should look something like Figure 1-7. If so, now is a good time to save your work. Xcode isn't perfect, and if it crashes, your unsaved changes go with it, so getting into a habit of saving often is recommended.

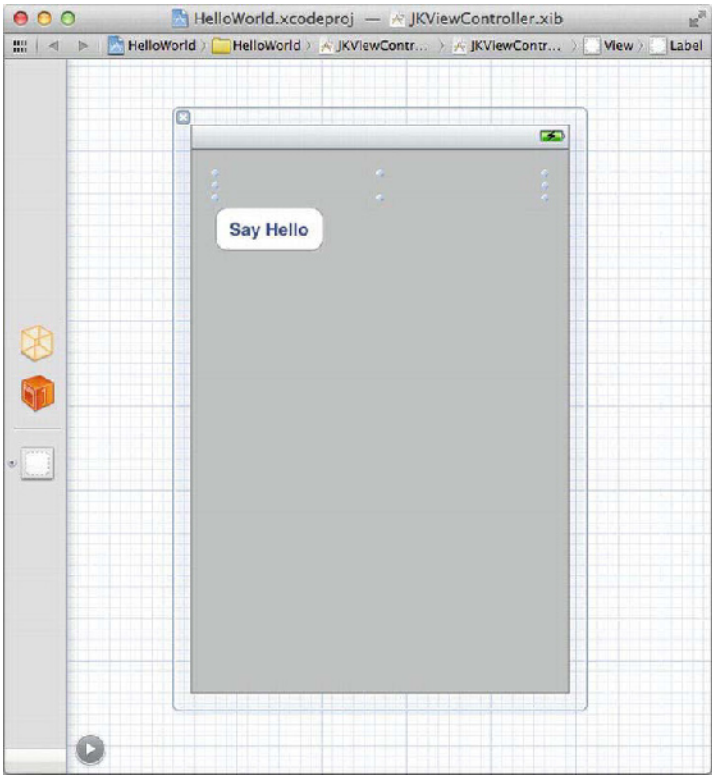


Figure 1-7. *The view set up for our "Hello, World" application*

Now let's add some code to this application. We want the label to say "Hello, World!" when the user presses the button. To do that, we'll add a *method* to our view controller. *Method* is Objective-C's word for function. If you're familiar with object-oriented programming, then methods will be familiar. If not, follow along in this chapter; we will discuss Objective-C later in much more detail.

The view controller's *header file* is a file that *describes* it. Headers are the "public" portion of your code; they describe what the code will do without actually showing how it works. When you receive source code that's already been compiled, typically you'll also receive the headers associated with it. In the file browser, select the file ending in `ViewController.h` with your prefix before it. In the header, we define the methods that we will create. By default, it should look like this (with some comments at the top):

```
//
//  LCTViewController.h
//  HelloWorld
//
//  Created by Jeff Kelley on 1/28/12.
//  Copyright (c) 2012 Jeff Kelley. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface LCTViewController : UIViewController

@end
```

The first part of creating a method is *declaring* it, that is, telling the code that there will be a method. So, add this line between the `@interface` and `@end` lines and save your changes:

```
- (IBAction)sayHelloButtonPressed:(id)sender;
```

We'll go into more detail later on what each part of this line means. For now, you should know that the name of the method is `sayHelloButtonPressed:`. Now that we've declared it, we can go back to the view and tell our app to run our method when the button is pressed. Head back to the view by opening `LCTViewController.xib` and select the button. Open the right utilities pane to the Connections Inspector, either by clicking the rightmost icon at the top of the pane or by pressing Command+Option+6. You'll see a list of empty circles on the right side of the list under Sent Events. We're interested in the event Touch Up Inside. These events represent different points of interaction the user has with the button. When they first place their finger on the button, the Touch Down event occurs, and when they lift it, the Touch Up Inside event occurs. Typically on iOS, we use the Touch Up Inside event for user interaction; that way, the user can cancel pressing the button by moving their finger away.

To connect the Touch Up Inside event to the method we created, click the empty circle next to it and drag. We're connecting it to the object called File's Owner, which looks like a transparent box and is to the left of our view. With File's Owner highlighted, release the mouse button, and a list of methods will pop up. The method we created should be the only one in the list. Select it, and the button is now connected to the method. It should look like Figure 1-8.

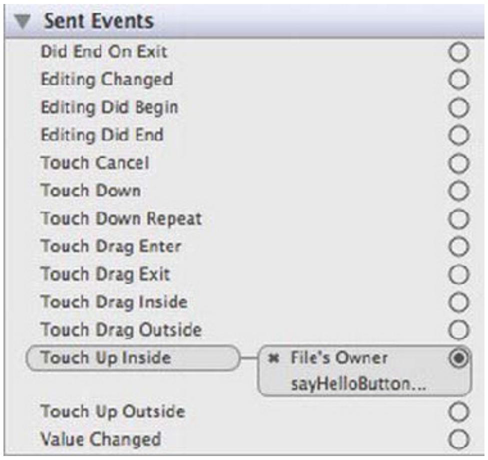


Figure 1-8. *The Connections Inspector view after we've connected the button to the method*

The next step is writing the code that will happen when we press the button. First, we need to create a way to get to the label from our code. Much like creating the method, we'll modify the header first and then connect the view to it. Modify the header to add this line:

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

@interface JKViewController : UIViewController {
        IBOutlet UILabel *helloWorldLabel;
}

- (IBAction)sayHelloButtonPressed:(id)sender;

@end
```

Now, we need to connect the label in our view to the IBOutlet we created. Select the label in your view, and then open the Connections Inspector. Drag the circle next to New Referencing Outlet to File's Owner and select `helloWorldLabel`. Now that we've done that, we can use `helloWorldLabel` in our code to refer to the label.

We have everything set up for our method, so let's create it. We define our methods in the view controller's *implementation* file, which ends in `.m`. Open the file and add the lines in bold:

```
#import "JKViewController.h"

@implementation JKViewController

// Other methods will be defined here
```

```
- (IBAction)sayHelloButtonPressed:(id)sender
{
        [helloWorldLabel setText:@"Hello, World!"];
}

@end
```

This code calls a method on your label, `setText:`, with the text "Hello, World!" Now that we've *implemented* our method, click Run again to run the application. Xcode will build the app and run it in the iOS Simulator. You'll see the button. Click it, and the label will say "Hello, World!"

Summary

While creating a "Hello, World!" app is an important beginner's task in any language, it's not going to sell too many copies in the App Store. It doesn't really access too many features of the device, and it doesn't push the envelope with an engaging user interface. It's a good step toward making a quality app, however, and that's what counts. In this chapter, we covered installing and using Xcode, as well as the beginnings of using it for programming. Now that we've created a simple app in Xcode, let's learn more about Objective-C, the programming language we'll be using throughout the book.

Chapter 2

Objective-C in a Nutshell

Objective-C is the primary language you'll be using to create iOS apps using Cocoa Touch. This chapter will walk you through the basics of the language, covering new developments in its evolution as well as tried-and-true methods that are decades old. In this book, I'm assuming that you have at least a basic understanding of the C programming language. If you're coming from a Java or C++ background, you can probably get by just fine, but if you're new to C-like languages altogether, I recommend familiarizing yourself with it. Some excellent books on the subject are *The C Programming Language* by Brian Kernighan and the late Dennis Ritchie, who originally designed the language; *Programming in C* by Stephen Kochan; *C Programming* by K. N. King; and *Learn C on the Mac* by Dave Mark.

Object-Oriented Programming

Objective-C is an object-oriented language, as are Java and C++, but Objective-C is unique in that it is a *superset* of C; that is, anything that is valid in C is also valid in Objective-C. C++ gets close, but not quite there. This means that if you already have code written in C, you can use it as is for iOS. You can also use existing C data structures, functions, and preprocessor macros. The more interesting parts, however, are those that Objective-C adds to turn C into an object-oriented programming language.

An *object* in Objective-C is used much like other data types (integers, floating-point values, characters, and so on) in C, but typically you'll use a pointer to refer to it. The following line is an example of creating an object in Objective-C:

```
NSString *myString = @"Hello, World!";
```

In that line, we created the object `myString`. Its *class*, or the *kind* of object it is, is `NSString`. `myString` is an *instance* of `NSString`. The asterisk (*) signifies that we're creating a pointer—technically speaking, `myString` isn't the object itself but rather a pointer *to* an instance of `NSString`.

> **NOTE:** We created `myString` as a constant string. The @ followed by a string in quotes signifies this to the compiler.

To declare a class, use the following syntax:

```
@interface ClassName : SuperClassName
```

The `@interface` is a *compiler directive*—that is, a special command to the compiler that gives it instructions on how to compile your code. In this case, `@interface` begins the class definition for a class. The `SuperClassName` is the name of another class from which the class you're creating will *inherit* variables and methods. The root object for most of the objects you'll create is `NSObject` (the *NS* stands for NeXTStep, NeXT's operating system). While there are technically other base classes, you're free to create your own. For now we'll use `NSObject`; it contains many functions that Cocoa Touch relies on.

> **NOTE:** The reason the *NS* prefix remains from NeXTStep has to do with the history of Mac OS X. Apple purchased NeXT Software, Inc., in 1996, and the NeXTStep operating system formed the basis of Mac OS X, introduced in 2001. iOS shares many of its system-level frameworks, including Objective-C and the Foundation framework, which contains `NSObject` and other essential classes, with Mac OS X, thereby inheriting the shared legacy of NeXTStep's *NS* prefix. One advantage of this is that in most cases, classes that begin with an *NS* prefix are also available on the Mac, so if you're interested in programming in Cocoa (the Mac OS X equivalent of Cocoa Touch), learning Cocoa Touch is a great first start.

To help explain this, we'll work toward a goal instead of talking in the abstract the whole time. Our goal is going to be to create an address book. Let's create a class that represents an entry in the address book. Each entry corresponds to an individual person, so we'll name the class `Person`: