

6TH EDITION

iOS Programming

THE BIG NERD RANCH GUIDE

Christian Keur and Aaron Hillegass

iOS Programming: The Big Nerd Ranch Guide

by Christian Keur and Aaron Hillegass

Copyright © 2016 Big Nerd Ranch, LLC

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, contact

Big Nerd Ranch, LLC
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
<http://www.bignerdranch.com/>
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, LLC.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
<http://www.informit.com>

The authors and publisher have taken care in writing and printing this book but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

App Store, Apple, Cocoa, Cocoa Touch, Finder, Instruments, iCloud, iPad, iPhone, iPod, iPod touch, iTunes, Keychain, Mac, Mac OS, Multi-Touch, Objective-C, OS X, Quartz, Retina, Safari, and Xcode are trademarks of Apple, Inc., registered in the U.S. and other countries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

ISBN-10 0134687442
ISBN-13 978-0134687445

Sixth edition, second printing, July 2017
Release D.6.1.1

Acknowledgments

While our names appear on the cover, many people helped make this book a reality. We would like to take this chance to thank them.

- First and foremost we would like to thank Joe Conway for his work on the earlier editions of this book. He authored the first three editions and contributed greatly to the fourth edition as well. Many of the words in this book are still his, and for that, we are very grateful.
- Juan Pablo Claude wrote some of the content and contributed his expertise and opinions to make this book even better. His work is greatly appreciated.
- A couple other people went above and beyond with their help on this book. They are Mikey Ward and Chris Morris.
- The other instructors who teach the iOS Bootcamp fed us a never-ending stream of suggestions and corrections. They are Ben Scheirman, Bolot Kerimbaev, Brian Hardy, Chris Morris, JJ Manton, John Gallagher, Jonathan Blocksom, Joseph Dixon, Juan Pablo Claude, Mark Dalrymple, Matt Bezark, Matt Mathias, Mike Zornek, Mikey Ward, Pouria Almassi, Robert Edwards, Rod Strougo, Scott Ritchie, Step Christopher, Thomas Ward, TJ Usiyan, Tom Harrington, and Zachary Waldowski. These instructors were often aided by their students in finding book errata, so many thanks are due to all the students who attend the iOS Bootcamp.
- Thanks to all of the employees at Big Nerd Ranch who helped review the book, provided suggestions, and found errata.
- Our tireless editor, Elizabeth Holaday, took our distracted mumblings and made them into readable prose.
- Anna Bentley and Simone Payment jumped in to provide copyediting and proofing.
- Ellie Volckhausen designed the cover. (The photo is of the bottom bracket of a bicycle frame.)
- Chris Loper at IntelligentEnglish.com designed and produced the print and ebook versions of the book.
- The amazing team at Pearson Technology Group patiently guided us through the business end of book publishing.

The final and most important thanks goes to our students, whose questions inspired us to write this book and whose frustrations inspired us to make it clear and comprehensible.

Table of Contents

Introduction	xiii
Prerequisites	xiii
What Has Changed in the Sixth Edition?	xiii
Our Teaching Philosophy	xiv
How to Use This Book	xv
How This Book Is Organized	xv
Style Choices	xvii
Typographical Conventions	xvii
Necessary Hardware and Software	xvii
1. A Simple iOS Application	1
Creating an Xcode Project	2
Model-View-Controller	5
Designing Quiz	6
Interface Builder	7
Building the Interface	8
Creating view objects	9
Configuring view objects	12
Running on the simulator	13
A brief introduction to Auto Layout	14
Making connections	17
Creating the Model Layer	22
Implementing action methods	23
Loading the first question	23
Building the Finished Application	24
Application Icons	25
Launch Screen	28
2. The Swift Language	29
Types in Swift	30
Using Standard Types	31
Inferring types	33
Specifying types	33
Literals and subscripting	35
Initializers	36
Properties	37
Instance methods	37
Optionals	38
Subscripting dictionaries	40
Loops and String Interpolation	40
Enumerations and the Switch Statement	42
Enumerations and raw values	43
Exploring Apple’s Swift Documentation	44
3. Views and the View Hierarchy	45
View Basics	46
The View Hierarchy	46
Creating a New Project	48

Views and Frames	49
Customizing the labels	57
The Auto Layout System	60
The alignment rectangle and layout attributes	60
Constraints	61
Adding constraints in Interface Builder	63
Intrinsic content size	64
Misplaced views	66
Adding more constraints	67
Bronze Challenge: More Auto Layout Practice	68
4. Text Input and Delegation	69
Text Editing	70
Keyboard attributes	74
Responding to text field changes	74
Dismissing the keyboard	78
Implementing the Temperature Conversion	79
Number formatters	81
Delegation	82
Conforming to a protocol	82
Using a delegate	83
More on protocols	85
Bronze Challenge: Disallow Alphabetic Characters	85
5. View Controllers	89
The View of a View Controller	90
Setting the Initial View Controller	91
UITabBarController	95
Tab bar items	98
Loaded and Appearing Views	100
Accessing subviews	101
Interacting with View Controllers and Their Views	101
Silver Challenge: Dark Mode	101
For the More Curious: Retina Display	102
6. Programmatic Views	105
Creating a View Programmatically	107
Programmatic Constraints	108
Anchors	109
Activating constraints	110
Layout guides	111
Margins	112
Explicit constraints	113
Programmatic Controls	114
Bronze Challenge: Another Tab	116
Silver Challenge: User's Location	116
Gold Challenge: Dropping Pins	116
For the More Curious: NSAutoresizingMaskLayoutConstraint	117
7. Localization	119
Internationalization	121
Formatters	121

Base internationalization	125
Preparing for localization	126
Localization	133
NSString and strings tables	136
Bronze Challenge: Another Localization	140
For the More Curious: Bundle's Role in Internationalization	140
For the More Curious: Importing and Exporting as XLIFF	141
8. Controlling Animations	143
Basic Animations	144
Closures	145
Another Label	147
Animation Completion	149
Animating Constraints	149
Timing Functions	154
Bronze Challenge: Spring Animations	156
Silver Challenge: Layout Guides	156
9. Debugging	157
A Buggy Project	157
Debugging Basics	159
Interpreting console messages	159
Fixing the first bug	161
Caveman debugging	162
The Xcode Debugger: LLDB	164
Setting breakpoints	165
Stepping through code	166
The LLDB console	175
10. UITableView and UITableViewController	177
Beginning the Homeowner Application	177
UITableViewController	179
Subclassing UITableViewController	180
Creating the Item Class	181
Custom initializers	181
UITableView's Data Source	183
Giving the controller access to the store	184
Implementing data source methods	186
UITableViewCell	187
Creating and retrieving UITableViewCell	189
Reusing UITableViewCell	190
Content Insets	193
Bronze Challenge: Sections	194
Silver Challenge: Constant Rows	194
Gold Challenge: Customizing the Table	194
11. Editing UITableView	195
Editing Mode	195
Adding Rows	200
Deleting Rows	202
Moving Rows	203
Displaying User Alerts	205

Design Patterns	209
Bronze Challenge: Renaming the Delete Button	209
Silver Challenge: Preventing Reordering	209
Gold Challenge: Really Preventing Reordering	209
12. Subclassing UITableViewCell	211
Creating ItemCell	212
Exposing the Properties of ItemCell	214
Using ItemCell	215
Dynamic Cell Heights	216
Dynamic Type	217
Responding to user changes	220
Bronze Challenge: Cell Colors	220
13. Stack Views	221
Using UIStackView	223
Implicit constraints	224
Stack view distribution	227
Nested stack views	228
Stack view spacing	228
Segues	230
Hooking Up the Content	231
Passing Data Around	236
Bronze Challenge: More Stack Views	238
14. UINavigationController	239
UINavigationController	241
Navigating with UINavigationController	245
Appearing and Disappearing Views	246
Dismissing the Keyboard	247
Event handling basics	248
Dismissing by pressing the Return key	249
Dismissing by tapping elsewhere	250
UINavigationController	252
Adding buttons to the navigation bar	254
Bronze Challenge: Displaying a Number Pad	257
Silver Challenge: A Custom UITextField	257
Gold Challenge: Pushing More View Controllers	257
15. Camera	259
Displaying Images and UIImageView	260
Adding a camera button	262
Taking Pictures and UIImagePickerController	265
Setting the image picker's sourceType	265
Setting the image picker's delegate	267
Presenting the image picker modally	267
Permissions	268
Saving the image	271
Creating ImageStore	272
Giving View Controllers Access to the Image Store	273
Creating and Using Keys	274
Wrapping Up ImageStore	277

Bronze Challenge: Editing an Image	278
Silver Challenge: Removing an Image	278
Gold Challenge: Camera Overlay	278
For the More Curious: Navigating Implementation Files	279
// MARK:	281
16. Saving, Loading, and Application States	283
Archiving	284
Application Sandbox	287
Constructing a file URL	288
NSKeyedArchiver and NSKeyedUnarchiver	289
Loading files	292
Application States and Transitions	293
Writing to the Filesystem with Data	295
Error Handling	298
Bronze Challenge: PNG	300
For the More Curious: Application State Transitions	301
For the More Curious: Reading and Writing to the Filesystem	302
For the More Curious: The Application Bundle	304
17. Size Classes	307
Modifying Traits for a Specific Size Class	308
Bronze Challenge: Stacked Text Field and Labels	313
18. Touch Events and UIResponder	315
Touch Events	316
Creating the TouchTracker Application	317
Creating the Line Struct	318
Structs	319
Value types vs reference types	319
Creating DrawView	320
Drawing with DrawView	321
Turning Touches into Lines	322
Handling multiple touches	323
@IBInspectable	328
Silver Challenge: Colors	330
Gold Challenge: Circles	330
For the More Curious: The Responder Chain	331
For the More Curious: UIControl	332
19. UIGestureRecognizer and UIMenuController	333
UIGestureRecognizer Subclasses	334
Detecting Taps with UITapGestureRecognizer	334
Multiple Gesture Recognizers	336
UIMenuController	339
More Gesture Recognizers	341
UILongPressGestureRecognizer	341
UIPanGestureRecognizer and simultaneous recognizers	342
More on UIGestureRecognizer	346
Silver Challenge: Mysterious Lines	347
Gold Challenge: Speed and Size	347
Platinum Challenge: Colors	347

For the More Curious: UINavigationController and UIResponderStandardEditActions	348
20. Web Services	349
Starting the Potorama Application	350
Building the URL	352
Formatting URLs and requests	352
URLComponents	353
Sending the Request	357
URLSession	357
Modeling the Photo	360
JSON Data	361
JSONSerialization	362
Enumerations and associated values	363
Parsing JSON data	364
Downloading and Displaying the Image Data	371
The Main Thread	374
Bronze Challenge: Printing the Response Information	375
Silver Challenge: Fetch Recent Photos from Flickr	375
For the More Curious: HTTP	376
21. Collection Views	379
Displaying the Grid	380
Collection View Data Source	382
Customizing the Layout	385
Creating a Custom UICollectionViewCell	388
Downloading the Image Data	392
Extensions	395
Image caching	397
Navigating to a Photo	398
Silver Challenge: Updated Item Sizes	401
Gold Challenge: Creating a Custom Layout	401
22. Core Data	403
Object Graphs	403
Entities	403
Modeling entities	404
Transformable attributes	406
NSManagedObject and subclasses	406
NSPersistentContainer	408
Updating Items	408
Inserting into the context	409
Saving changes	411
Updating the Data Source	412
Fetch requests and predicates	412
Bronze Challenge: Photo View Count	416
For the More Curious: The Core Data Stack	416
NSManagedObjectModel	416
NSPersistentStoreCoordinator	416
NSManagedObjectContext	416
23. Core Data Relationships	417
Relationships	418

Adding Tags to the Interface	421
Background Tasks	432
Silver Challenge: Favorites	436
24. Accessibility	437
VoiceOver	437
Testing VoiceOver	439
Accessibility in Photorama	441
25. Afterword	445
What to Do Next	445
Shameless Plugs	445
Index	449

Introduction

As an aspiring iOS developer, you face three major tasks:

- *You must learn the Swift language.* Swift is the recommended development language for iOS. The first two chapters of this book are designed to give you a working knowledge of Swift.
- *You must master the big ideas.* These include things like delegation, archiving, and the proper use of view controllers. The big ideas take a few days to understand. When you reach the halfway point of this book, you will understand these big ideas.
- *You must master the frameworks.* The eventual goal is to know how to use every method of every class in every framework in iOS. This is a project for a lifetime: There are hundreds of classes and thousands of methods available in iOS, and Apple adds more classes and methods with every release of iOS. In this book, you will be introduced to each of the subsystems that make up the iOS SDK, but you will not study each one deeply. Instead, our goal is to get you to the point where you can search and understand Apple's reference documentation.

We have used this material many times at our iOS bootcamps at Big Nerd Ranch. It is well tested and has helped thousands of people become iOS developers. We sincerely hope that it proves useful to you.

Prerequisites

This book assumes that you are already motivated to learn to write iOS apps. We will not spend any time convincing you that the iPhone, iPad, and iPod touch are compelling pieces of technology.

We also assume that you have some experience programming and know something about object-oriented programming. If this is not true, you should probably start with *Swift Programming: The Big Nerd Ranch Guide*.

What Has Changed in the Sixth Edition?

All of the code in this book has been updated for Swift 3.0, which was a major update to the Swift language. Throughout the book, you will see how to use Swift's capabilities and features to write better iOS applications. We have come to love Swift at Big Nerd Ranch and believe you will, too.

Other additions include new chapters on debugging and accessibility and improved coverage of Core Data. We have also updated various chapters to use the technologies and APIs introduced in iOS 10.

This edition assumes that the reader is using Xcode 8.1 or later and running applications on an iOS 10 or later device.

Besides these obvious changes, we made thousands of tiny improvements that were inspired by questions from our readers and our students. Every chapter of this book is just a little better than the corresponding chapter from the fifth edition.

Our Teaching Philosophy

This book will teach you the essential concepts of iOS programming. At the same time, you will type in a lot of code and build a bunch of applications. By the end of the book, you will have knowledge *and* experience. However, all the knowledge should not (and, in this book, will not) come first. That is the traditional way of learning we have all come to know and hate. Instead, we take a learn-while-doing approach. Development concepts and actual coding go together.

Here is what we have learned over the years of teaching iOS programming:

- We have learned what ideas people must grasp to get started programming, and we focus on that subset.
- We have learned that people learn best when these concepts are introduced *as they are needed*.
- We have learned that programming knowledge and experience grow best when they grow together.
- We have learned that “going through the motions” is much more important than it sounds. Many times we will ask you to start typing in code before you understand it. We realize that you may feel like a trained monkey typing in a bunch of code that you do not fully grasp. But the best way to learn coding is to find and fix your typos. Far from being a drag, this basic debugging is where you really learn the ins and outs of the code. That is why we encourage you to type in the code yourself. You could just download it, but copying and pasting is not programming. We want better for you and your skills.

What does this mean for you, the reader? To learn this way takes some trust – and we appreciate yours. It also takes patience. As we lead you through these chapters, we will try to keep you comfortable and tell you what is happening. However, there will be times when you will have to take our word for it. (If you think this will bug you, keep reading – we have some ideas that might help.) Do not get discouraged if you run across a concept that you do not understand right away. Remember that we are intentionally *not* providing all the knowledge you will ever need all at once. If a concept seems unclear, we will likely discuss it in more detail later when it becomes necessary. And some things that are not clear at the beginning will suddenly make sense when you implement them the first (or the twelfth) time.

People learn differently. It is possible that you will love how we hand out concepts on an as-needed basis. It is also possible that you will find it frustrating. In case of the latter, here are some options:

- Take a deep breath and wait it out. We will get there, and so will you.
- Check the index. We will let it slide if you look ahead and read through a more advanced discussion that occurs later in the book.
- Check the online Apple documentation. This is an essential developer tool, and you will want plenty of practice using it. Consult it early and often.
- If Swift or object-oriented programming concepts are giving you a hard time (or if you think they will), you might consider backing up and reading our *Swift Programming: The Big Nerd Ranch Guide*.

How to Use This Book

This book is based on the class we teach at Big Nerd Ranch. As such, it was designed to be consumed in a certain manner.

Set yourself a reasonable goal, like, “I will do one chapter every day.” When you sit down to attack a chapter, find a quiet place where you will not be interrupted for at least an hour. Shut down your email, your Twitter client, and your chat program. This is not a time for multitasking; you will need to concentrate.

Do the actual programming. You can read through a chapter first, if you like. But the real learning comes when you sit down and code as you go. You will not really understand the idea until you have written a program that uses it and, perhaps more importantly, debugged that program.

A couple of the exercises require supporting files. For example, in the first chapter you will need an icon for your Quiz application, and we have one for you. You can download the resources and solutions to the exercises from www.bignerdranch.com/solutions/iOSProgramming6ed.zip.

There are two types of learning. When you learn about the Peloponnesian War, you are simply adding details to a scaffolding of ideas that you already understand. This is what we will call “Easy Learning.” Yes, learning about the Peloponnesian War can take a long time, but you are seldom flummoxed by it. Learning iOS programming, on the other hand, is “Hard Learning,” and you may find yourself quite baffled at times, especially in the first few days. In writing this book, we have tried to create an experience that will ease you over the bumps in the learning curve. Here are two things you can do to make the journey easier:

- Find someone who already knows how to write iOS applications and will answer your questions. In particular, getting your application onto a device the first time is usually very frustrating if you are doing it without the help of an experienced developer.
- Get enough sleep. Sleepy people do not remember what they have learned.

How This Book Is Organized

In this book, each chapter addresses one or more ideas of iOS development through discussion and hands-on practice. For more coding practice, most chapters include challenge exercises. We encourage you to take on at least some of these. They are excellent for firming up your grasp of the concepts introduced in the chapter and for making you a more confident iOS programmer. Finally, most chapters conclude with one or two For the More Curious sections that explain certain consequences of the concepts that were introduced earlier.

Chapter 1 introduces you to iOS programming as you build and deploy a tiny application called Quiz. You will get your feet wet with Xcode and the iOS simulator along with all the steps for creating projects and files. The chapter includes a discussion of Model-View-Controller and how it relates to iOS development.

Chapter 2 provides an overview of Swift, including basic syntax, types, optionals, initialization, and how Swift is able to interact with the existing iOS frameworks. You will also get experience working in a playground, Xcode’s prototyping tool.

In Chapter 3, you will focus on the iOS user interface as you learn about views and the view hierarchy and create an application called WorldTrotter.

Chapter 4 introduces delegation, an important iOS design pattern. You will also add a text field to `WorldTrotter`.

In Chapter 5, you will expand `WorldTrotter` and learn about using view controllers for managing user interfaces. You will get practice working with views and view controllers as well as navigating between screens using a tab bar.

In Chapter 6, you will learn how to manage views and view controllers in code. You will add a segmented control to `WorldTrotter` that will let you switch between various map types.

Chapter 7 introduces the concepts and techniques of internationalization and localization. You will learn about **Locale**, strings tables, and **Bundle** as you localize parts of `WorldTrotter`.

In Chapter 8, you will learn about and add different types of animations to the Quiz project that you created in Chapter 1.

Chapter 9 will walk you through some of the tools at your disposal for debugging – finding and fixing issues in your application.

Chapter 10 introduces the largest application in the book – `Homepwner`. (“`Homepwner`” is not a typo; you can find the definition of “pwn” at www.wiktionary.org.) This application keeps a record of your items in case of fire or other catastrophe. `Homepwner` will take eight chapters to complete.

In Chapter 10 – Chapter 12, you will work with tables. You will learn about table views, their view controllers, and their data sources. You will learn how to display data in a table, how to allow the user to edit the table, and how to improve the interface.

Chapter 13 introduces stack views, which will help you create complex interfaces easily. You will use a stack view to add a new screen to `Homepwner` that displays an item’s details.

Chapter 14 builds on the navigation experience gained in Chapter 5. You will use **UINavigationController** to give `Homepwner` a drill-down interface and a navigation bar.

Chapter 15 introduces the camera. You will take pictures and display and store images in `Homepwner`.

In Chapter 16, you will add persistence to `Homepwner`, using archiving to save and load the application data.

In Chapter 17, you will learn about size classes, and you will use these to update `Homepwner`’s interface to scale well across various screen sizes.

In Chapter 18 and Chapter 19, you will create a drawing application named `TouchTracker` to learn about touch events. You will see how to add multitouch capability and how to use **UIGestureRecognizer** to respond to particular gestures. You will also get experience with the first responder and responder chain concepts and more practice using structures and dictionaries.

Chapter 20 introduces web services as you create the `Photorama` application. This application fetches and parses JSON data from a server using **NSURLSession** and **JSONSerialization**.

In Chapter 21, you will learn about collection views as you build an interface for `Photorama` using **UICollectionView** and **UICollectionViewCell**.

In Chapter 22 and Chapter 23, you will add persistence to `Photorama` using Core Data. You will store and load images and associated data using an **NSManagedObjectContext**.

Chapter 24 will walk you through making your applications accessible to more people by adding VoiceOver information.

Style Choices

This book contains a lot of code. We have attempted to make that code and the designs behind it exemplary. We have done our best to follow the idioms of the community, but at times we have wandered from what you might see in Apple’s sample code or code you might find in other books. In particular, you should know up front that we nearly always start a project with the simplest template project: the single view application. When your app works, you will know it is because of your efforts – not because of behavior built into the template.

Typographical Conventions

To make this book easier to read, certain items appear in certain fonts. Classes, types, methods, and functions appear in a bold, fixed-width font. Classes and types start with capital letters, and methods and functions start with lowercase letters. For example, “In the **loadView()** method of the **RexViewController** class, create a constant of type **String**.”

Variables, constants, and filenames appear in a fixed-width font but are not bold. So you will see, “In `ViewController.swift`, add a variable named `fido` and initialize it to “Rufus”.”

Application names, menu choices, and button names appear in a sans serif font. For example, “Open Xcode and select New Project... from the File menu. Select Single View Application and then click Next.”

All code blocks are in a fixed-width font. Code that you need to type in is bold; code that you need to delete is struck through. For example, in the following code, you would delete the line `import Foundation` and type in the two lines beginning **`@IBOutlet`**. The other lines are already in the code and are included to let you know where to add the new lines.

```
import Foundation
import UIKit

class ViewController: UIViewController {

    @IBOutlet var questionLabel: UILabel!
    @IBOutlet var answerLabel: UILabel!

}
```

Necessary Hardware and Software

To build the applications in this book, you must have Xcode 8.1, which requires a Mac running macOS El Capitan version 10.11.4 or later. Xcode, Apple’s Integrated Development Environment, is available on the App Store. Xcode includes the iOS SDK, the iOS simulator, and other development tools.

You should join the Apple Developer Program, which costs \$99/year, because:

- Downloading the latest developer tools is free for members.
- You cannot put an app in the store until you are a member.

If you are going to take the time to work through this entire book, membership in the Apple Developer Program is worth the cost. Go to developer.apple.com/programs/ios/ to join.

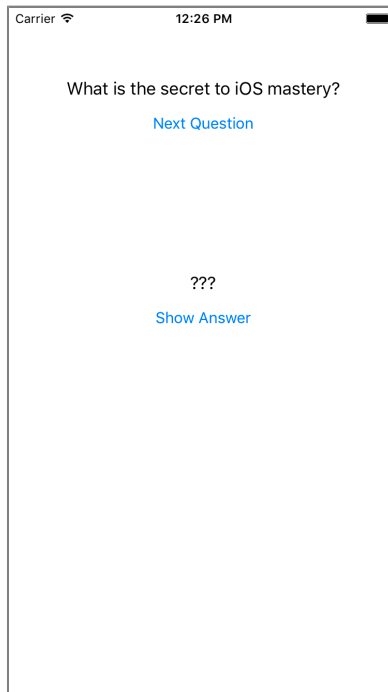
What about iOS devices? Most of the applications you will develop in the first half of the book are for iPhone, but you will be able to run them on an iPad. On the iPad screen, iPhone applications appear in an iPhone-sized window. Not a compelling use of iPad, but that is OK when you are starting with iOS. In the early chapters, you will be focused on learning the fundamentals of the iOS SDK, and these are the same across iOS devices. Later in the book, you will see how to make applications run natively on both iOS device families.

Excited yet? Good. Let's get started.

A Simple iOS Application

In this chapter, you are going to write an iOS application named Quiz. This application will show a question and then reveal the answer when the user taps a button. Tapping another button will show the user a new question (Figure 1.1).

Figure 1.1 Your first application: Quiz



When you are writing an iOS application, you must answer two basic questions:

- How do I get my objects created and configured properly? (Example: “I want a button here that says Next Question.”)
- How do I make my app respond to user interaction? (Example: “When the user taps the button, I want this piece of code to be executed.”)

Most of this book is dedicated to answering these questions.

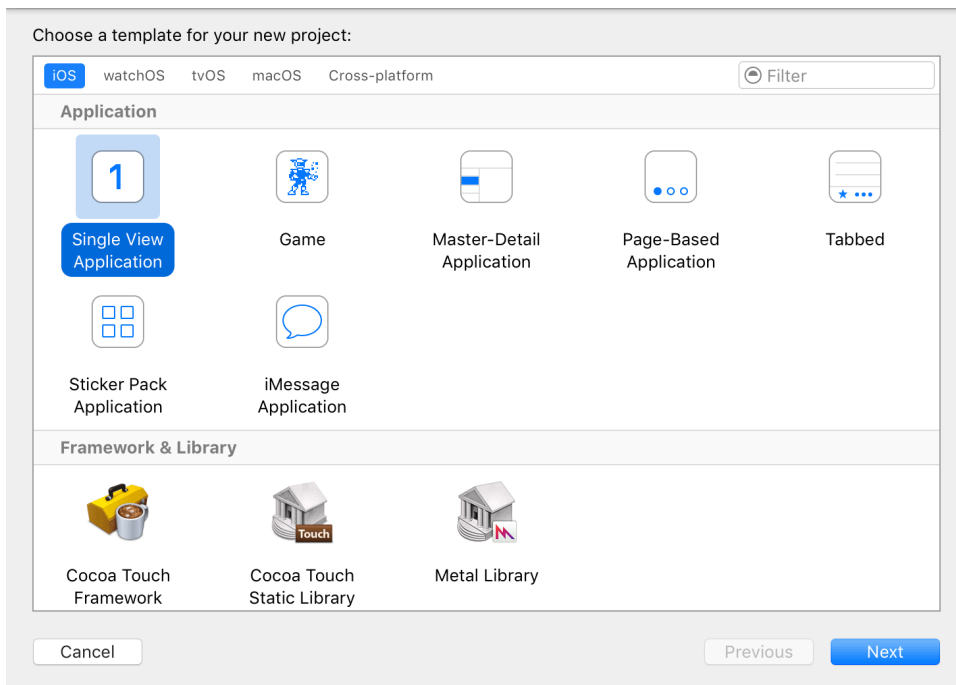
As you go through this first chapter, you will probably not understand everything that you are doing, and you may feel ridiculous just going through the motions. But going through the motions is enough for now. Mimicry is a powerful form of learning; it is how you learned to speak, and it is how you will start iOS programming. As you become more capable, you will experiment and challenge yourself to do creative things on the platform. For now, go ahead and do what we show you. The details will be explained in later chapters.

Creating an Xcode Project

Open Xcode and, from the File menu, select New → Project.... (If Xcode opens to a welcome screen, select Create a new Xcode project.)

A new workspace window will appear and a sheet will slide down from its toolbar. At the top, find the iOS section and then the Application area (Figure 1.2). You are offered several application templates to choose from. Select Single View Application.

Figure 1.2 Creating a project



This book was created for Xcode 8.1. The names of these templates may change with new Xcode releases. If you do not see a Single View Application template, use the simplest-sounding template. You can also visit the Big Nerd Ranch forum for this book at forums.bignerdranch.com for help working with newer versions of Xcode.

Click Next and, in the next sheet, enter Quiz for the Product Name (Figure 1.3). The organization name and identifier are required to continue. You can use Big Nerd Ranch or any organization name you would like. For the organization identifier, you can use `com.bignerdranch` or `com.yourcompanynamehere`.

From the Language pop-up menu, choose Swift, and from the Devices pop-up menu, choose Universal. Make sure that the Use Core Data checkbox is unchecked.

Figure 1.3 Configuring a new project

Choose options for your new project:

Product Name: Quiz

Team: None

Organization Name: Big Nerd Ranch

Organization Identifier: com.bignerdranch

Bundle Identifier: com.bignerdranch.Quiz

Language: Swift

Devices: Universal

☐ Use Core Data

☒ Include Unit Tests

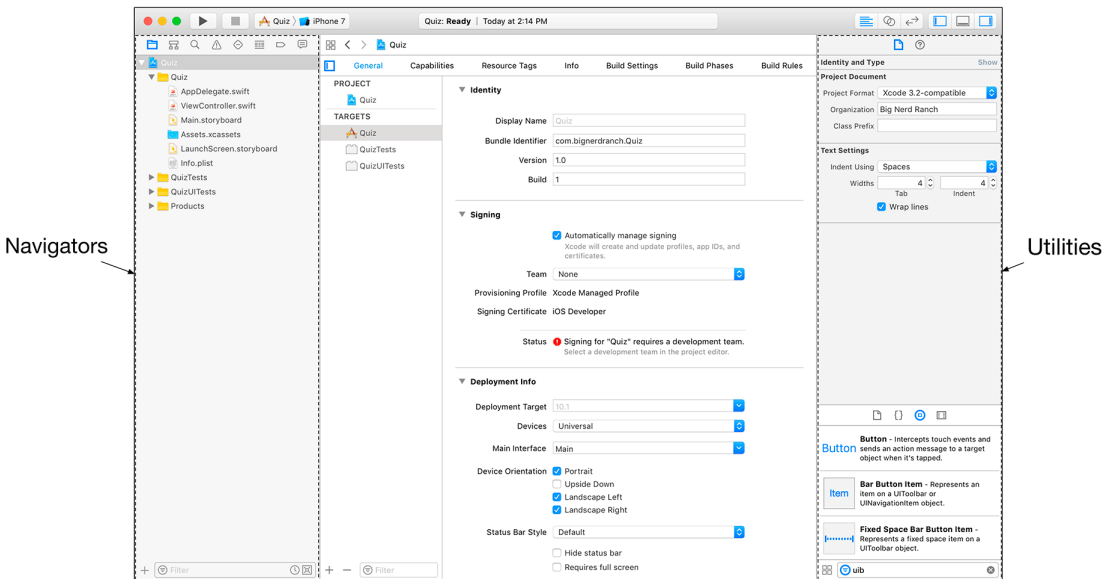
☒ Include UI Tests

Cancel Previous Next

Click Next and, in the final sheet, save the project in the directory where you plan to store the exercises in this book. Click Create to create the Quiz project.

Your new project opens in the Xcode workspace window (Figure 1.4).

Figure 1.4 Xcode workspace window

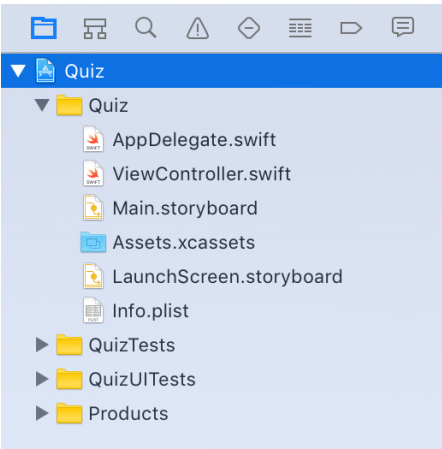


The lefthand side of the workspace window is the *navigator area*. This area displays different *navigators* – tools that show you different parts of your project. You can open a navigator by selecting one of the icons in the *navigator selector*, which is the bar just above the navigator area.

The navigator currently open is the *project navigator*. The project navigator shows you the files that make up a project (Figure 1.5). You can select one of these files to open it in the *editor area* to the right of the navigator area.

The files in the project navigator can be grouped into folders to help you organize your project. A few groups have been created by the template for you. You can rename them, if you want, or add new ones. The groups are purely for the organization of files and do not correlate to the filesystem in any way.

Figure 1.5 Quiz application’s files in the project navigator



Model-View-Controller

Before you begin your application, let's discuss a key concept in application architecture:

Model-View-Controller, or MVC. MVC is a design pattern used in iOS development. In MVC, every instance belongs to either the *model layer*, the *view layer*, or the *controller layer*. (*Layer* here simply refers to one or more objects that together fulfill a role.)

- The *model layer* holds data and knows nothing about the user interface, or UI. In Quiz, the model will consist of two ordered lists of strings: one for questions and another for answers.

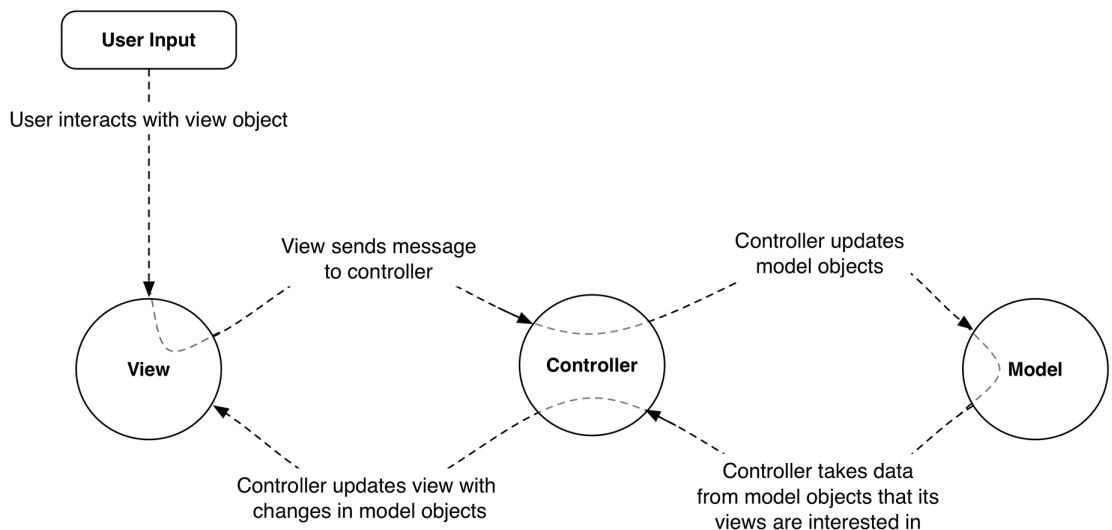
Usually, instances in the model layer represent real things in the world of the user. For example, when you write an app for an insurance company, your model will almost certainly contain a custom type called **InsurancePolicy**.

- The *view layer* contains objects that are visible to the user. Examples of *view objects*, or *views*, are buttons, text fields, and sliders. View objects make up an application's UI. In Quiz, the labels showing the question and answer and the buttons beneath them are view objects.
- The *controller layer* is where the application is managed. *Controller objects*, or *controllers*, are the managers of an application. Controllers configure the views that the user sees and make sure that the view and model objects stay synchronized.

In general, controllers typically handle “And then?” questions. For example, when the user selects an item from a list, the controller determines what the user sees next.

Figure 1.6 shows the flow of control in an application in response to user input, such as the user tapping a button.

Figure 1.6 MVC pattern



Notice that models and views do not talk to each other directly; controllers sit squarely in the middle of everything, receiving messages and dispatching instructions.

Designing Quiz

You are going to write the Quiz application using the MVC pattern. Here is a breakdown of the instances you will be creating and working with:

- The model layer will consist of two instances of **[String]**.
- The view layer will consist of two instances of **UILabel** and two instances of **UIButton**.
- The controller layer will consist of an instance of **ViewController**.

These instances and their relationships are laid out in the diagram for Quiz shown in Figure 1.7.

Figure 1.7 Object diagram for Quiz

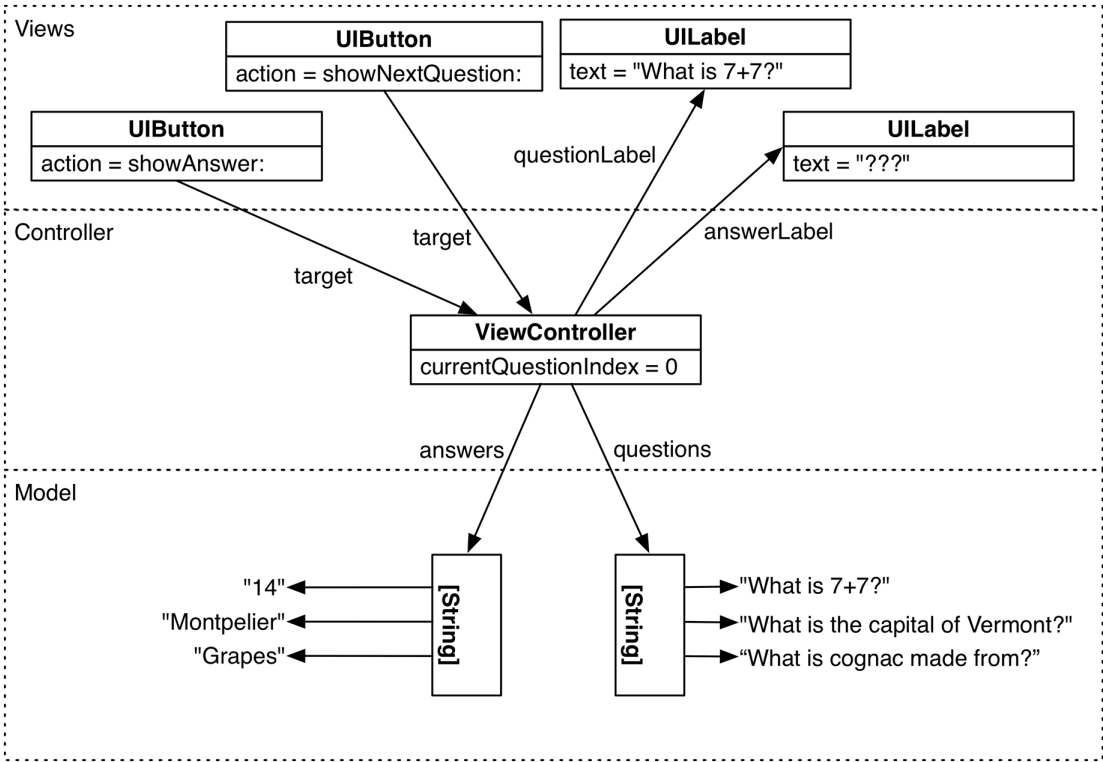


Figure 1.7 is the big picture of how the finished Quiz application will work. For example, when the Next Question button is tapped, it will trigger a *method* in **ViewController**. A method is a lot like a function – a list of instructions to be executed. This method will retrieve a new question from the array of questions and ask the top label to display that question.

It is OK if this diagram does not make sense yet – it will by the end of the chapter. Refer back to it as you build the app to see how it is taking shape.

You are going to build Quiz in steps, starting with the visual interface for the application.

Interface Builder

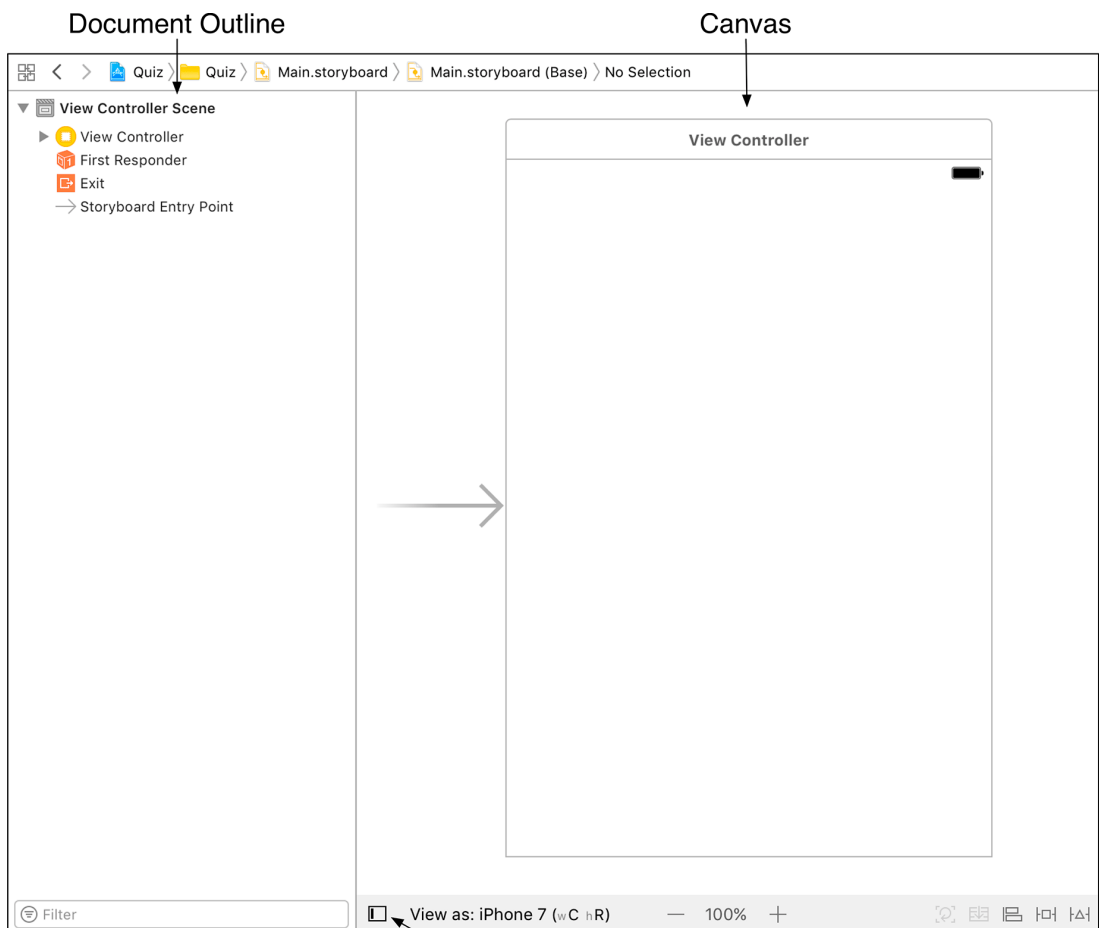
You are using the Single View Application template because it is the simplest template that Xcode offers. Still, this template has a significant amount of magic in that some critical components have already been set up for you. For now, you will just use these components, without attempting to gain a deep understanding of how they work. The rest of the book will be concerned with those details.

In the project navigator, click once on the `Main.storyboard` file. Xcode will open its graphic-style editor called Interface Builder.

Interface Builder divides the editor area into two sections: the *document outline*, on the lefthand side, and the *canvas*, on the right.

This is shown in Figure 1.8. If what you see in your editor area does not match the figure, you may have to click on the Show Document Outline button. (If you have additional areas showing, do not worry about them.) You may also have to click on the disclosure triangles in the document outline to reveal content.

Figure 1.8 Interface Builder showing `Main.storyboard`



Show / Hide Document Outline

The rectangle that you see in the Interface Builder canvas is called a *scene* and represents the only “screen” or view your application has at this time (remember that you used the single view application template to create this project).

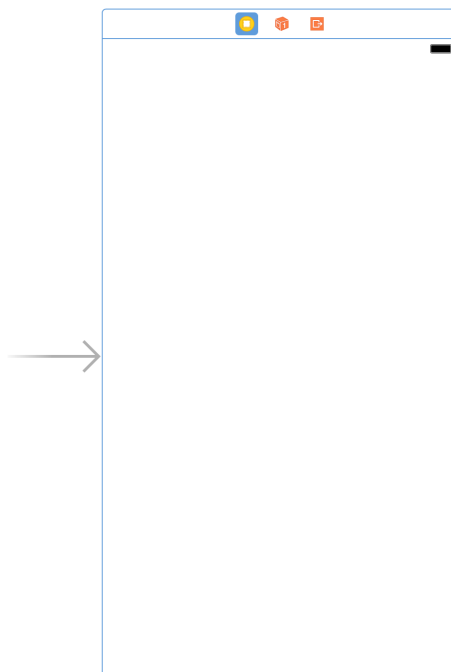
In the next section, you will learn how to create a UI for your application using Interface Builder. Interface Builder lets you drag objects from a library onto the canvas to create instances and also lets you establish connections between those objects and your code. These connections can result in code being called by a user interaction.

A crucial feature of Interface Builder is that it is not a graphical representation of code contained in other files. Interface Builder is an object editor that can create instances of objects and manipulate their properties. When you are done editing an interface, it does not generate code that corresponds to the work you have done. A `.storyboard` file is an archive of object instances to be loaded into memory when necessary.

Building the Interface

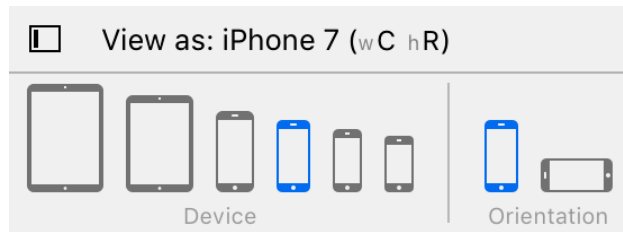
Let’s get started on your interface. You have selected `Main.storyboard` to reveal its single scene in the canvas (Figure 1.9).

Figure 1.9 The scene in `Main.storyboard`




To start, make sure your scene is sized for iPhone 7. At the bottom of the canvas, find the **View as** button. It will likely say something like **View as: iPhone 7 (wC hR)**. (The **wC hR** will not make sense right now; we will explain it in Chapter 17.) If it says **iPhone 7** already, then you are all set. If not, click on the **View as** button and select the fourth device from the left, which corresponds to iPhone 7 (Figure 1.10).

Figure 1.10 Viewing the scene for iPhone 7



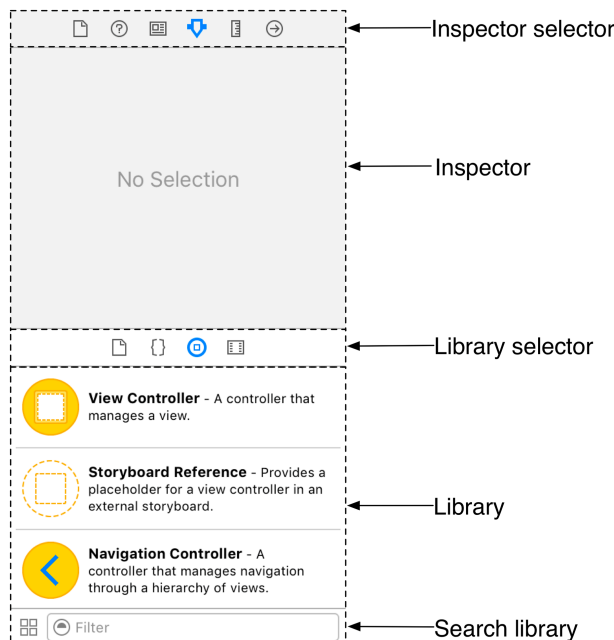
It is time to add your view objects to that blank slate.


Creating view objects

Make sure that the utility area within Xcode's window is visible. You may need to click on the rightmost button of the  control in the top-right corner of the window. The utility area is to the right of the editor area and has two sections: the *inspector* and the *library*. The top section is the inspector, which displays settings for a file or object that is selected in the editor area. The bottom section is the library, which lists items that you can add to a file or project.

At the top of each section in the utility area is a selector for different inspectors and libraries (Figure 1.11).

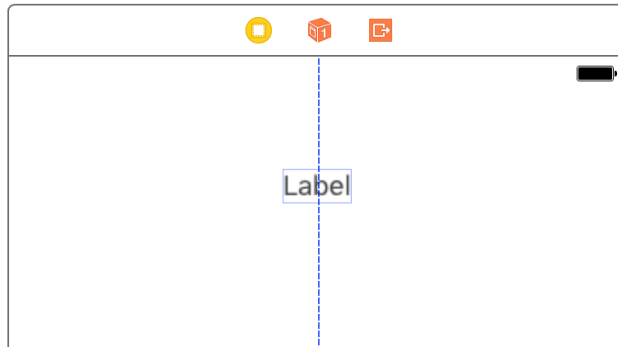
Figure 1.11 Xcode utility area



Your application interface requires four view objects: two buttons to accept user input and two text labels to display information. To add them, first make sure you can see the object library, as shown in Figure 1.11, by selecting the  tab from the library selector.

The object library contains the objects that you can add to a storyboard file to compose your interface. Find the **Label** object by either scrolling down through the list or by using the search bar at the bottom of the library. Select this object in the library and drag it onto the view object on the canvas. Drag the label around the canvas and notice the dashed blue lines that appear when the label is near the center of the canvas (Figure 1.12). These guidelines will help you lay out your interface.

Figure 1.12 Adding a label to the canvas

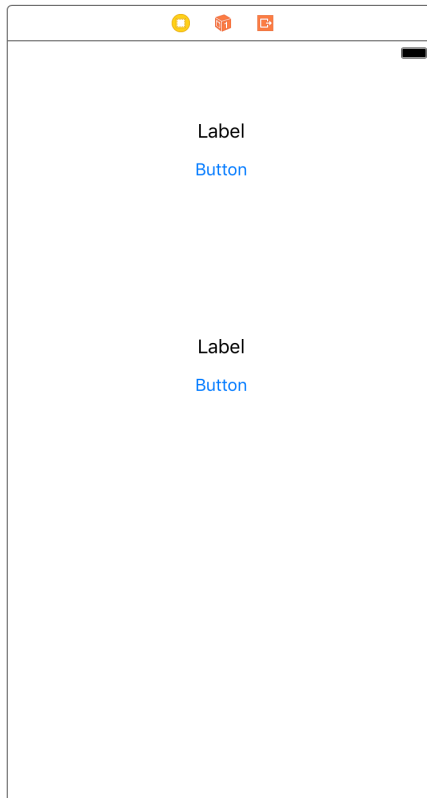


Using the guidelines, position the label in the horizontal center of the view and near the top, as shown in Figure 1.12. Eventually, this label will display questions to the user. Drag a second label onto the view and position it in the horizontal center, closer to the middle. This label will display answers.

Next, find **Button** in the object library and drag two buttons onto the view. Position one below each label.

You have now added four view objects to the **ViewController**'s UI. Notice that they also appear in the document outline. Your interface should look like Figure 1.13.

Figure 1.13 Building the Quiz interface

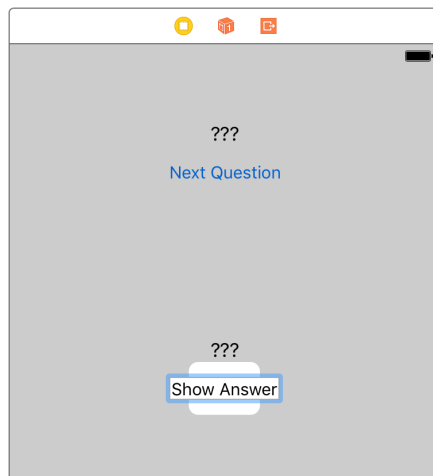


Configuring view objects

Now that you have created the view objects, you can configure their attributes. Some attributes of a view, like size, position, and text, can be changed directly on the canvas. For example, you can resize an object by selecting it in the canvas or the document outline and then dragging its corners and edges in the canvas.

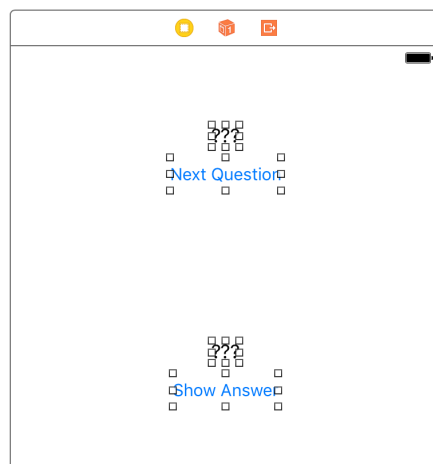
Begin by renaming the labels and buttons. Double-click on each label and replace the text with ??? . Then double-click the upper button and change its name to Next Question . Rename the lower button to Show Answer . The results are shown in Figure 1.14.

Figure 1.14 Renaming the labels and buttons



You may have noticed that because you have changed the text in the labels and buttons, and therefore their widths, they are no longer neatly centered in the scene. Click on each of them and drag to center them again, as shown in Figure 1.15.

Figure 1.15 Centering the labels and buttons

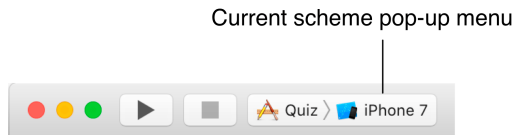


Running on the simulator

To test your UI, you are going to run Quiz on Xcode's iOS simulator.

To prepare Quiz to run on the simulator, find the current scheme pop-up menu on the Xcode toolbar (Figure 1.16).

Figure 1.16 iPhone 7 scheme selected



If it says something generic like iPhone 7, then the project is set to run on the simulator and you are good to go. If it says something like Christian's iPhone, then click and choose iPhone 7 from the pop-up menu. The iPhone 7 scheme will be your simulator default throughout this book.

Click the triangular play button in the toolbar. This will build (compile) and then run the application. You will be doing this often enough that you may want to learn and use the keyboard shortcut Command-R.

After the simulator launches you will see that the interface has all the views you added, neatly centered as you configured them in Interface Builder.

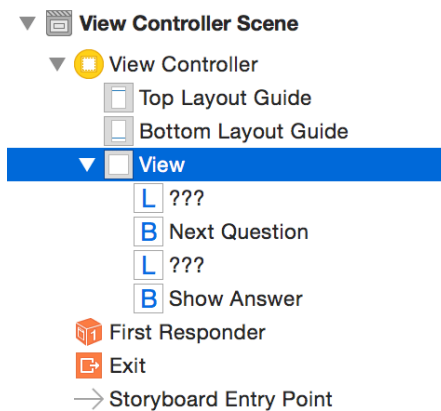
Now go back to the current scheme pop-up menu and select iPhone 7 Plus as your simulator of choice. Run the application again and you will notice that while the views you added are still present, they are not centered as they were on iPhone 7. This is because the labels and buttons currently have a fixed position on a screen, and they do not remain centered on the main view. To correct this problem, you will use a technology called *Auto Layout*.

A brief introduction to Auto Layout

As of now, your interface looks nice in the Interface Builder canvas. But iOS devices come in ever more screen sizes, and applications are expected to support all screen sizes and orientations – and perhaps more than one device type. You need to guarantee that the layout of view objects will be correct regardless of the screen size or orientation of the device running the application. The tool for this task is Auto Layout.

Auto Layout works by specifying position and size *constraints* for each view object in a scene. These constraints can be relative to neighboring views or to *container* views. A container view is just a view object that, as the name suggests, contains another view. For example, take a look at the document outline for Main.storyboard (Figure 1.17).

Figure 1.17 Document layout with a container view



You can see in the document outline that the labels and buttons you added are indented with respect to a View object. This view object is the container of the labels and buttons, and the objects can be positioned and sized relative to this view.

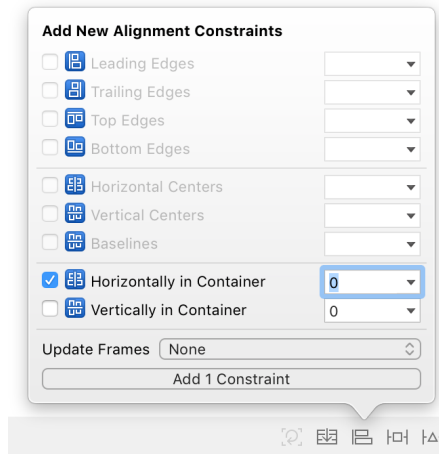
To begin specifying Auto Layout constraints, select the top label by clicking on it either on the canvas or in the document outline. At the bottom of the canvas, notice the Auto Layout menus, shown in Figure 1.18.

Figure 1.18 The Auto Layout menus



With the top label still selected, click on the  icon to reveal the Align menu shown in Figure 1.19.

Figure 1.19 Centering the top label in the container



Within the Align menu, check the Horizontally in Container checkbox to center the label in the container. Then click the Add 1 Constraint button. This constraint guarantees that on any size screen, in any orientation, the label will be centered horizontally.


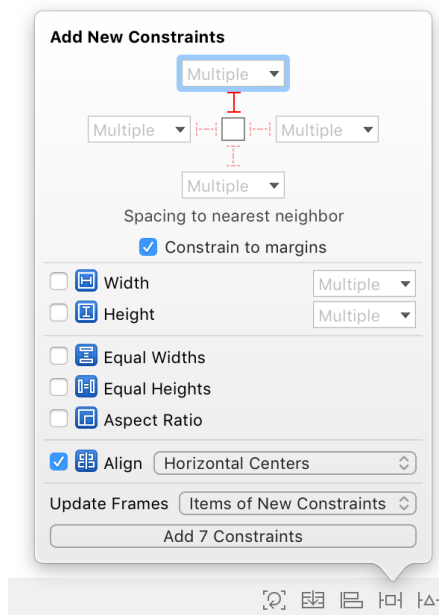
Now you need to add more constraints to center the lower label and the buttons with respect to the top label and to lock the spacing between them. Select the four views by Command-clicking on them one after another and then click on the  icon to open the *Add New Constraints* menu shown in Figure 1.20.

Figure 1.20 Adding constraints to center and fix the spacing between views



Click on the red vertical dashed segment near the top of the menu. When you click on the segment, it will become solid red (shown in Figure 1.20), indicating that the distance of each view is pinned to its nearest top neighbor. Also, check the Align box and then select **Horizontal Centers** from the pop-up menu. For Update Frames, make sure that you have **Items of New Constraints** selected. Finally, click on the **Add 7 Constraints** button at the bottom of the menu.


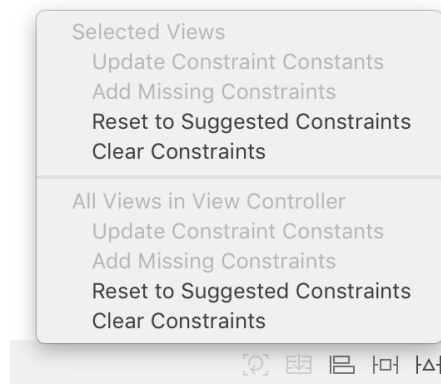
If you made any mistakes while adding constraints, you may see red or orange constraints and frames on the canvas instead of the correct blue lines. If that is the case, you will want to clear the existing constraints and go through the steps above again. To clear constraints, first select the background (container) view. Then click the  icon to open the **Resolve Auto Layout Issues** menu. Select **Clear Constraints** under the **All Views in View Controller** section (Figure 1.21). This will clear away any constraints that you have added and give you a fresh start on adding the constraints back in.

Figure 1.21 Clearing constraints



Auto Layout can be a difficult tool to master, and that is why you are starting to use it in the first chapter of this book. By starting early, you will have more chances to use it and get used to its complexity. Also, dealing with problems before things get too complicated will help you debug layout issues with confidence.

To confirm that your interface behaves correctly, build and run the application on the iPhone 7 Plus simulator. After confirming that the interface looks correct, build and run the application on the iPhone 7 simulator. The labels and buttons should be centered on both.

Making connections

A *connection* lets one object know where another object is in memory so that the two objects can communicate. There are two kinds of connections that you can make in Interface Builder: outlets and actions. An *outlet* is a reference to an object. An *action* is a method that gets triggered by a button or some other view that the user can interact with, like a slider or a picker.

Let's start by creating outlets that reference the instances of **UILabel**. Time to leave Interface Builder and write some code.

Declaring outlets

In the project navigator, find and select the file named `ViewController.swift`. The editor area will change from Interface Builder to Xcode's code editor.

In `ViewController.swift`, start by deleting any code that the template added between `class ViewController: UIViewController {` and the final brace, so that the file looks like this:

```
import UIKit

class ViewController: UIViewController {
}
```

(For simplicity, we will not show the line `import UIKit` again for this file.)

Next, add the following code that declares two properties. (Throughout this book, new code for you to add will be shown in bold. Code for you to delete will be struck through.) Do not worry about understanding the code or properties right now; just get it in.

```
class ViewController: UIViewController {
    @IBOutlet var questionLabel: UILabel!
    @IBOutlet var answerLabel: UILabel!
}
```

This code gives every instance of **ViewController** an outlet named `questionLabel` and an outlet named `answerLabel`. The view controller can use each outlet to reference a particular **UILabel** object (i.e., one of the labels in your view). The `@IBOutlet` keyword tells Xcode that you will connect these outlets to label objects using Interface Builder.

Setting outlets

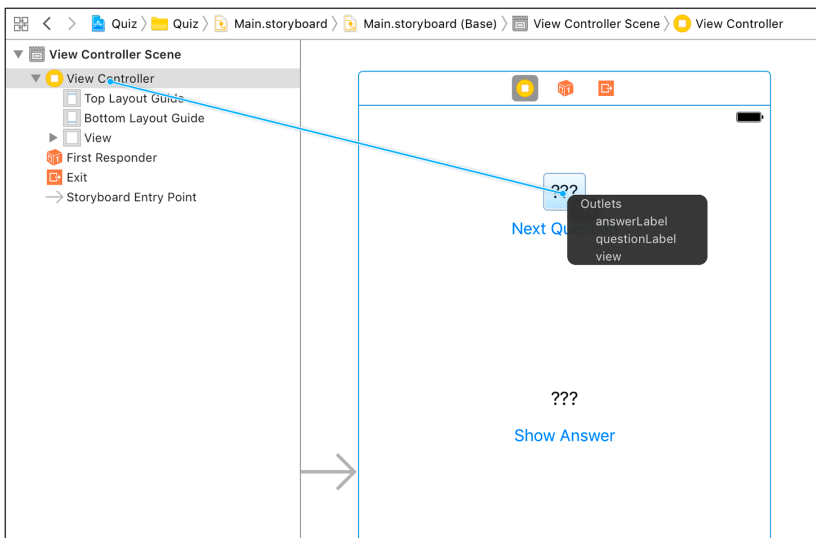
In the project navigator, select `Main.storyboard` to reopen Interface Builder.

You want the `questionLabel` outlet to point to the instance of **UILabel** at the top of the UI.

In the document outline, find the View Controller Scene section and the View Controller object within it. In your case, the View Controller stands in for an instance of **ViewController**, which is the object responsible for managing the interface defined in `Main.storyboard`.

Control-drag (or right-click and drag) from the View Controller in the document outline to the top label in the scene. When the label is highlighted, release the mouse and keyboard; a black panel will appear. Select `questionLabel` to set the outlet, as shown in Figure 1.22.

Figure 1.22 Setting `questionLabel`

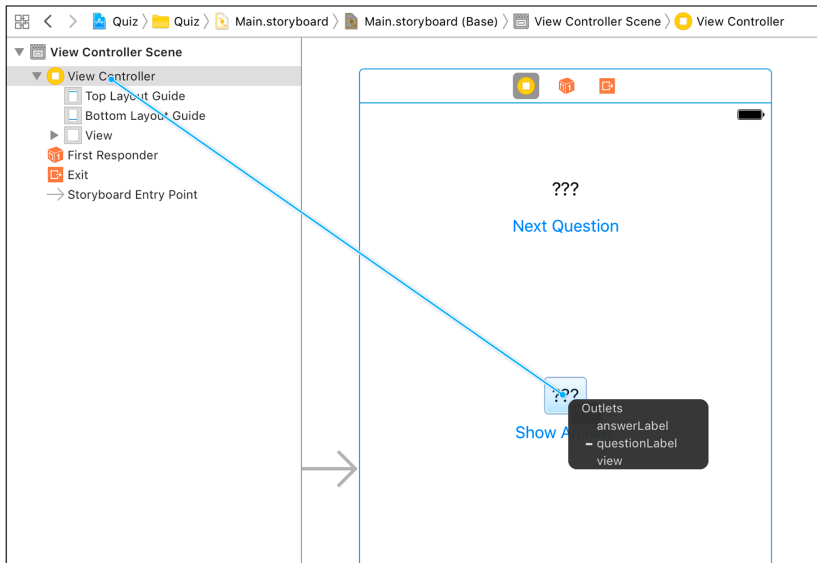


(If you do not see `questionLabel` in the connections panel, double-check your `ViewController.swift` file for typos.)

Now, when the storyboard file is loaded, the **ViewController**'s `questionLabel` outlet will automatically reference the instance of **UILabel** at the top of the screen, which will allow the **ViewController** to tell the label what question to display.

Set the `answerLabel` outlet the same way: Control-drag from the **ViewController** to the bottom **UILabel** and select `answerLabel` (Figure 1.23).

Figure 1.23 Setting answerLabel



Notice that you drag *from* the object with the outlet that you want to set *to* the object that you want that outlet to point to.

Your outlets are all set. The next connections you need to make involve the two buttons.

Defining action methods

When a **UIButton** is tapped, it calls a method on another object. That object is called the *target*. The method that is triggered is called the *action*. This action is the name of the method that contains the code to be executed in response to the button being tapped.

In your application, the target for both buttons will be the instance of **ViewController**. Each button will have its own action. Let's start by defining the two action methods: **showNextQuestion(_)** and **showAnswer(_)**.

Reopen `ViewController.swift` and add the two action methods after the outlets.

```
class ViewController: UIViewController {
    @IBOutlet var questionLabel: UILabel!
    @IBOutlet var answerLabel: UILabel!

    @IBAction func showNextQuestion(_ sender: UIButton) {

    }

    @IBAction func showAnswer(_ sender: UIButton) {

    }
}
```

You will flesh out these methods after you make the target and action connections. The `@IBAction` keyword tells Xcode that you will be making these connections in Interface Builder.

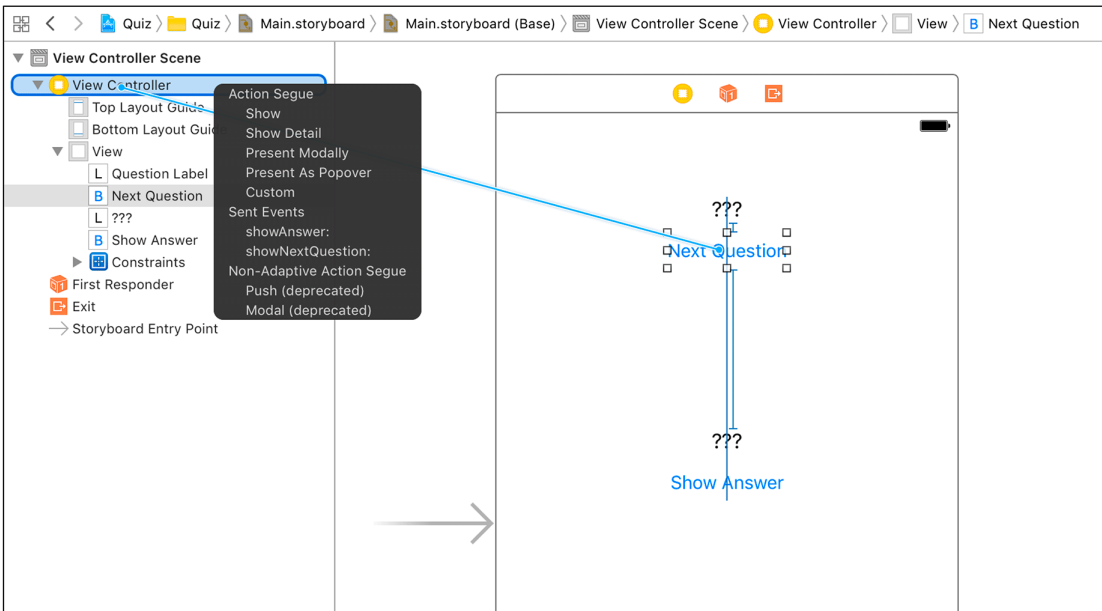
Setting targets and actions

Switch back to Main.storyboard. Let’s start with the Next Question button. You want its target to be **ViewController** and its action to be **showNextQuestion(_:)**.

To set an object’s target, you Control-drag *from* the object *to* its target. When you release the mouse, the target is set, and a pop-up menu appears that lets you select an action.

Select the Next Question button in the canvas and Control-drag to the View Controller in the document outline. When the View Controller is highlighted, release the mouse button and choose **showNextQuestion:** under Sent Events in the pop-up menu, as shown in Figure 1.24.

Figure 1.24 Setting Next Question target/action



Now for the Show Answer button. Select the button and Control-drag from the button to the View Controller. Choose **showAnswer:** from the pop-up menu.

Summary of connections

There are now five connections between the **ViewController** and the view objects. You have set the properties `answerLabel` and `questionLabel` to reference the label objects – two connections. The **ViewController** is the target for both buttons – two more. The project's template made one additional connection: The `view` property of **ViewController** is connected to the View object that represents the background of the application. That makes five.


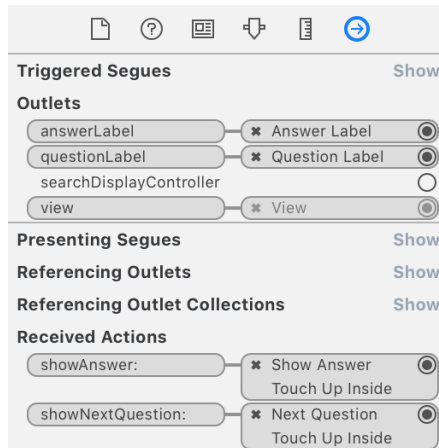
You can check these connections in the *connections inspector*. Select the View Controller in the document outline. Then, in the utilities area, click the  tab to reveal the connections inspector (Figure 1.25).

Figure 1.25 Checking connections in the connections inspector



Your storyboard file is complete. The view objects have been created and configured and all the necessary connections have been made to the controller object. Let's move on to creating and connecting your model objects.

Creating the Model Layer

View objects make up the UI, so developers typically create, configure, and connect view objects using Interface Builder. The parts of the model layer, on the other hand, are typically set up in code.

In the project navigator, select `ViewController.swift`. Add the following code that declares two arrays of strings and an integer.

```
class ViewController: UIViewController {
    @IBOutlet var questionLabel: UILabel!
    @IBOutlet var answerLabel: UILabel!

    let questions: [String] = [
        "What is 7+7?",
        "What is the capital of Vermont?",
        "What is cognac made from?"
    ]
    let answers: [String] = [
        "14",
        "Montpelier",
        "Grapes"
    ]
    var currentQuestionIndex: Int = 0
    ...
}
```

The arrays are ordered lists containing questions and answers. The integer will keep track of what question the user is on.

Notice that the arrays are declared using the `let` keyword, whereas the integer is declared using the `var` keyword. A *constant* is denoted with the `let` keyword; its value cannot change. The questions and answers arrays are constants. The questions and answers in this quiz will not change and, in fact, cannot be changed from their initial values.

A *variable*, on the other hand, is denoted by the `var` keyword; its value is allowed to change. You made the `currentQuestionIndex` property a variable because its value must be able to change as the user cycles through the questions and answers.

Implementing action methods

Now that you have questions and answers, you can finish implementing the action methods. In `ViewController.swift`, update `showNextQuestion(_:)` and `showAnswer(_:)`.

```
...
@IBAction func showNextQuestion(_ sender: UIButton) {
    currentQuestionIndex += 1
    if currentQuestionIndex == questions.count {
        currentQuestionIndex = 0
    }

    let question: String = questions[currentQuestionIndex]
    questionLabel.text = question
    answerLabel.text = "???"
}

@IBAction func showAnswer(_ sender: UIButton) {
    let answer: String = answers[currentQuestionIndex]
    answerLabel.text = answer
}
...
```

Loading the first question

Just after the application is launched, you will want to load the first question from the array and use it to replace the ??? placeholder in the `questionLabel` label. A good way to do this is by *overriding* the `viewDidLoad()` method of `ViewController`. (“Override” means that you are providing a custom implementation for a method.) Add the method to `ViewController.swift`.

```
class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        questionLabel.text = questions[currentQuestionIndex]
    }
}
```

All the code for your application is now complete!

Building the Finished Application

Build and run the application on the iPhone 7 simulator, as you did earlier.


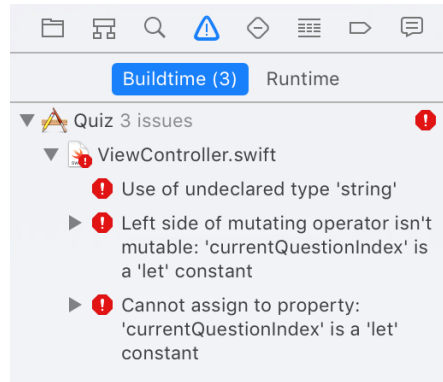
If building turns up any errors, you can view them in the *issue navigator* by selecting the  tab in the navigator area (Figure 1.26).

Figure 1.26 Issue navigator with example errors and warnings



Click on any error or warning in the issue navigator to be taken to the file and the line of code where the issue occurred. Find and fix any problems (i.e., code typos!) by comparing your code with the code in this chapter. Then try running the application again. Repeat this process until your application compiles.

After your application has compiled, it will launch in the iOS simulator. Play around with the Quiz application. You should be able to tap the Next Question button and see a new question in the top label; tapping Show Answer should show the right answer. If your application is not working as expected, double-check your connections in `Main.storyboard`.

You have built a working iOS app! Take a moment to bask in the glory.

OK, enough basking. Your app works, but it needs some spit and polish.

Application Icons

While running Quiz, select Hardware → Home from the simulator's menu. You will see that Quiz's icon is a boring, default tile. Let's give Quiz a better icon.

An *application icon* is a simple image that represents the application on the iOS Home screen. Different devices require different-sized icons, some of which are shown in Table 1.1.

Table 1.1 Application icon sizes by device

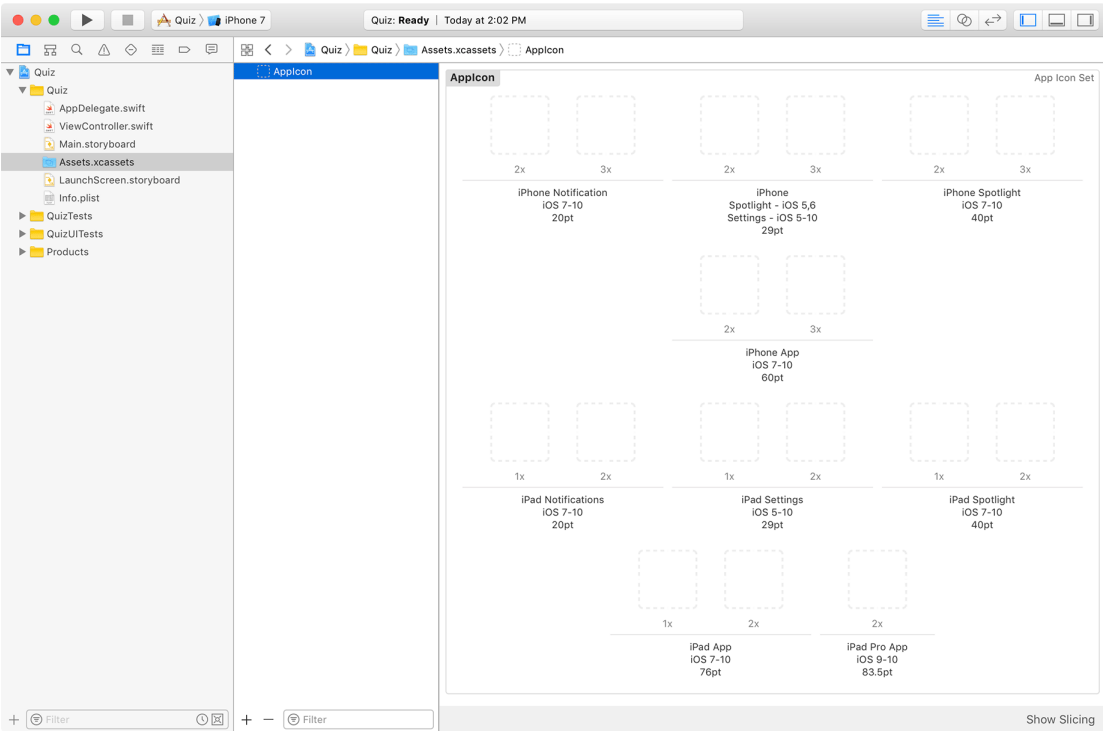
Device	Application icon sizes
5.5-inch iPhone	180x180 pixels (@3x)
4.7-inch and 4.0-inch iPhone	120x120 pixels (@2x)
7.9-inch and 9.7-inch iPad	152x152 pixels (@2x)
12.9-inch iPad	167x167 pixels (@2x)

We have prepared an icon image file (size 120x120) for the Quiz application. You can download this icon (along with resources for other chapters) from www.bignerdranch.com/solutions/iOSProgramming6ed.zip. Unzip `iOSProgramming6ed.zip` and find the `Quiz-120.png` file in the `0-Resources/Project App Icons` directory of the unzipped folder.

You are going to add this icon to your application bundle as a *resource*. In general, there are two kinds of files in an application: code and resources. Code (like `ViewController.swift`) is used to create the application itself. Resources are things like images and sounds that are used by the application at runtime.

In the project navigator, find `Assets.xcassets`. Select this file to open it and then select `AppIcon` from the resource list on the lefthand side (Figure 1.27).

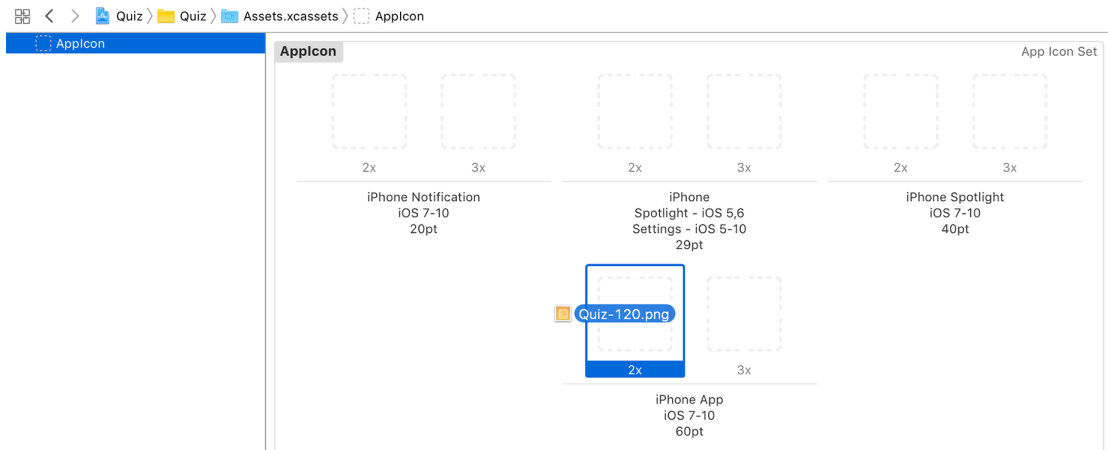
Figure 1.27 Showing the Asset Catalog



This panel is the *Asset Catalog*, where you can manage all of the images that your application will need.

Drag the Quiz-120.png file from Finder onto the 2x slot of the iPhone App section (Figure 1.28). This will copy the file into your project's directory on the filesystem and add a reference to that file in the Asset Catalog. (You can Control-click on a file in the Asset Catalog and select the option to Show in Finder to confirm this.)

Figure 1.28 Adding the app icon to the Asset Catalog



Build and run the application again. Switch to the simulator's Home screen either by clicking Hardware → Home, as you did before, or by using the keyboard shortcut Command-Shift-H. You should see the new icon.

(If you do not see the icon, delete the application and then build and run again to redeploy it. To do this, the easiest option is to reset the simulator by clicking Simulator → Reset Content and Settings.... This will remove all applications and reset the simulator to its default settings. You should see the app icon the next time you run the application.)

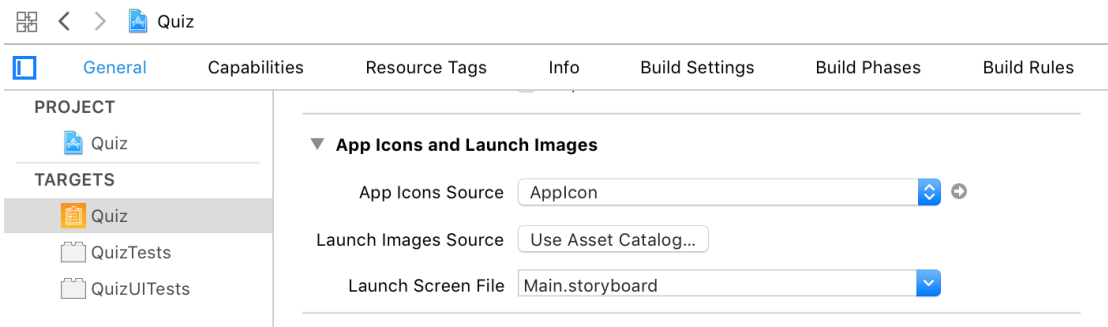
Launch Screen

Another item you should set for the project is the *launch image*, which appears while an application is loading. The launch image has a specific role in iOS: It conveys to the user that the application is indeed launching and depicts the UI that the user will interact with once the application loads. Therefore, a good launch image is a content-less screenshot of the application. For example, the Clock application’s launch image shows the four tabs along the bottom, all in the unselected state. Once the application loads, the correct tab is selected and the content becomes visible. (Keep in mind that the launch image is replaced after the application has launched; it does not become the background image of the application.)

An easy way to accomplish this is to allow Xcode to generate the possible launch screen images for you using a *launch screen file*.

Open the project settings by clicking on the top-level Quiz in the project navigator. Under App Icons and Launch Images, choose Main.storyboard from the Launch Screen File dropdown (Figure 1.29). Launch images will now be generated from Main.storyboard.

Figure 1.29 Setting the launch screen file



It is difficult to see the results of this change, because the launch image is typically shown for only a short time. However, it is a good practice to set the launch image even though its role is so brief.

Congratulations! You have written your first application and even added some details to make it polished. You will return to Quiz later in the book. The next chapter covers some basics of Swift to prepare you for more coding.

The Swift Language

Swift is a new language that Apple introduced in 2014. It replaces Objective-C as the recommended development language for iOS and Mac. In this chapter, you are going to focus on the basics of Swift. You will not learn everything, but you will learn enough to get started. Then, as you continue through the book, you will learn more Swift while you learn iOS development.

Swift maintains the expressiveness of Objective-C while introducing a syntax that is safer, succinct, and readable. It emphasizes type safety and adds advanced features such as optionals, generics, and sophisticated structures and enumerations. Most importantly, Swift allows the use of these new features while relying on the same tested, elegant iOS frameworks that developers have built upon for years.

If you know Objective-C, then the challenge is recasting what you know. It may seem awkward at first, but we have come to love Swift at Big Nerd Ranch and believe you will, too.

If you do not think you will be comfortable picking up Swift at the same time as iOS development, you may want to start with *Swift Programming: The Big Nerd Ranch Guide* or Apple's Swift tutorials, which you can find at developer.apple.com/swift. But if you have some programming experience and are willing to learn “on the job,” you can start your Swift education here and now.

Types in Swift

Swift types can be arranged into three basic groups: *structures*, *classes*, and *enumerations* (Figure 2.1). All three can have:

- *properties* – values associated with a type
- *initializers* – code that initializes an instance of a type
- *instance methods* – functions specific to a type that can be called on an instance of that type
- *class or static methods* – functions specific to a type that can be called on the type itself

Figure 2.1 Swift building blocks

Structures <pre>struct MyStruct { // properties // initializers // methods }</pre>	Enumerations <pre>enum MyEnum { // properties // initializers // methods }</pre>	Classes <pre>class MyClass: SuperClass { // properties // initializers // methods }</pre>
--	--	---

Swift’s structures (or “structs”) and enumerations (or “enums”) are significantly more powerful than in most languages. In addition to supporting properties, initializers, and methods, they can also conform to protocols and can be extended.

Swift’s implementation of typically “primitive” types such as numbers and Boolean values may surprise you: They are all structures. In fact, all of these Swift types are structures:

Numbers: **Int, Float, Double**

Boolean: **Bool**

Text: **String, Character**

Collections: **Array<Element>, Dictionary<Key:Hashable, Value>, Set<Element:Hashable>**

This means that standard types have properties, initializers, and methods of their own. They can also conform to protocols and be extended.

Finally, a key feature of Swift is *optionals*. An optional allows you to store either a value of a particular type or no value at all. You will learn more about optionals and their role in Swift later in this chapter.