

Up to date for iOS 13,
Xcode 11 & Swift 5.1



iOS Apprentice

EIGHTH EDITION

Beginning iOS Development with SwiftUI and UIKit



By the **raywenderlich Tutorial Team**

Joey deVilla, Eli Ganim, & Matthijs Hollemans

iOS Apprentice

Joey deVilla, Eli Ganim & Matthijs Hollemans

Copyright ©2019 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.



Table of Contents: Overview

Book License	16
Book Source Code & Forums	20
About the Cover	22
Section 1: Getting Started with SwiftUI	23
Chapter 1: Introduction	25
Chapter 2: Getting Started with SwiftUI.....	38
Chapter 3: Building User Interfaces.....	93
Chapter 4: Swift Basics.....	140
Chapter 5: A Fully Working Game	173
Chapter 6: Refactoring	191
Chapter 7: The New Look	218
Chapter 8: The Final App	260
Section 2: Checklists.....	281
Chapter 9: List Views.....	283
Chapter 10: A “Checkable” List	324
Chapter 11: The App Structure	356
Chapter 12: Adding Items to the List	392
Chapter 13: Editing Checklist Items	428
Chapter 14: Saving and Loading.....	458
Section 3: Getting Started with UIKit	486
Chapter 15: UIKit and The One-Button App	488

Chapter 16: Slider & Labels	507
Chapter 17: Outlets	523
Chapter 18: Polish	547
Chapter 19: The New Look	560
Chapter 20: Table Views	590
Chapter 21: The Data Model.....	613
Chapter 22: Navigation Controllers.....	626
Chapter 23: Edit High Score Screen.....	645
Chapter 24: Delegates & Protocols.....	663
Chapter 25: The Final App	679
Section 4: My Locations.....	699
Chapter 26: Swift Review	701
Chapter 27: Get Location Data	728
Chapter 28: Use Location Data	753
Chapter 29: Objects vs. Classes.....	784
Chapter 30: The Tag Location Screen	799
Chapter 31: Adding Polish.....	831
Chapter 32: Saving Locations.....	854
Chapter 33: The Locations Tab	889
Chapter 34: Maps	919
Chapter 35: Image Picker	942
Chapter 36: Polishing the App.....	980
Section 5: Store Search.....	1020

Chapter 37: Search Bar.....	1022
Chapter 38: Custom Table Cells.....	1052
Chapter 39: Networking	1083
Chapter 40: Asynchronous Networking.....	1118
Chapter 41: URLSession.....	1136
Chapter 42: The Detail Pop-Up.....	1168
Chapter 43: Polish the Pop-up.....	1193
Chapter 44: Landscape	1217
Chapter 45: Refactoring.....	1258
Chapter 46: Internationalization.....	1285
Chapter 47: The iPad	1312
Chapter 48: Distributing the App.....	1355
Conclusion.....	1383

Table of Contents: Extended

Book License	16
About the Authors	17
About the Editor	17
About the Artist.....	18
Book Source Code & Forums	20
About the Cover	22
Section 1: Getting Started with SwiftUI.....	23
Chapter 1: Introduction.....	25
About this book	26
Is this book right for you?	28
iOS 13 and later only.....	29
What you need.....	29
Xcode	31
What's ahead: An overview.....	32
The language of the computer	35
Chapter 2: Getting Started with SwiftUI	38
SwiftUI and UIKit	39
The Bullseye game	40
Getting started	43
Object-oriented programming.....	67
Adding interactivity.....	71
State and SwiftUI	79
Dealing with error messages	87
The anatomy of your app	90
Chapter 3: Building User Interfaces	93
Portrait vs. landscape.....	94
Adding the other views	98

Solving the mystery of the stuck slider	126
Data types	130
Making the slider less annoyingly precise	134
Key points	139
Chapter 4: Swift Basics	140
Generating and displaying the target value	141
Calculating and displaying the points scored.....	144
Writing your own methods.....	148
Improving the code.....	158
Key points	172
Chapter 5: A Fully Working Game.....	173
Improving the pointsForCurrentRound() algorithm	174
What's the score?.....	177
One more round.....	181
Key points	190
Chapter 6: Refactoring	191
Improvements.....	192
More refactoring	202
Starting over.....	206
Making the code less self-ish.....	209
A couple more enhancements.....	214
Key points	216
Chapter 7: The New Look	218
Landscape orientation revisited.....	219
Spicing up the graphics	221
The "About" screen	246
Chapter 8: The Final App	260
Adding a title to the navigation bar.....	261
Including animation	264
Adding an icon.....	264

Display name	267
Running the app on a device.....	269
The end... or the beginning?	279
Section 2: Checklists.....	281
Chapter 9: List Views.....	283
NavigationView and List views.....	285
Arrays	296
Loops	303
Deleting items from the list	311
Moving list items	319
Key points	323
Chapter 10: A “Checkable” List.....	324
Creating checklist item objects.....	325
A quick check before moving on.....	338
Toggling checklist items	341
Key points	355
Chapter 11: The App Structure	356
Design patterns: MVC and MVVM.....	357
Renaming the view	362
Adding a file for the model	364
Adding a file for the ViewModel.....	366
Structs and classes	372
Connecting the view to the ViewModel.....	377
What happens when you launch an app?.....	383
Key points	391
Chapter 12: Adding Items to the List.....	392
Setting up the user interface	393
Adding a new item to the list	409
Dealing with a SwiftUI bug	419

Improving the user interface	423
Key points	426
Chapter 13: Editing Checklist Items.....	428
Changing how the user changes checklist items	429
Giving checklist rows their own view	431
@Binding properties.....	447
Introducing extensions.....	450
Updating EditChecklistItemView	455
Key points	456
Chapter 14: Saving and Loading.....	458
Data persistence	459
The Documents directory	460
Saving checklist items	466
Loading the file	477
Next steps	485
Section 3: Getting Started with UIKit.....	486
Chapter 15: UIKit and The One-Button App.....	488
The Bullseye game.....	489
The one-button app	490
The anatomy of an app	504
Chapter 16: Slider & Labels	507
Portrait vs. landscape	508
Understanding objects, data and methods	510
Adding the other controls	514
Chapter 17: Outlets	523
Improving the slider	524
Generating the random number.....	529
Adding rounds to the game.....	531
Displaying the target value.....	537

Calculating the points scored	541
Showing the total score	542
Displaying the score.....	543
One more round...	544
Chapter 18: Polish.....	547
Tweaks	548
The alert.....	553
Start over.....	556
Chapter 19: The New Look	560
Landscape orientation revisited.....	561
Spicing up the graphics	562
The About screen.....	575
Chapter 20: Table Views.....	590
Table views and navigation controllers.....	591
Adding a table view.....	592
The table view delegates	598
Chapter 21: The Data Model	613
Model-View-Controller	614
The data model	616
Chapter 22: Navigation Controllers.....	626
Navigation controller.....	627
Deleting rows.....	631
Saving and loading high scores	635
Adding new high scores	636
The Edit High Score screen.....	638
Chapter 23: Edit High Score Screen.....	645
Static table cells	646
Working with the text field.....	650
Polishing it up.....	653

Chapter 24: Delegates & Protocols	663
Updating HighScoreItem.....	664
Chapter 25: The Final App.....	679
Supporting different screen sizes	680
Crossfade.....	696
UIKit knowledge unlocked!	697
Section 4: My Locations.....	699
Chapter 26: Swift Review	701
Variables, constants and types	702
Methods and functions.....	710
Making decisions	715
Loops	720
Objects	723
Protocols	726
Chapter 27: Get Location Data.....	728
Get GPS coordinates	730
Core Location	739
Displaying coordinates.....	749
Chapter 28: Use Location Data.....	753
Handling GPS errors.....	754
Improving GPS results.....	760
Reverse geocoding	765
Testing on device	774
Chapter 29: Objects vs. Classes	784
Classes	785
Inheritance	787
Overriding methods.....	791
Casts.....	795
Chapter 30: The Tag Location Screen	799

The screen.....	800
The new view controller.....	801
Making the cells	804
Displaying location info	814
The category picker	822
Chapter 31: Adding Polish.....	831
Improving the user experience.....	832
The HUD	837
Handling the navigation.....	849
Chapter 32: Saving Locations	854
Core Data overview	855
Adding Core Data	855
The data store	864
Passing the context.....	866
Browsing the data	872
Saving the locations	878
Handling Core Data errors	881
Chapter 33: The Locations Tab	889
The Locations tab.....	890
Creating a custom table view cell subclass.....	898
Editing locations.....	900
Using NSFetchedResultsController.....	907
Deleting locations	915
Table view sections	917
Chapter 34: Maps	919
Adding a map view	920
Making your own pins	932
Chapter 35: Image Picker	942
Adding an image picker.....	943
Showing the image	953

UI improvements	959
Saving the image.....	964
Editing the image.....	971
Thumbnails	973
Chapter 36: Polishing the App.....	980
Converting placemarks to strings	981
Back to black	985
The map screen.....	992
Fixing the table views	994
Polishing the main screen.....	1004
Making some noise.....	1013
The icon and launch images.....	1015
Where to go from here?.....	1019
Section 5: Store Search.....	1020
Chapter 37: Search Bar	1022
Creating the project	1024
Creating the UI.....	1029
Doing fake searches.....	1036
UI Improvements	1042
Creating the data model	1045
No results found	1046
Chapter 38: Custom Table Cells	1052
Custom table cells and nibs	1053
Changing the look of the app.....	1067
Tagging commits	1072
The debugger	1073
Chapter 39: Networking	1083
Query the iTunes web service	1084
Sending an HTTP request.....	1088

Parsing JSON	1093
Working with the JSON results.....	1103
Sorting the search results.....	1114
Chapter 40: Asynchronous Networking	1118
Extreme synchronous networking.....	1119
The activity indicator	1121
Making it asynchronous.....	1128
Chapter 41: URLSession	1136
Branching it	1137
Putting URLSession into action.....	1139
Cancelled operations.....	1149
Searching different categories.....	1151
Downloading the artwork	1158
Merge the branch	1165
Chapter 42: The Detail Pop-Up	1168
The new view controller	1169
Adding the rest of the controls.....	1177
Showing data in the pop-up	1185
Chapter 43: Polish the Pop-up	1193
Gradients in the background.....	1206
Animation!	1210
Chapter 44: Landscape	1217
The landscape view controller	1218
Fixing issues.....	1229
Adding a scroll view.....	1235
Adding result buttons.....	1241
Paging.....	1250
Download the artwork.....	1253
Chapter 45: Refactoring	1258

Refactoring the search.....	1259
Improving the categories	1266
Enums with associated values	1269
Spin me right round	1276
Nothing found.....	1279
The Detail pop-up.....	1281
Chapter 46: Internationalization	1285
Adding a new language.....	1286
Localizing on-screen text	1298
InfoPlist.strings	1307
Regional settings.....	1309
Chapter 47: The iPad.....	1312
Universal apps	1313
The split view controller	1314
Improving the detail pane.....	1322
Size classes in the storyboard.....	1329
Your own popover	1337
Sending e-mail from the app	1340
Landscape on iPhone Plus	1345
Dark mode support	1349
Chapter 48: Distributing the App.....	1355
Join the Apple Developer program	1356
Beta testing	1357
Submit for review	1377
The end	1380
Conclusion	1383

Book License

By purchasing *iOS Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *iOS Apprentice* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *iOS Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *iOS Apprentice*, available at www.raywenderlich.com”.
- The source code included in *iOS Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *iOS Apprentice* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.



About the Authors



Joey deVilla is an author of this book. Joey is a developer turned accordionist turned tech evangelist turned developer cat-herder. A Canadian turned Tampanian, he works at Fintech, blogs at globalnerdy.com and joeydevilla.com, and does what he can to avoid becoming a “Florida Man” news story.



Eli Ganim is an author of this book. He is an iOS engineer who's passionate about teaching, writing and sharing knowledge with others. He lives in Israel with his wife and kids.



Matthijs Hollemans is an author of this book. He is a mystic who lives at the top of a mountain where he spends all of his days and nights coding up awesome apps. Actually he lives below sea level in the Netherlands and is pretty down-to-earth but he does spend too much time in Xcode. Check out his website at www.matthijshollemans.com.

About the Editor



Adam Rush is the final pass editor for this book. He is a passionate iOS developer with over 7 years of commercial experience, contracting all over the UK & Europe. He's a tech addict and #Swift enthusiast. When he's not writing code, he enjoys watching sports and spending time with his family. You can reach him by @adam9rush

About the Artist



Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Dedications

"To my loved ones: Moriah, Lia and Ari."

— Eli Ganem

"To my family: Megan, Dexter and Aubrey."

— Adam Rush



Book Source Code & Forums

If you bought the digital edition

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded here:

- <https://store.raywenderlich.com/products/ios-apprentice>.

If you bought the print version

You can get the source code for the print edition of the book here: <https://store.raywenderlich.com/products/ios-apprentice-source-code>.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

Forums

We've also set up an official forum for the book here:

- <https://forums.raywenderlich.com/c/books/ios-apprentice>.

This is a great place to ask questions about the book or to submit any errors you may find.

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our *iOS Apprentice* store page here:

- <https://store.raywenderlich.com/products/ios-apprentice>.

About the Cover

Striped dolphins live to about 55-60 years of age, can travel in pods numbering in the thousands and can dive to depths of 700 m to feed on fish, cephalopods and crustaceans. Baby dolphins don't sleep for a full a month after they're born. That puts two or three sleepless nights spent debugging code into perspective, doesn't it?



Section 1: Getting Started with SwiftUI

This section introduces you to the first of the five apps you'll build throughout this book: *Bullseye*. It's a simple game that challenges the user to move a slider to a specific position without any hints or markers. While it won't make you an App Store millionaire, the exercise of writing it will introduce you to the basics of writing iOS apps.

This section will introduce you to several things you'll use when coding apps. You'll be introduced to Xcode, the integrated development environment for writing programs for Apple devices. You'll also be introduced to Swift, Apple's programming language, which has grown to become one of the most popular among programmers because it's simple, powerful, and fun.

Finally, you'll get your first taste of something that even the most experienced iOS developers haven't had much time to try: SwiftUI. It's the new way to build user interfaces for Apple platforms, and like the programming language from which it takes its name, it's simple and powerful. The first two apps that you'll make in this book will give you a great head start in building interfaces with SwiftUI.

This section aims to be beginner-friendly, and you may be tempted to skip it. Please don't, especially if you're new to iOS development. You'll need the fundamentals introduced in this section for later parts of the book, and you'll miss out on the basics of the all-new SwiftUI.

This section contains the following chapters:

1. Introduction: Welcome to **The iOS Apprentice!** In this book, you're about to deep dive into the latest and greatest Swift and iOS best practices. Throughout this five-section book, you will build four iOS projects using both UIKit and SwiftUI. Good luck!

2. The One-Button App: Take the first step of building a SwiftUI game by creating your iOS project, add some interactivity with a UIButton and learn all about the anatomy of an app.



- 3. Slider & Labels:** Bullseye is all about the slider, get sliding in this chapter by using the Slider control and learn all about different Swift data types.
- 4. A Basic Working Game:** In this chapter, you'll be well on your way to a working game. Learn all about generating random numbers and how best to improve your code afterward.
- 5. Rounds & Score:** A fully working Game: It's all about winning; in this chapter learn how best to score each game and introduce one more round functionality.
- 6. Refactoring:** At this point, you have created a fully functional game, wow! It's time to take a step back and look at best practices in the industry. It's time to clean up some code and make it more readable for the future!
- 7. The New Look:** Apps are known for their clean and simple UI. We will spice up the artwork in this chapter and make it look like a *real* game. We will also make improvements to landscape orientation.
- 8. The Final App:** To finish our game we will add some animations, an icon, and display name ready for the App Store!

Chapter 1: Introduction

Joey deVilla

Hi, welcome to *The iOS Apprentice: Beginning iOS Development with Swift, Eighth Edition*, the swiftest way (pardon the pun) to iOS development mastery!

In this book, you'll learn how to make your own iPhone and iPad apps using Apple's Swift programming language, Xcode 11, and the SwiftUI and UIKit user interface frameworks. You'll do this by building four interesting iOS apps.



The apps you'll make in The iOS Apprentice

Everybody likes games, right? So you'll start by building a simple but fun iPhone game named *Bullseye* that will teach you the basics of iPhone programming. The other apps will build on what you learn there.



Taken together, the four apps that you'll build cover everything you need to know to make your own apps. By the end of the book, you'll be experienced enough to turn your ideas into real apps that you can put on the App Store!

If you've never programmed before or you're new to iOS, don't worry. You should be able to follow along with the step-by-step instructions and understand how to make these apps. Each chapter has a ton of illustrations to keep you from getting lost. Not everything might make sense right away, but hang in there and all will become clear in time.

Writing your own iOS apps is a lot of fun, but it's also hard work. If you have the imagination and perseverance, there's no limit to what you can make your apps do. It's my sincere belief that this book can turn you from a complete newbie into an accomplished iOS developer, but you do have to put in the time and effort. By writing this book, I've done my part. The rest is up to you...

About this book

The *iOS Apprentice* will help you become an excellent iOS developer, but only if you let it. Here are some tips that will help you get the most out of this book.

Learn through repetition

You're going to make several apps in this book. Even though the apps are quite simple, you may find the instructions hard to follow at first — especially if you've never done any computer programming before. You'll be facing a lot of new concepts.

It's OK if you don't understand everything right away, as long as you get the general idea. As you proceed through the book, you'll go over many of those concepts again and again until they solidify in your mind.

Follow the instructions yourself

It's important that you not just read the instructions, but also actually **follow them**. Open Xcode, type in the source code fragments and run the app in the simulator as instructed. This helps you to see how the app develops, step by step.

Even better, play around with the code and with the Xcode settings. Feel free to modify any part of the app and see what the results are. Make a small change to the code and see how it affects the entire app. Experiment and learn! Don't worry about breaking stuff — that's half the fun. You can always find your way back to the



beginning. But better still, you might even learn something from simply breaking the code and learning how to fix it.

If you try everything but are still stuck, drop by the forums for this book at forums.raywenderlich.com. I'm around most of the time and will be happy to answer any questions related to the book and issues you might have run into.

Don't panic – bugs happen!

You'll run into problems, guaranteed. Your programs will have strange bugs that will leave you stumped. Trust me, I've been programming for 30 years and that still happens to me, too. We're only humans and our brains have a limited capacity to deal with complex programming problems. In this book, I'll give you tools for your mental toolbox that will allow you to find your way out of any hole you've dug for yourself.

Understanding beats copy-pasting

Too many people attempt to write iOS apps by blindly copy-pasting code that they find on blogs and other websites, without really knowing what that code does or how it should fit into their programs.

There's nothing wrong with looking on the web for solutions — I do it all the time — but I want to give you the tools and knowledge to understand what you're doing and why. That way, you'll learn more quickly and write better programs.

This is hands-on, practical advice, not a bunch of dry theory (although we can't avoid *some* theory). You're going to build real apps right from the start, and I'll explain how everything works along the way, with lots of images to illustrate what is going on.

I'll do my best to make it clear how everything fits together, why we do things a certain way and what the alternatives are.

Do the exercises

I'll also ask you to do some thinking of your own — yes, there are exercises! It's in your best interest to actually do these exercises. There's a big difference between knowing the path and walking the path... And the only way to learn programming is to do it.

I encourage you to not just do the exercises but also to play with the code you'll be writing. Experiment, make changes, try to add new features. Software is a complex piece of machinery and to find out how it works, you sometimes have to put some spokes in the wheels and take the whole thing apart. That's how you learn!

Have fun!

Last but not least, remember to have fun! Step by step, you'll build your understanding of programming while making fun apps. By the end of this book, you'll have learned the essentials of Swift, the iOS Software Development Kit (SDK) and both the SwiftUI and the UIKit frameworks. More importantly, you should have a pretty good idea of how everything goes together and how to think like a programmer.

This book has one ultimate goal: That by the time you reach the end, you'll have learned enough to stand on your own two feet as a developer. We're confident that, eventually, you'll be able to write any iOS app that you want as long as you understand the basics. You still may have a lot to learn but, when you're through with *The iOS Apprentice*, you can do it without the training wheels.

Is this book right for you?

Whether you're completely new to programming or you come from a different programming background and want to learn iOS development, this book is for you!

If you're a complete beginner, don't worry — this book doesn't assume you know anything about programming or making apps. Of course, if you do have programming experience, that helps. Swift is a new programming language but, in many ways, it's similar to other popular languages such as Python, C# or JavaScript.

If you've tried iOS development before with the old language, Objective-C, then its low-level nature and strange syntax may have put you off. Well, there's good news: Now that we have a modern language in Swift, iOS development has become a lot easier to pick up.

This book can't teach you all the ins and outs of iOS development. The iOS SDK is huge and grows with each new release, so there's no way we can cover everything. Fortunately, we don't need to. You just need to master the essential building blocks of Swift and the iOS SDK. Once you understand these fundamentals, you can easily figure out how the other parts of the SDK work and learn the rest on your own terms.



The most important thing I'll teach you is how to think like a programmer. That will help you approach any programming task, whether it's a game, a utility, a mobile app that uses web service or anything else you can imagine.

As a programmer, you'll often have to think your way through difficult computational problems and find creative solutions. By methodically analyzing these problems, you'll be able to solve them, no matter how complex. Once you possess this valuable skill, you can program anything!

iOS 13 and later only

The code in this book is written exclusively for iOS version 13 and later. Each new release of iOS is such a big departure from the previous one that it doesn't make sense to keep developing for older devices and iOS versions. Things move quickly in the world of mobile computing!

Most iPhone, iPod touch and iPad users are quick to upgrade to the latest version of iOS anyway, so you don't need to be too worried that you're leaving potential users behind.

Owners of older devices may be stuck with older iOS versions, but this is only a tiny portion of the market. The cost of supporting these older iOS versions for your apps is usually greater than the handful of extra customers it brings you.

It's ultimately up to you to decide whether it's worth making your app available to users with older devices, but my recommendation is that you focus your efforts where they matter most. Apple, as a company, relentlessly looks towards the future — if you want to play in Apple's back yard, it's wise to follow its lead. So back to the future it is!

What you need

It's a lot of fun to develop for the iPhone and iPad but, like most hobbies (or businesses!), it will cost some money. Of course, once you get good at it and build an awesome app, you'll have the potential to make that money back many times.

You'll have to invest in the following.



An **iPhone, iPad or iPod touch**. I'm assuming that you have at least one of these. iOS 13 runs on the following devices:

- iPhone 6s or newer
- iPad Pro (any generation)
- iPad 5th generation or newer
- iPad Air 2 or 3rd generation
- iPad Mini 5th generation
- iPod Touch 7th generation

If you have an older device, then this is a good time to think about getting an upgrade. But don't worry if you don't have a suitable device: You can do most of your testing on iOS Simulator.

Note: Even though this book is about developing apps for the iPhone, everything within applies equally to the iPad and iPod touch. Aside from small hardware differences, these devices use iOS or its close cousin iPadOS and you program them in exactly the same way. You should also be able to run the apps from this book on your iPad or iPod touch without problems.

A Mac computer with an Intel processor: Any Mac that you've bought in the last few years will do, even a Mac mini or MacBook Air. It needs to have at least macOS 10.14.4 Mojave, and I strongly recommend that you use macOS 10.15 Catalina, especially when building apps with SwiftUI.

Xcode, the development environment for iOS apps, is a memory-hungry tool. These days, even the most inexpensive MacBook Air in the Apple Store comes with 8GB of RAM. You should consider this the minimum RAM for development. Keep this general rule in mind: The more RAM, the better. A smart developer invests in good tools!

With some workarounds, it's possible to develop iOS apps on a Windows or a Linux machine, or on a regular PC that has macOS installed (a "Hackintosh"). If you're up for a challenge, you can try iOS development on these machines, but you'll save yourself a lot of time and hassle by just getting a Mac.

If you can't afford to buy the latest model, then consider getting a second-hand Mac from eBay or some other resale vendor. It should be a Mac that dates from mid-2012

or later; just make sure it meets the minimum requirements (Intel CPU, preferably with more than 4 GB RAM). If it helps, you should know that the code for the SwiftUI portions of this book was written using a mid-2014 MacBook Pro with 16GB of RAM. Should you happen to buy a machine that has an older version of macOS, you should upgrade to the latest version of the operating system from the online Mac App Store for free.

Eventually, an **Apple Developer Program account**: You can download all the development tools for free and you can try out your apps on your own iPhone, iPad or iPod touch while you’re developing, so you don’t have to join the Apple Developer Program just yet. But to submit finished apps to the App Store, you’ll have to enroll in the paid Developer Program. This will cost you \$99 per year.

See developer.apple.com/programs/ for more info.

Xcode

The first order of business is to download and install Xcode and the iOS SDK.

Xcode is the development tool for iOS apps. It has a text editor where you’ll type your source code and a visual editor for designing your app’s user interface.

Xcode transforms the source code that you write into an executable app and launches it in the iOS Simulator or on your iPhone. Because no app is bug-free, Xcode also has a debugger that helps you find defects in your code. Unfortunately, it won’t automatically fix them for you; that’s still something you have to do yourself.

You can download Xcode for free from the Mac App Store (apple.co/2wzi1L9). This requires at least macOS High Sierra (10.14.4). If you’re still running an older version of macOS, you’ll first have to upgrade to the latest version of macOS (also available for free from the Mac App Store). Get ready for a big download, as the full Xcode package is almost 8 GB.

Important: You may already have a version of Xcode on your system that came pre-installed with your version of macOS. That version could be hopelessly outdated, so don’t use it. Apple puts out new releases on a regular basis and you always want to develop with the latest Xcode and the latest available SDK on the latest version of macOS.

This revision of the book was written using **Xcode version 11** and the **iOS 13** SDK on **macOS Catalina (10.15) beta**. By the time you read this, the version numbers might have gone up again.



We'll do our best to keep the PDF versions of the book up-to-date with new releases of the development tools and iOS versions, but don't panic if the screenshots don't correspond 100% to what you see on your screen. In most cases, the differences will be minor.

Many older books and blog posts (anything before 2010) talk about Xcode 3, which is radically different from Xcode 11. If it predates the iPad, it's seriously out of date.

More recent materials may mention Xcode versions 4 through 10, which are similar to Xcode 11 but differ in many of the details. Xcode 11 also introduces a new framework, SwiftUI, and you won't find anything about it in older tutorials.

If you're reading an article and you see a picture of Xcode that looks different from yours, the author might be writing about an older version. You may still be able to get something out of those articles, as the programming examples are still valid. It's just Xcode that is slightly different.

What's ahead: An overview

The iOS Apprentice is spread across four apps, and moves from beginning to intermediate topics. You'll build each app from start to finish, from scratch! Let's take a look at what's ahead.

Building apps with SwiftUI

You'll build the first two apps using the newly-announced SwiftUI framework. Announced in 2019 at Apple's WWDC (World Wide Developer Conference), it's a completely different way for developers to design apps. Since it looks like SwiftUI will eventually become the preferred way to build apps, we decided to introduce it to you with your first two apps.

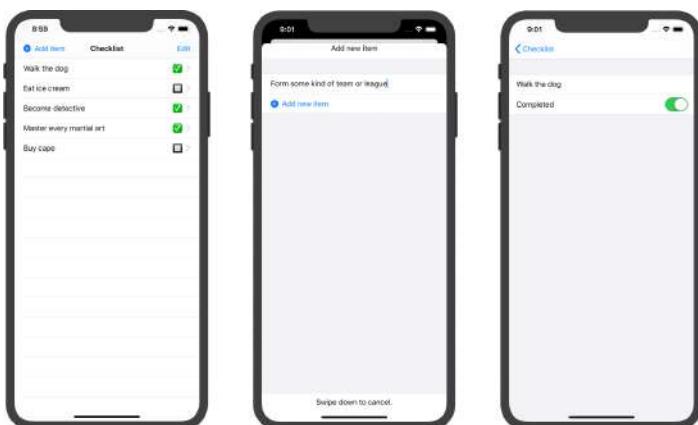
App 1: Bullseye

You'll start by building a game called *Bullseye*. You'll learn how to use Xcode, Swift and SwiftUI in a way that's easy to understand.



App 2: Checklist

For your next trick, you'll create your own to-do list app. You'll learn more about SwiftUI, the fundamental design patterns that all iOS apps use, data structures, sharing information between objects, and saving the user's information. Now you're making apps for real!



Building apps with UIKit

From the very first iPhone OS and all the way up to iOS 12, iOS apps were written using Apple's original user interface framework, UIKit. With over a decade's worth of UIKit-based code, documentation and tutorials, today's iOS developers can't rely on SwiftUI alone – they'll have to know both user interface frameworks. That's why you'll build the next two apps with UIKit.

App 3: MyLocations

For your third app, you'll develop a location-aware app that lets you keep a list of spots that you find interesting. In the process, you'll learn about UIKit, Core Location, Core Data, Map Kit and much more!



App 4: StoreSearch

Mobile apps often need to talk to web services, and that's what you'll do in your final app. You'll make a stylish app that lets you search for products on the iTunes store using HTTP requests and JSON.



Let's get started and turn you into a full-fledged iOS developer!

The language of the computer

The iPhone may pretend that it's a phone, but it's really a pretty advanced handheld computer that happens to have the ability to make phone calls. Like any computer, the iPhone works with ones and zeros. When you write software to run on the iPhone, you somehow have to translate the ideas in your head into those ones and zeros so that the computer can understand you.

Fortunately, you don't have to write any ones and zeros yourself. That would be a bit too much to ask of the human brain. On the other hand, everyday English — or any other natural human language — just isn't precise enough to use for programming computers.

So you'll use an intermediary language, Swift. It's a little bit like English, and reasonably straightforward for us humans to understand. At the same time, it's ordered and structured enough so that it can be easily translated into something the computer can understand as well.

This is an approximation of the language that the computer speaks:

```
Ltmp96:  
.cfi_def_cfa_register %ebp  
pushl %esi  
subl $36, %esp  
Ltmp97:  
.cfi_offset %esi, -12  
call L7$pb  
L7$pb:  
popl %eax  
movl 16(%ebp), %ecx  
movl 12(%ebp), %edx  
movl 8(%ebp), %esi  
movl %esi, -8(%ebp)  
movl %edx, -12(%ebp)  
movl %ecx, (%esp)  
movl %eax, -24(%ebp)  
call _objc_retain  
movl %eax, -16(%ebp)  
.loc 1 161 2 prologue_end
```

Actually, what the computer sees is this:

```
000110010100111101001000110011111001010  
001010001001111010110111001110101101001  
010100011100111110101110110000111000110  
100100000111000101001101001111001100111
```

The `movl` and `call` instructions are part of what's called *assembly language*, which is just there to make machine code more readable for humans. I don't know about you, but for me, it's still hard to make much sense out of it.

It certainly is possible to write programs in that arcane language. In the days of 8- and 16-bit computers, if you were writing a videogame or some other app that had to eke the most performance out of those slow machines, you had to. Even today, programmers who need to work at the system level or need maximum performance

from minimal hardware will write some assembly language. I'll take having my apps run fractions of a second slower if I can write programs that I can follow because they look like this:

```
func handleMusicEvent(command: Int, noteNumber: Int, velocity: Int) {  
  
    if command == NoteOn && velocity != 0 {  
        playNote(noteNumber + transpose, velocityCurve[velocity] /  
127)  
  
    } else if command == NoteOff ||  
        (command == NoteOn && velocity == 0) {  
        stopNote(noteNumber + transpose, velocityCurve[velocity] /  
127)  
  
    } else if command == ControlChange {  
        if noteNumber == 64 {  
            damperPedal(velocity)  
        }  
    }  
}
```

The above code snippet is from a sound synthesizer program. It looks like something that almost makes sense. Even if you've never programmed before, you can sort of figure out what's going on. It's almost English.

Swift is a hot new language that combines traditional object-oriented programming with aspects of functional programming. Fortunately, it has many things in common with other popular programming languages, so if you're already familiar with C#, Python, Ruby or JavaScript, you'll feel right at home with Swift.

Swift isn't the only option for making apps. Until recently, iOS apps were programmed in Objective-C, which first appeared in 1984. Objective-C is an object-oriented extension of the tried-and-true C programming language, which was first released in 1972. Objective-C comes with a lot of '70s and '80s baggage and doesn't have a lot of the niceties that modern developers have come to expect. That's why Apple created a new language.

Objective-C will still be around for a while, but the future of iOS development is Swift. All the cool kids are using it already.

C++ is another language that adds object-oriented programming to C. It's very powerful but, as a beginning programmer, you probably want to stay away from it. I only mention it because you can also use C++ to write iOS apps. There's also an unholy marriage of C++ and Objective-C named Objective-C++ that you may come across from time to time. If you see it, back away slowly and don't make eye contact.



I could have started *The iOS Apprentice* with an in-depth exploration of the features of Swift, but this is a tutorial, not a sleep aid! So, instead, I'll follow the adage of "Show, don't tell" and explain the language as we go along, very briefly at first, but in more depth later.

In the beginning, the general concepts — what is a variable, what is an object, how do you call a method, and so on — are more important than the details. Slowly but surely, all the arcane secrets of the Swift language will be revealed to you!

Are you ready to begin writing your first iOS app?

Chapter 2: Getting Started with SwiftUI

Joey deVilla

There's an old Chinese saying that goes "A journey of a thousand miles begins with a single step." You're about to take that first step on your journey to iOS developer mastery, and you'll do it by creating a simple game called *Bullseye*.

This chapter covers the following:

- **SwiftKit and UIKit:** These are two ways to build apps and user interfaces, and you'll learn both.
- **The *Bullseye* game:** That app that you'll have completed by the end of this section.
- **Getting started:** Enough preamble — let's create a new project!
- **Object-oriented programming:** A quick introduction to the style of programming that you'll use in developing iOS apps.
- **Adding interactivity:** An app that just sits there is no fun. Let's make it respond to the user!
- **State and SwiftUI:** What is "state," and what does it have to do with SwiftUI?
- **Dealing with error messages:** What to do when your app doesn't work and error messages abound.
- **The anatomy of an app:** A brief explanation of the inner workings of an app.



SwiftUI and UIKit

There's another saying (erroneously) attributed to the Chinese: "May you live in interesting times." Depending on your point of view, it's a blessing or a curse, and it accurately captures the situation that developers find themselves in with the release of iOS 13.

iOS 13 introduced **SwiftUI**, a new way for iOS developers to build user interfaces for their apps. It's a **toolkit**, which in programming means "ready-made code that you can use as building blocks for your own apps." Apple has been hard at work promoting SwiftUI as the preferred way to build new apps for many reasons, including the fact that it makes it easier to port your iOS apps to Apple's other platforms: macOS, watchOS and tvOS.

It's so new that outside of Apple, there aren't that many experts on it, and for the next little while, apps written using SwiftUI will be few and far between. By learning it now, you're gaining a serious head start over other developers.

UIKit is SwiftUI's long-standing predecessor. It's been around since iOS 2.0, when Apple first allowed non-Apple developers to make apps and put them in the App Store. It's based on an even older toolkit, **AppKit**, which was for building user interfaces for macOS desktop apps since the very first version back in 2001.

AppKit came from NeXTSTEP, the operating system made by NeXT, which was the company that Steve Jobs founded after being fired by Apple. Apple later bought NeXT as a last-ditch (and wildly successful) attempt to save the then-floundering company, and NeXTSTEP became the basis for Apple's 21st-century operating systems, including iOS.

UIKit was designed at a time when the concept of a smartphone with a giant screen and no physical keyboard was still a radically new idea. Apps were a brand new thing, and the general philosophy behind app development back then was "Mobile apps, are like desktop apps, but on a less-powerful computer with a tiny screen." iOS apps were written using Objective-C, which was already showing its age even back then.

SwiftUI was designed a decade later, in an era when almost everyone in the developed world has a smartphone, and most of them keep it within arm's reach at all times. Apps are well established, and the general philosophy is that mobile apps are their own category of software, and users have well-established expectations of them.

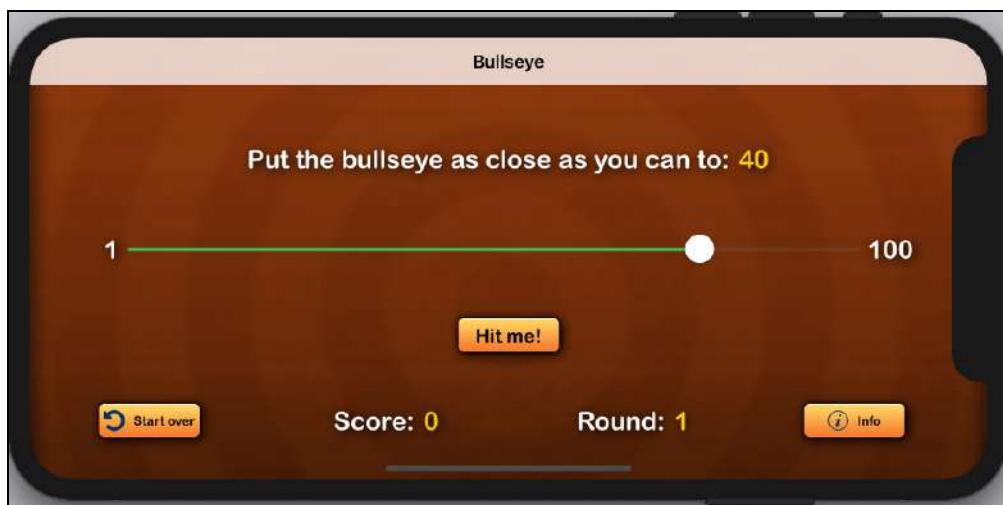
The preferred language for writing iOS apps is now Swift. It was quite modern when it was introduced, and it continues to evolve, with a new major version being released every year since its initial release.

Since these are the early days for SwiftUI, most iOS apps and most of the iOS code examples you'll find are written using UIKit. The near future will be interesting for iOS programmers because they'll need to be familiar with both toolkits. That's why this book covers SwiftUI *and* UIKit.

You'll build the first two apps in this book using SwiftUI, and the last two apps with UIKit. Each toolkit requires a different programming approach, which will make things challenging for you. We also hope that learning both will be rewarding and fun!

The Bullseye game

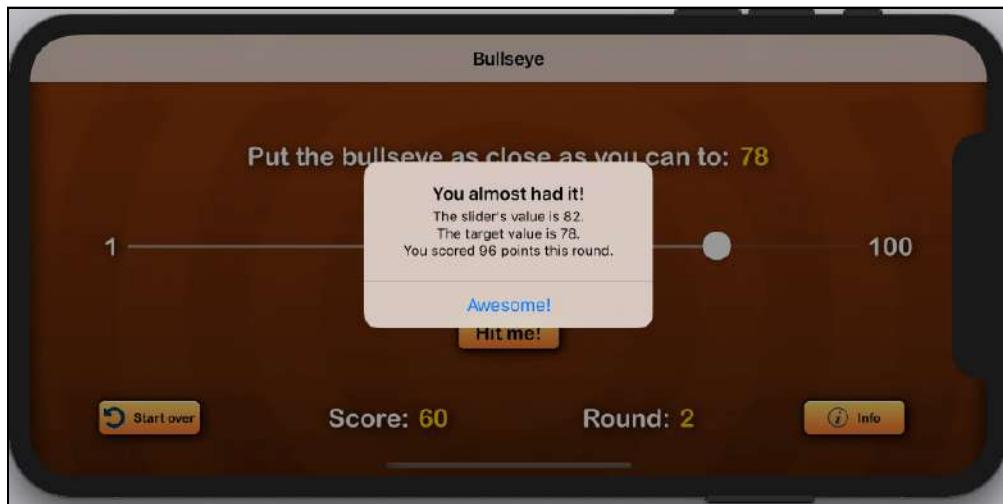
As we mentioned earlier, you're going to create a simple game called *Bullseye*. Here's what it'll look like when you're finished:



The finished Bullseye game

The objective of the game is to put the bullseye as close to the target as you can. The bullseye is on a slider that goes from 1 to 100, and the target value is randomly chosen. In the screenshot above, you're challenged to put the bullseye at 40. Since you can't see the current value of the slider and there aren't any markings to help you, you have to "eyeball" it.

When you're confident of your estimate, you press the **Hit Me!** button and a pop-up will tell you what your score is:



An alert pop-up shows the score

The closer to the target value you are, the more points you score. After you dismiss the alert pop-up by pressing the **OK** button, a new round begins with a new random target. The game repeats until the player presses the **Start Over** button (the one near the bottom-left corner), which resets the score to 0 and the round to 1.

This game probably won't make you an instant millionaire on the App Store, but it will show you the basics of making an app and building user interfaces with SwiftUI. Hey, even future millionaires have to start somewhere!

Making a programming to-do list

Now that you've seen what the game should look like, and what the gameplay rules are, make a list of all the things that you think you'll need to do in order to build this game. It's okay if you draw a blank, but try it anyway.

Here's an example:

The app needs to put the "Hit Me!" button on the screen and show an alert pop-up when the user presses it.

Try to think of other things the app needs to do. It doesn't matter if you don't actually know how to accomplish these tasks. The first step is to figure out *what* you need to do. *How* to do these things isn't important yet.

Once you know what you want, you can also figure out how to do it, even if you have to ask someone or look it up. But the *what* comes first.

You'd be surprised at how many people start writing code without a clear idea of what they're actually trying to achieve. No wonder they get stuck! Whenever you start working on a new app, it's a good idea to make a list of all the different pieces of functionality you think the app will need. This will become your programming to-do list. Having a list that breaks up a design into several smaller steps is a great way to deal with the complexity of a project.

Note: If you ever need a fancy way of saying that you're breaking down a big complex task into a set of smaller, simpler tasks, just say that you're performing *functional decomposition*.

You may have a cool idea for an app, but when you sit down to write the program it can seem overwhelming. There is so much to do, and there's always the question: "Where do I begin?" By cutting up the project into small steps, you make it less daunting. You may find that some of those small steps can be divided into even smaller steps. You can always find a step that is simple and small enough to make a good starting point and take it from there.

Don't worry if you find this exercise challenging. You're new to all of this! As you gain more programming experience, you'll find it easier to identify the different parts that make up a design and become better at splitting it into manageable pieces.

Here's an example of a to-do list based on the description of *Bullseye*:

- Put a button on the screen and label it "Hit Me!"
- When the player presses the "Hit Me!" button, the app has to show a pop-up that shows the player a score indicating how close they were to the target.
- Put text on the screen, such as "Score:" and "Round:". The score increases as the player earns more points, and the number of rounds increases with each attempt by the player.
- Put a slider on the screen with a range between the values 1 and 100. The player moves the slider as closely to the target value as they can.
- Come up with the target value at the start of each round and display it on the screen. This needs to be a random number between 1 and 100, inclusive.



- Determine the value of the slider (based on its position) after the player presses the “Hit Me!” button.
- Compare the value of the slider to the target value and calculate a score based on how far off the player is. Show this score in the alert pop-up.
- Put a "Start Over" button on the screen. Make it reset the score to zero and round to one.
- Put the app in landscape orientation.
- Make it look pretty.

There may be a detail or two missing from this list, but it's a good starting point. Even for a game as basic as this one, there are quite a few things you need to do. Making apps is fun, but it's definitely a lot of work, too!

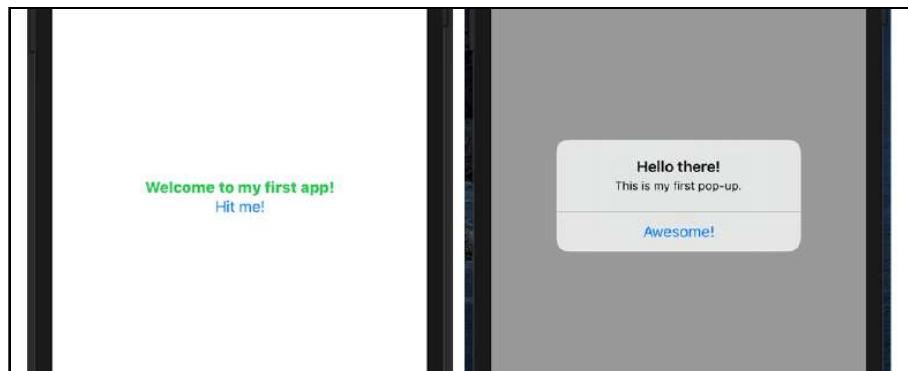
Getting started

The first two items on the *Bullseye* to-do list are, essentially:

1. Put a button on the screen.
2. Show a pop-up when the player presses the button.

You'll start by building an app that does only these two things. Once you've done this, you'll build the rest of *Bullseye* on this foundation.

This initial app will look like this:



The app contains a line of text and a single button (left) that shows an alert when pressed (right)

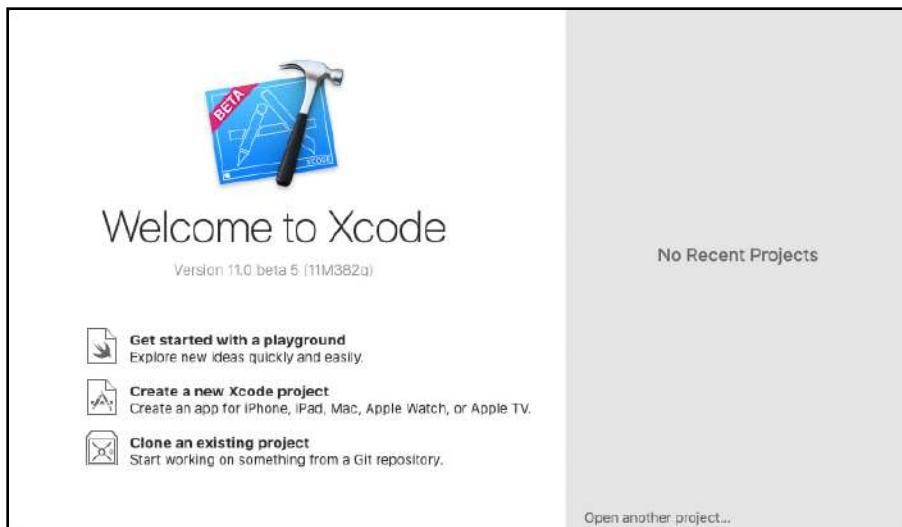
Time to start coding! To follow the coding exercises in this book, you'll need:

- **Coding tools.** The exercises in this book require **Xcode 11.0** or later, which you can download for free using the **App Store** on your Mac. Xcode 11 requires **macOS 10.15**, also known as “Catalina”. It won’t run on prior versions of macOS. The differences between Xcode 11 and earlier versions are so big that we don’t recommend using an earlier version.
- **Optionally – but *ideally* – a device.** The apps you’ll make will run on an iPhone running **iOS 13.0** or later, or an iPad running **iPadOS 13.0** or later. If you don’t have a device, you can make do with the **Simulator**, an application that runs on your Mac and acts as if it were an iPhone, iPad, Apple Watch, or Apple TV. It allows you to test the apps you write without having to deploy it to a device. You can do the exercises in this book without a device with a few limitations, but there’s no substitute for the real thing.

Creating a new project

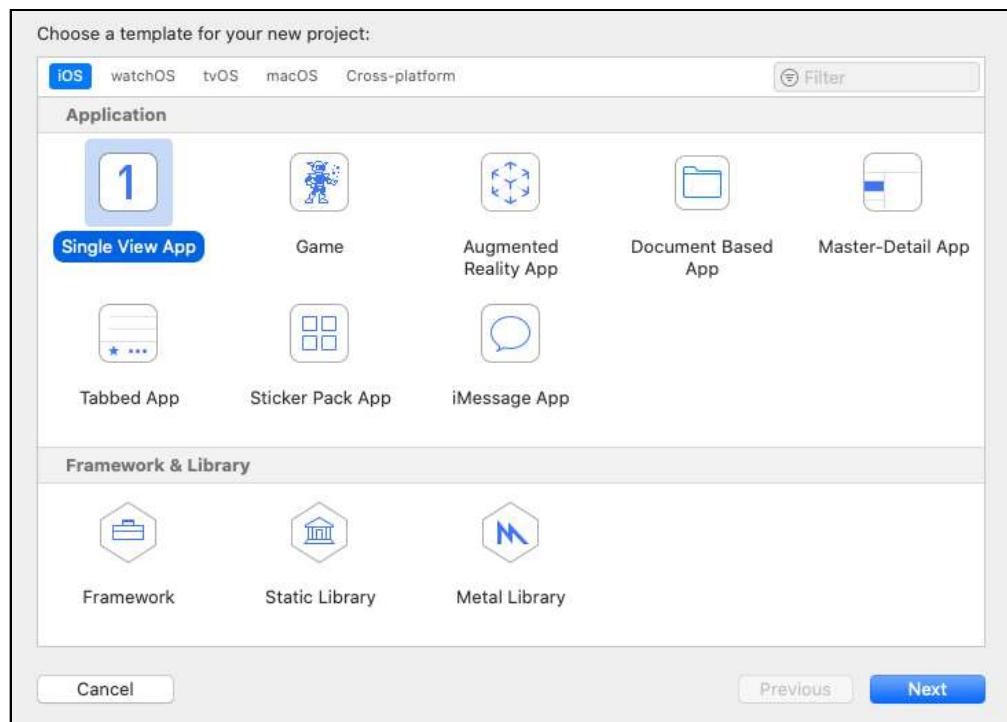
► Launch **Xcode**. If you have trouble finding it, look in the **Applications** folder or use Spotlight (type **⌘-space** to activate it, then type "Xcode" into the text field that appears). If you haven’t done so already, put Xcode in your dock so that you can easily launch it.

You’ll see the “Welcome to Xcode” window when it starts:



Xcode bids you welcome

- Choose **Create a new Xcode project**. The main Xcode window appears with an assistant that lets you choose a template:



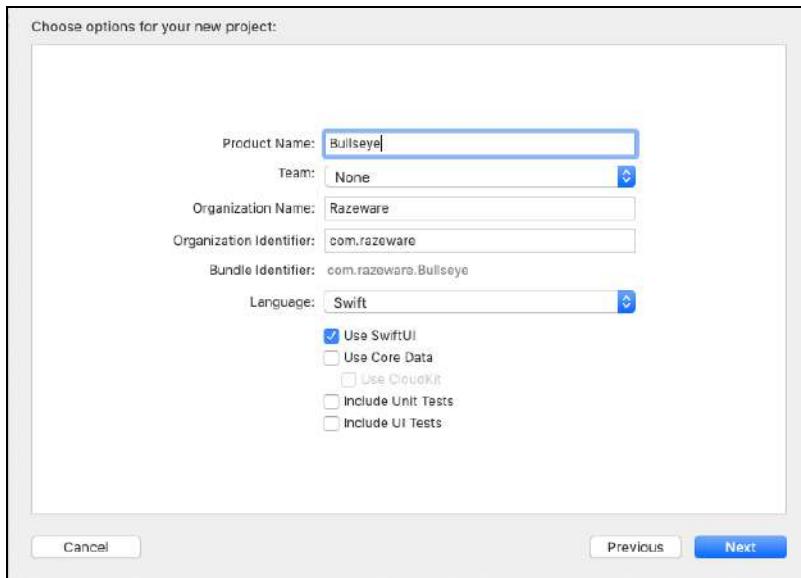
Choosing the template for the new project

Xcode comes with templates for a variety of app styles, each of which is pre-configured with code for a different kind of application. When you choose one of these templates, Xcode creates a new project that includes the **source files** — files containing the code that make up an app — that are necessary to create the kind of app you selected. These templates are handy because they're ready-made starting points that can save you a lot of effort.

- Select **Single View App** and click **Next**.

Single View App is the the simplest of the iOS app templates. It's a simple app with a single view — which is the term we tend to use for “screen” or “page” — that displays the text “Hello World”. You’ll use this as the basis for *Bullseye*.

You'll be shown a pop-up where you enter options for your new project.



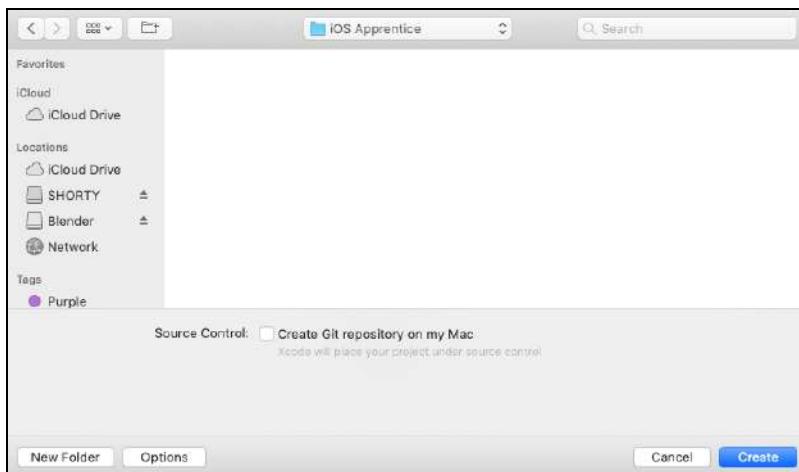
Configuring the new project

► Fill out these options as follows:

- **Product Name:** Enter **Bullseye** here.
- **Team:** If you're already a member of the Apple Developer Program, this will show your team name. For now, it's best to leave this set to **None**. We'll cover this in more detail later on.
- **Organization Name:** Put your own name or the name of your company here.
- **Organization Identifier:** You should fill this with something that uniquely identifies you or your organization. The standard practice is to enter your personal or organization domain name in reverse here. For example, if your domain name is *mydomain.com*, enter **com.mydomain** into this field. If you don't have your own domain name, enter **com.example**. Don't worry too much about what you enter here right now, it doesn't really matter until you submit your app to the App Store, and you can always change this setting later.

- **Language:** Make sure that this is set to **Swift**. All the exercises in this book are in Swift.
- **Use SwiftUI:** Make sure that this is selected. You'll use SwiftUI to create the user interface for *Bullseye*.
- **Use Core Data, Include Unit Tests, and Include UI Tests:** Make sure that these are *not* selected. You won't use any of these features in this project.

► Press **Next**. Now, Xcode will ask where to save your project:



Choosing where to save the project

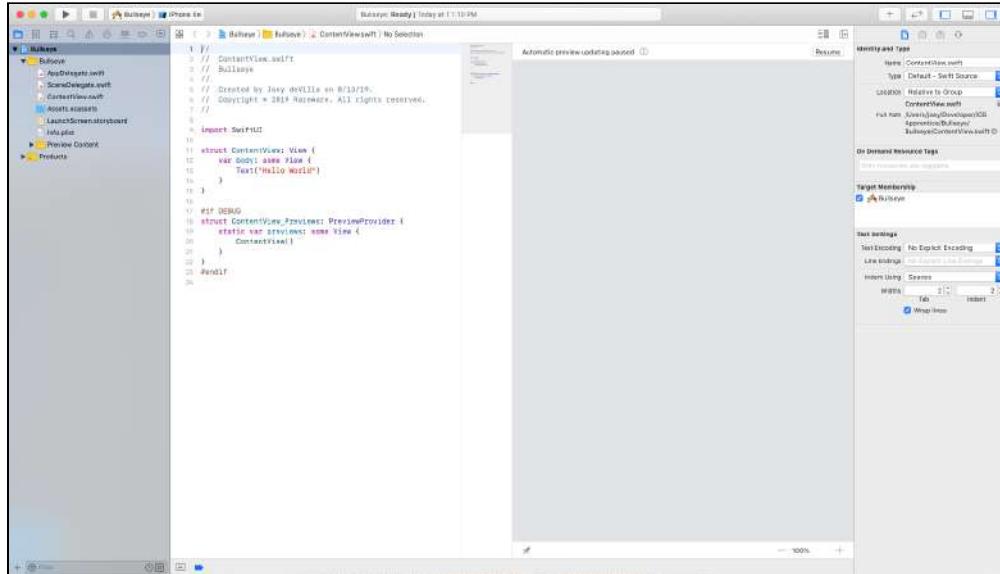
► Choose a location for the project files. For example, the Desktop or your Documents folder.

Xcode will automatically make a new folder for the project using the Product Name that you entered in the previous step, **Bullseye** in this case, so you don't need to make a new folder yourself.

At the bottom of the File Save dialog, there is a checkbox labeled **Create Git repository on My Mac**. You can ignore this for now. You'll learn about the Git version control system later on.

► Click **Create** to finish.

Xcode will now create a new project named "Bullseye," based on the Single View Application template, in the folder you specified. When it is done, the screen should look something like this:



The main Xcode window at the start of your project

There may be small differences between the screenshot above and what you see on your own computer. As long as you're running Xcode version 11.0 or later, any differences you see should only be superficial.

Important: Before you continue, examine the list of files on the left side of the Xcode window. If you see a file named **ContentView.swift**, your project is set up properly and you can proceed to the next step. If you don't see a file named **ContentView.swift** in the list, but instead see **ViewController.swift**, it means that you forgot to check the **UseSwiftUI** checkbox when choosing the options for the project. If you see files with the names **ViewController.h** and **ViewController.m**, then you picked the wrong language (Objective-C) when you created the project. In either case, start over and be sure to check the **UseSwiftUI** checkbox and choose **Swift** as the programming language.

Now, let's take a closer look at your project.

Looking at the Editor

The first thing you should look at is the **Editor**, which takes up most of the left side of the Xcode window:



```
1 /**
2 //  Content View.swift
3 //  Bullseye
4 //
5 //  Created by Joey deVilla on 8/13/19.
6 //  Copyright © 2019 Razeware. All rights reserved.
7 //
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Hello World")
14     }
15 }
16
17 #if DEBUG
18 struct ContentView_Previews: PreviewProvider {
19     static var previews: some View {
20         ContentView()
21     }
22 }
23 #endif
24
```

The Editor in a newly created Single View Application project

You'll spend a lot of time in the Editor, as it's where you enter — and as its name implies; *edit* — code. Right now, it's displaying the source code inside the **ContentView.swift** file. This file defines what goes into and what happens on the app's single "screen" or "page," which is also referred to in programming terms as a *view*.

For now, you should concern yourself with the part of the code that actually determines what the app does. It's the middle section, shown below:

```
import SwiftUI
struct ContentView : View {
    var body: some View {
        Text("Hello World")
    }
}
```

Don't worry if this makes no sense to you right now, we'll review this line-by-line later in this chapter. However, here's a quick sneak peek if you're especially curious:

The first line, `import SwiftUI`, is an instruction to make use of the SwiftUI toolkit. SwiftUI provides a lot of features that you can call on to make user interfaces and respond to user actions.

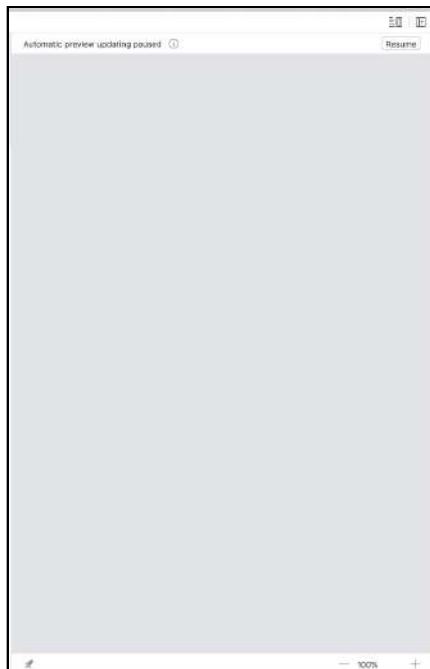
The rest of the code defines the app's single view (remember, in this case, when we say "view", you should think "screen" or "page"). It says that there is a thing called `ContentView` and this it's a `View`. It also says that `ContentView` contains a single `Text` object that displays the text "Hello World".

Note that the source code above leaves out the part at the beginning: The handful of bluish-gray lines of text ending with a copyright notice. This is a block of *comments*, which are notes intended for people who will read the code. They have no effect on the app or how it runs. We'll discuss comments in more detail in the following chapter.

The source code above also leaves out the part at the end that starts with the line `#if DEBUG`. This code is responsible for drawing a preview of your app, and we'll play around with it later.

Looking at the Canvas

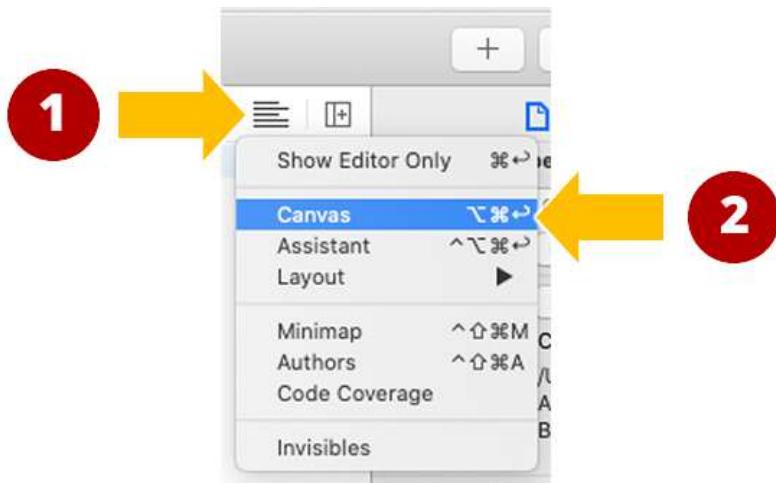
It's often difficult to get an idea of what a view would look like just by looking at its code. That's what the **Canvas** is for. It's located just to the right of the Editor and looks like this:



The Canvas pane at the start

The Canvas pane shows the message **Automatic preview updating paused** in its upper left corner. This means that it's currently *not* updating its contents to show the visual results of the code in the Editor. Let's un-pause it so that you can see what the view should look like.

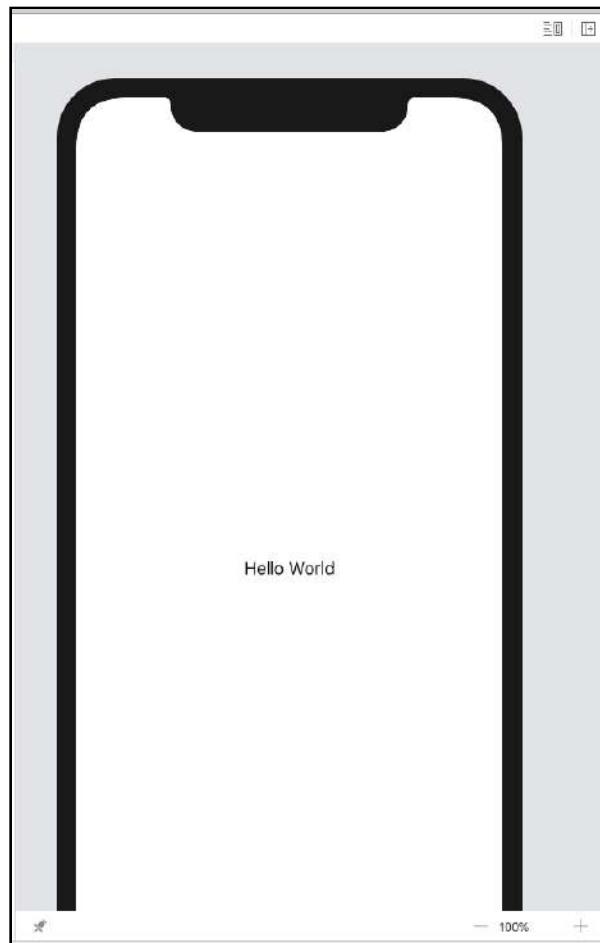
Note: If you don't see the Canvas in the Xcode window, click the **Editor Options** button. A menu will appear; select **Canvas**:



The Editor Options button and menu

- Click **Resume** (in the upper right corner of the Canvas pane) to see a preview of the app's view.

A spinning progress indicator will appear in the Canvas pane and, after a few moments, you should see this:



The Canvas pane after pressing the Resume button

The canvas is a much easier way to visualize your user interface, especially as your views become more complicated. As you'll learn later, you can edit your user interfaces in the canvas too!

Running your project

Now, let's bring your project to life by running it in the Simulator.

Once again, the Simulator is a macOS application that pretends to be various Apple devices: iPhones, iPads, Apple Watches, and Apple TVs. It's useful for running quick tests of your apps and for trying it out on devices that you don't have. In the beginning, you'll run your apps on the Simulator. Later on, you'll learn how to deploy them to a device.

- Click on the device picker near the top-left corner of the Xcode window:



The device picker

- In the menu that appears, under the section marked **iOS Simulators**, select **iPhone XR**:



The device picker menu with iPhone XR selected

- Click the **Run** button near the top-left corner of the Xcode window, to the left of the device picker:



Press Run to launch the app

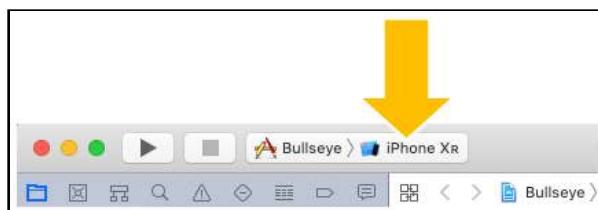
Note: If this is the first time you're using Xcode, it may ask you to enable developer mode. Click **Enable** and enter your password to allow Xcode to make these changes. Also, make sure that you do not have your iPhone or iPad plugged into your computer at this point. Otherwise, Xcode might try to run the app on the actual device instead of the Simulator. Since you're not yet set up for running on a device, this could result in errors that might leave you scratching your head. Stick with the Simulator for now; you'll learn how to deploy the app to your phone later.

Xcode will labor for a bit, and will eventually launch your brand new app in the Simulator. The app doesn't look like much — and there's not much you can do with it, either. That said, it's an actual running app, and an important first milestone in your journey!



What an app based on the Single View Application template looks like

If the app doesn't run and Xcode says **Build Failed** or **A build only device cannot be used to run this target** when you click the Run button, make sure that **iPhone XR** (or any other iPhone model listed under **iOS Simulators** in the device picker's menu) — not **Generic iOS Device** — is selected in the device picker:



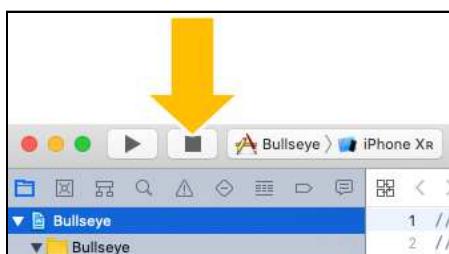
Making Xcode run the app on the Simulator

Until you press **Stop**, Xcode's Activity viewer at the top says, “Running Bullseye on iPhone XR”:



The Xcode activity viewer

- Click the **Stop** button to exit the app:



Press the stop button to stop the app

On your phone, or in the Simulator, you'd use the Home button to exit an app. On the Simulator, you could also use the **Hardware ▶ Home** item from the menu bar or use the handy $\Delta + \text{⌘} + \text{H}$ shortcut), but that won't actually terminate the app. It will disappear from the Simulator's screen, but the app stays suspended in the Simulator's memory, just as it would on a real iPhone.

It's not really necessary to stop the app. You can go back to Xcode and make changes to the source code while the app is still running. However, these changes won't become active until you click **Run** again. This will terminate any running version of the app, build a new version and launch it in the Simulator.

What happens when you click **Run**?:

Xcode will first *compile* your source code — that is, translate it from Swift into executable binary code. Languages like Swift, which are called *high-level languages*, are for human programmers, who are better at things like creativity and the overall design and logic of the application that they're creating.

Executable binary code — also called machine code — is a *low-level language*; it's for processor chips like the one in the iPhone (and the simulated one in the Simulator), which are better at things like performing up to trillions of math calculations in a second. As you might expect, human and machine languages are quite different, and so a translation step, called *compilation*, is necessary.

The *compiler* is the part of Xcode that converts your Swift source code into machine code. It also gathers all the other components that go into an app, which can include things such as images, icons, and sounds, and puts them into the *application bundle*. The application bundle contains everything the app needs to run.

This entire process is also known as *building* the app. If there are any errors come up during compilation (spelling mistakes in your code are a common cause of these), the build will fail. If compilation finishes without any errors, Xcode creates the application bundle, and then copies to its target — either the iPhone or the Simulator — and launches the app. All that happens with a single press of the **Run** button!

Changing the text

It's a long-time computer programming tradition to write a program that simply displays "Hello, world!" when learning how to program in a new language or for a new platform. That's why Apple made it part of the Single View Application template. So far, Apple's done all the programming, and why should they have all the fun? Let's take the app they've provided and use it to make your own.

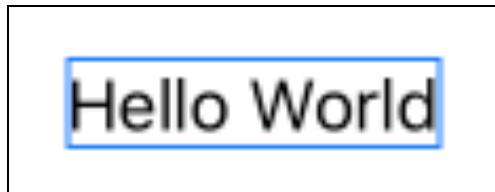
Let's update the text and make it a little less generic.

- In the Canvas, click on “Hello World”:



'Hello World', highlighted in both the Editor and the Canvas

Notice that “Hello World” is highlighted in both the Canvas and the Editor. In the Canvas, the highlighting looks like a fine blue rectangle drawn around “Hello World”:



'Hello World', highlighted in the Canvas

And in the Editor, the line `Text("Hello World")` is highlighted:

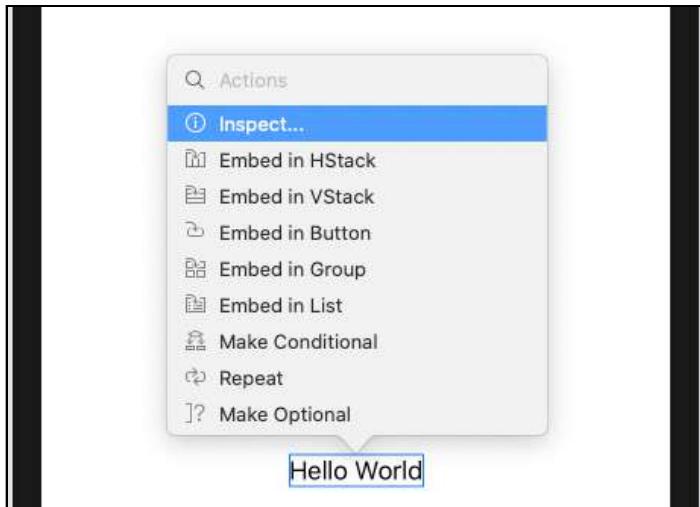


'Hello World', highlighted in the Editor

“Hello World” is highlighted in both because they're different ways of looking at the same things. The Editor shows you the user interface in the form of code, while the Canvas shows you the user interface as it will appear to the user.

Let's make changes to the text using both the Canvas and the Editor.

- In the Canvas, Command-click on “Hello World”. A pop-up menu appears, listing a number of actions you can take:



'Hello World', after being Command-clicked

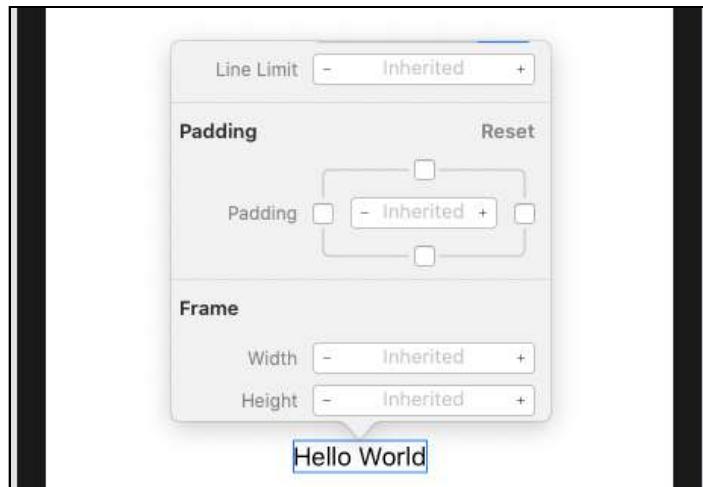
- Click the **Inspect...** item in the pop-up menu:



'Hello World' and the inspector

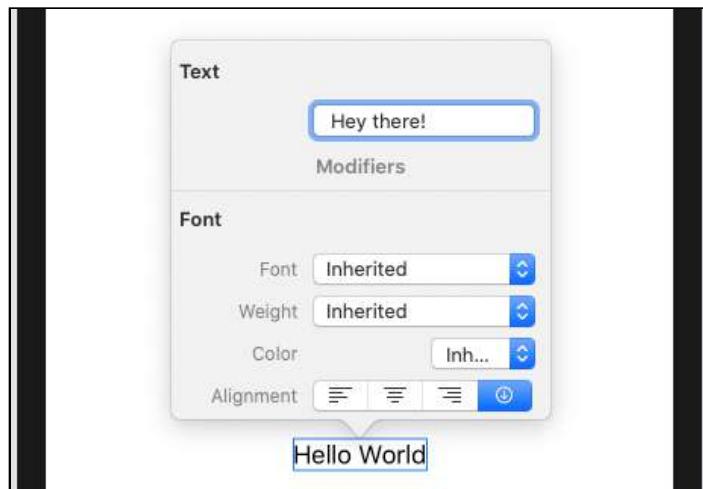
This brings up the Inspector, which displays the properties of “Hello World” and lets you change them. It may not be immediately apparent, but the Inspector is bigger than it appears.

If you scroll while the cursor is over the Inspector, you can see all the properties:



Scrolling the Inspector to the bottom

- Make sure the Inspector is scrolled to the top and enter “Hey there!” into the text field just below the **Text** heading:



Editing the text to say 'Hey there!'

- Click anywhere on the Inspector to dismiss it.



'Hey there', highlighted in the Editor and Canvas

"Hello World" is now "Hey there!" The change you made in the Canvas is reflected in the Editor. The line that once read `Text("Hello World")` now reads `Text("Hey there!")`.

Note: If the Canvas hasn't updated itself to show the new text, click the **Resume** button near the upper-right corner of the Canvas.

This ability to edit works both ways. Let's try another change to the text, this time using the code.

- In the Editor, change `Text("Hey there!")` to the following:

```
Text("Welcome to my first app!")
```

The section of code near the line you just changed should now look like this:

```
struct ContentView : View {
    var body: some View {
        Text("Welcome to my first app!")
    }
}
```

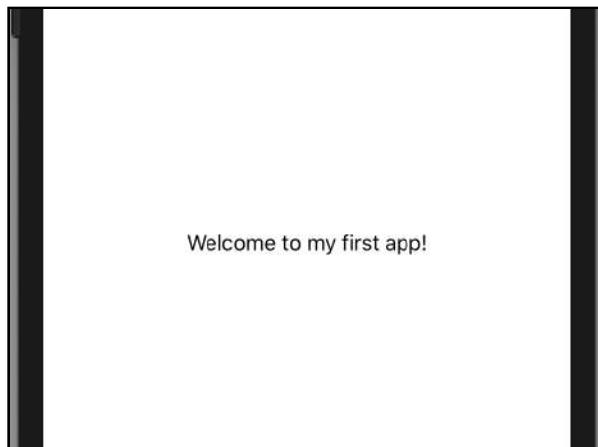
The change you made in the Editor is reflected in the Canvas:



'Welcome to my first app!' in the Editor and Canvas

- Click the **Run** button.

You should see your changes when the app starts in the Simulator:



Simulator displaying Welcome to my first app!

Making the text bolder

The text needs some sprucing up. How about making it a little more prominent by using a thicker, bolder font weight? Let's try to do that, using both the Canvas and the Editor.



In the Canvas, Command-click on “Welcome to my first app!” and select **Inspect...** from the pop-up menu.



'Welcome to my first app!' and the Inspector

- In the **Weight** menu, select **Semibold**:



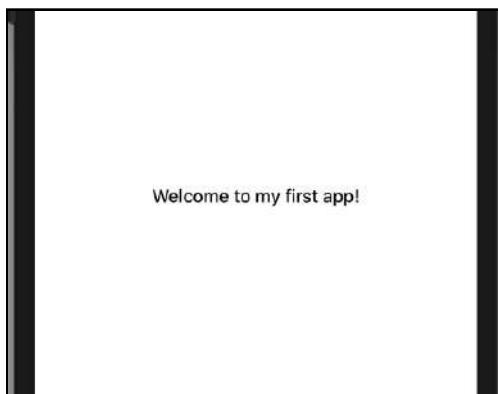
Choosing a new font weight for 'Welcome to my first app!'

- Click anywhere on the Inspector to dismiss it.



'Welcome to my first app!' with semibold font weight, in the Editor and Canvas

- Click the **Run** button. “Welcome to my first app!” is now a little more pronounced:



'Welcome to my first app!' with semibold font weight, in the Simulator

Once again, the change you made in the Canvas has a matching change in the Editor. The line that defines the text now reads like this:

```
Text("Welcome to my first app!")  
    .fontWeight(.semibold)
```

The newly added `.fontWeight(.semibold)` is a **method** that changes the **Weight** property of “Welcome to my first app!”. You’ll look at methods in a little more detail soon, but in the meantime, think of them as small pieces of code that belong to an object which make that object perform a specific task.

You use a method by *calling* it. This is done by first specifying the name of the object whose method you want to call, followed by a period (.), followed by the name of the method.

Hint: Any time you see the . character in Swift code, think of it as being shorthand for “Use this method on the object I just mentioned”. For example, you should read `Text().fontWeight()` as “Use the `fontWeight` method of my `Text` object.”

`fontWeight` is one of many methods that belong to the `Text` object. It changes the weight of its `Text` object’s font to the setting you provide it with. In this particular case, that setting is `.semibold`, which is a pre-defined value describing a font weight best described as “bold, but not too bold.”

Note: `.semibold` is the short version for the font weight setting. Its full name is `Font.Weight.bold`. Swift lets us get away with omitting the `Font.Weight`. part is because it knows that the only kind of settings you can provide the `fontWeight()` method are the `Font.Weight` kind.

Swift considers `.fontWeight(.semibold)` to be part of the same line as `Text("Welcome to my first app!")`. However, adding it as its own indented line makes it easier to read. The indentation is a convention that programmers use to say “this line is a continuation of the previous one,” and the Swift compiler simply ignores it.

Note: If you’re in an experimental mood, try editing the code for the text so that it’s all in a single line:

```
Text("Welcome to my first app!").fontWeight(.semibold)
```

and then press the **Run** button. The app still works, which means that Swift didn’t find anything wrong with the change. Once you’ve confirmed that the code works, change it back to its original formatting.

You’re probably asking “Why use formatting that the compiler’s going to ignore, anyway?” The answer is best summed up by notable MIT computer science professor Harold Abelson, who wrote “Programs must be written for people to read, and only incidentally for machines to execute.”

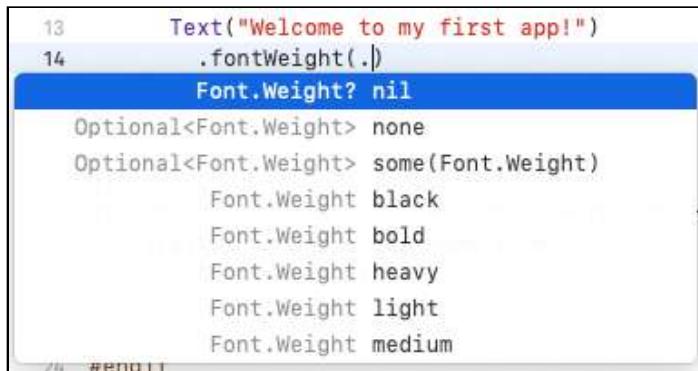
The compiler doesn't care about the how the code is formatted, but a little formatting makes it easier for humans to read, understand, make changes to, and more easily find errors in code.

Now that you've made a change to the font weight of "Welcome to my first app!" in a graphical way, it's time to try it in code.

- In the Editor, change the lines of code starting with `Text` so that the font weight is `black` instead of `semibold`:

```
Text("Welcome to my first app!")  
    .fontWeight(.black)
```

If you made the change by entirely deleting `.semibold`, a pop-up appeared when you typed in the `.`:

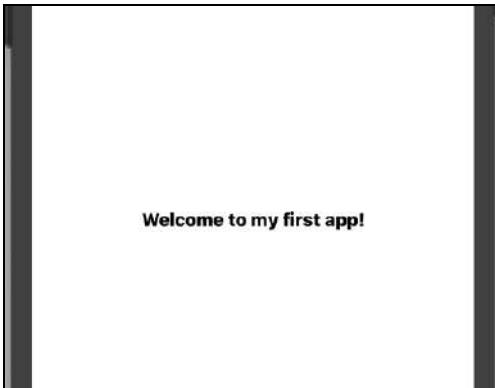


Code completion appearing when changing the font weight

This is Xcode being helpful with its *code completion* feature. It knows that what values you can put into `.fontWeight` modifier and provides them for you in a handy list. If Xcode presented this to you, choose **Font.Weight black** from the list.

Once you've made the change to the code, you'll see a matching change in the Canvas. (If you don't, click the Canvas' **Resume** button.)

- Click the **Run** button. “Welcome to my first app!” is even more pronounced:



'Welcome to my first app!' with black font weight, in the Simulator

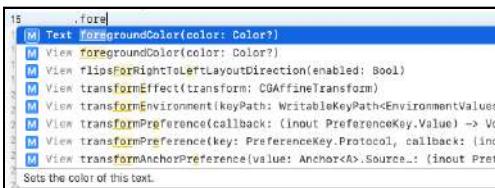
Changing the text's color

Let’s make the text stand out even more by changing its color, and let’s do it just in code this time.

- In the Editor, change the lines of code starting with `Text` so that it looks like the following:

```
Text("Welcome to my first app!")
    .fontWeight(.black)
    .foregroundColor(.green)
```

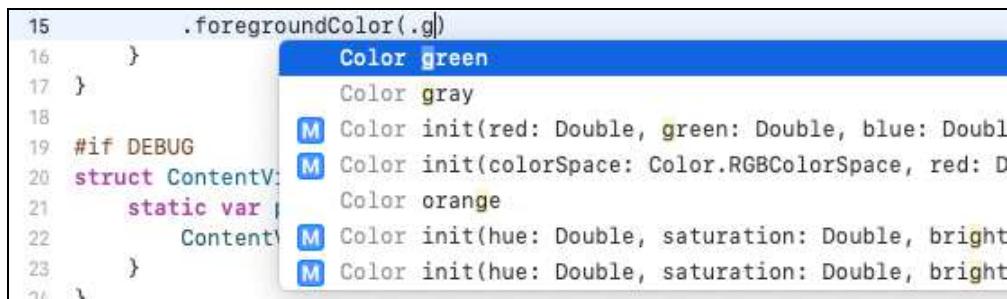
While entering the code, as you type the `.` character that comes before `color`, Xcode will try to help you by presenting a code completion pop-up. It will present a list of features of the `Text` object that you might want to use:



Code completion appearing adding a new modifier to 'Welcome to my first app!'

You should either choose or type in the `color()` method, which sets the color of the `Text` object’s text to the color you specify. You specify that color between the parentheses — the `(` and `)` characters.

When you type the `.` inside these parentheses, Xcode presents a different list:



Code completion appearing adding specifying a color for 'Welcome to my first app!'

Choose or type in `.green`, and then check the Canvas to confirm that “Welcome to my first app!” is now green. Once again, you might need to click the **Resume** button near the upper right corner of the Canvas.

- Click the **Run** button. Just like the Canvas, “Welcome to my first app!” is now green in the app.

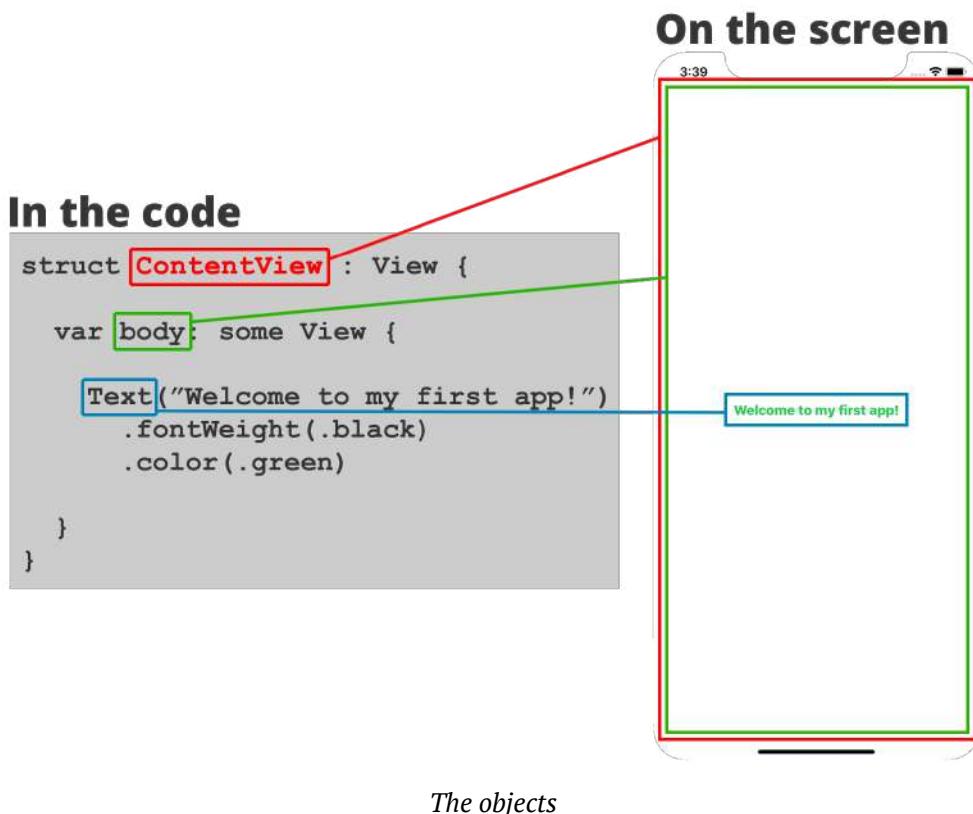
Object-oriented programming

Before you continue with the app, it’s time to look a little more closely at the topic of **object-oriented programming**. You may not realize it, but you’ve already been doing it!

Objects

Object-oriented programming is an approach that tries to manage the complexity of writing programs by dividing them into *objects*. Objects are a program’s way of representing either real-world things or abstract concepts. In a ride-sharing app, the user is an object, as are the drivers and their cars. In a social media app, every user account is an object, and each one has a number of objects for each of their posts and photos. In that game where the objective is to clear the current level by rearranging matching candies into groups, the candies are objects, and so is the board where the player rearranges the candies.

In your app, you've already been working with a number of objects, both in the code and on the screen:



Let's take a look at the code that defines one of these objects: the `ContentView`. We've looked at this once before, but this time we'll go into more detail.

```
struct ContentView : View {
```

The whole line, translated from Swift to plain language, means “This is the definition of a `struct` named `ContentView`, and it’s a `View`.”

This translation is still a little technical, and could use some further explanation. The word `struct` is short for *structure*, which is a description of a kind of object in Swift. Swift has other kinds of objects that you'll learn about later in this book, such as `class`.

A `View` is a kind of object that comes built-in with SwiftUI, and it represents anything that's drawn onscreen, including the screen itself. Views can contain other views.

Since `ContentView` is a `View`, it has the same properties and can have the same methods that `View` has. Any properties and methods defined inside `ContentView` is *in addition to* any properties and methods it gets from being a `View`.

Hint: Any time you see the `:` character in Swift code, think of it as being shorthand for “is a”. For example, you should read `ContentView : View` as “`ContentView` is a view.”

Now that you have a rough idea on how to define objects, let's move on to discussing how you can make them **know things** and **do things**: By discussing properties and methods.

Properties and methods

Objects are made up of at least one of these two kinds of things:

1. **Properties:** Without data, computer programs have nothing to work with. Properties are where objects store their data. To continue with the ride-sharing app example, the user objects have properties such as the user's name and current location. Each car object would have properties that store the make, model and year of the car, as well as its current location, and if applicable, the current passenger (which would be a user object). **Properties are the things that objects know.**
2. **Methods:** These are groups of code that perform actions, often on the data in the object. In the example of the ride-sharing app, the user object might have a method to update the user's current location. Each car object may have a method to update the car's current location, or calculate its distance from a given user. That user, once again, would be a user object, and the calculation would require the current location data from that user object. Remember, to call a method, you first specify the name of the object whose method you want to call, followed by a period `(.)`, followed by the name of the method. **Methods are the things that objects do.**

Let's go back to reviewing the code. You'll see that it already includes examples of both properties and methods!



Let's start with properties. Right now, `ContentView` defines a property in addition to the properties it gets from being a `View`. That property is `body`, and it's an object that acts as the container for all the objects on the screen that `ContentView` represents.

```
var body: some View {
```

This line, translated from Swift to plain language, means "This is the definition of a variable named `body`, and it's a `some View`." Basically, this is letting `ContentView` know what to display as its main body: The text field.

`var` is short for *variable*, which means two things:

1. It's a container for data.
2. Its contents can change.

Unlike `ContentView`, whose definition says that it's a `View`, the definition of `body` says that it's a `some View`. The `some` in front of `View` broadens the possibilities for `body`. Without getting into too much detail for now, it means that `body` can contain either a `View` or something that *behaves like* a `View` (meaning that it's an object *isn't* a `View`, but can have the same properties and methods as `View`).

The contents of `body` are currently defined by these lines, which also include some examples of calling methods:

```
Text("Welcome to my first app!")
    .fontWeight(.black)
    .color(.green)
```

The first line creates a `Text` object, which is a piece of read-only text that gets drawn on the screen. It also sets the `Text` object's content — which is one of its properties — to the text "Welcome to my app!"

The next two lines are examples of calling methods. Specifically, they call two methods on the `Text` object:

1. The first line calls the `Text` object's `fontWeight()` method, and provides it with a value representing the font weight `black`. In response to the message, `fontWeight()` applies the requested change in font weight to the original `Text` object, creating a new, bolder `Text` object.
2. The second line calls the new, bolder `Text` object's `color()` method, and provides it with a value representing the color `green`. In response to the message, `color()` applies the requested change in color to the new, bolder, `Text` object, creating an even newer green `Text` object with the same font weight.



This is the final Text object, which is drawn in the view.

This process is illustrated in the diagram below:



Method chaining explained

The process of applying two or more methods to an object, one after the other, is called *method chaining*. If you've done programming in JavaScript, you've probably seen this before.

You've already done some object-oriented programming up to this point:

- You've created — or as programmers would say, *instantiated* — a Text object that says "Welcome to my app!".
- You called on the Text object's methods to perform tasks: `fontWeight()` to make its text bold, and `color()` to change its color.

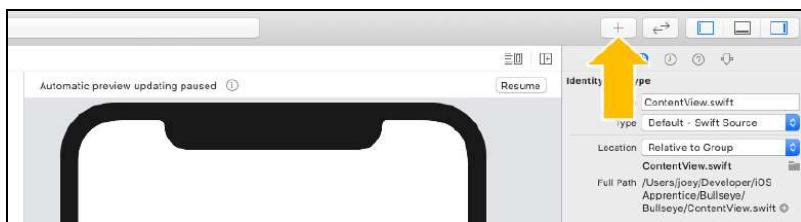
Phew — that was a lot to cover. Don't worry if you don't get everything right away or if you have trouble remembering everything; there's a lot concepts here that may be brand new to you. We'll review these concepts again and again through the book until they feel like second nature. Again: it's all about **learning via repetition**.

Adding interactivity

Right now, the app simply displays text and then just sits there. That simply won't do: It's time to add some interactivity! You'll do this by adding a button labeled "Hit me!", which was one of the key items on the to-do list for the app.

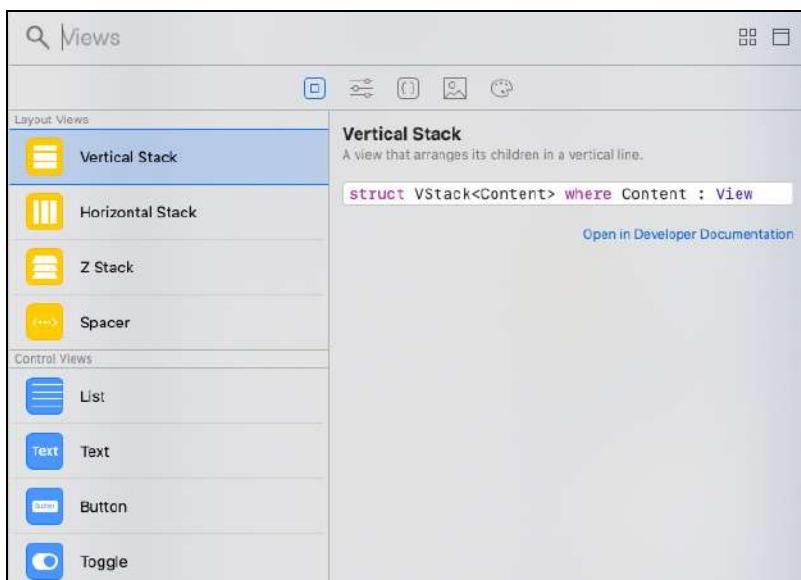
You could add the button by adding code into the Editor, but you should also learn how to add user interface items the drag-and-drop way, using the Library and Canvas.

- Press the + button located near the upper right corner of the Xcode window. That's the **Library** button:



The Library button

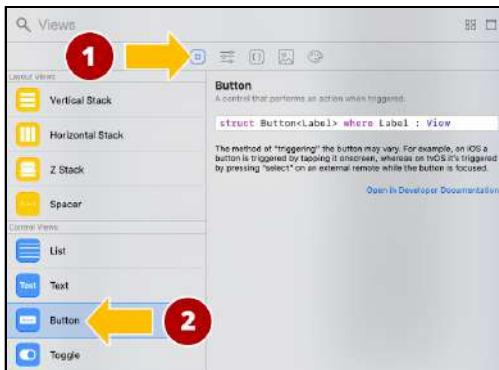
The window for the **Library** will appear:



The Library window

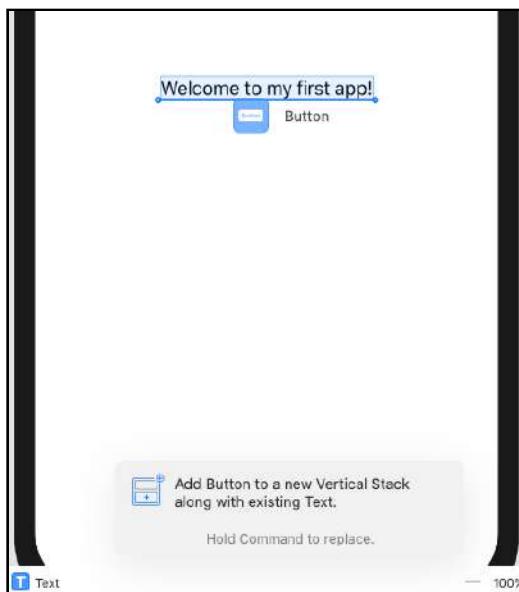
The Library is a collection of useful pre-made resources that you can drag and drop into your projects. These resources are divided into five major categories — Views, Modifiers, Snippets, Media, and Color — with one tab for each. It's good for experimenting with ideas, and in a number cases, can save you some typing.

- With the Library window in view, make sure that the **Views** tab (the leftmost one, with the square-within-a-square icon) is selected. Remember, a view is anything that can be drawn on the screen, which means that a button is a view. Highlight the **Button** item in the Library list:



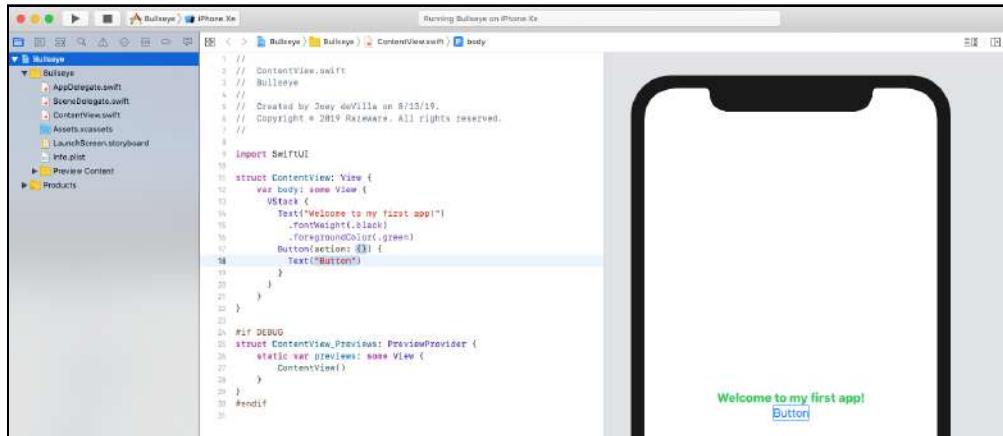
The Library window, with the Button view selected

- Click and start dragging the **Button** item from the Library list and onto the Canvas. Drag it to just below the “Welcome to my first app!” text. A blue line should appear just below the text, and a pop-up window that reads **Add Button to a new Vertical Stack along with existing Text**. should appear at the bottom of the Canvas:



Dragging a button onto the view

If you look at the Editor, you'll notice some new code has appeared in the Editor. (Once again, you might need to click the **Resume** button near the upper-right corner of the Canvas.) Xcode should look like this:



The Editor and Canvas, with the button added

Take a closer look at the code in the Editor:

```
struct ContentView : View {
    var body: some View {
        VStack {
            Text("Welcome to my first app!")
                .fontWeight(Font.Weight.black)
                .color(Color.green)
            Button(action: {}) {
                Text("Button")
            }
        }
    }
}
```

A couple of new elements have been added by dragging a button onto the Canvas. First, let's take a look at the new **Button** object:

```
Button(action: {}) {
    Text("Button")
}
```

You may have noticed a couple of things about **Button**:

- In its parentheses, there's `action: {}`, followed by braces. You'll put code that responds to the **Button** being pressed inside these parentheses.

Remember that in Swift, `:` means “is a”, and braces mark a group of lines of code, or mini-program. You might want to read `action: {}` as “action is a mini-program.”

- Button contains a Text object, and that object determines what the button says.

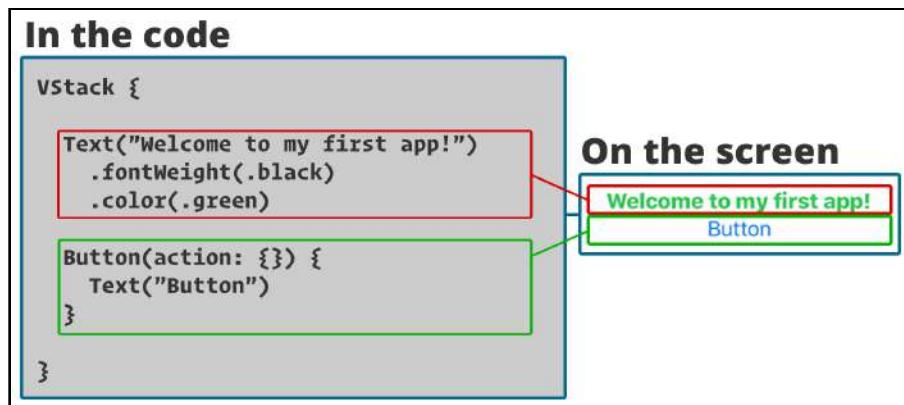
The other new element is a `VStack`, which is short for *vertical stack*, which is a view that acts as a container for many views, and it arranges them in a vertical line. Before we look more closely at the `VStack` code, let’s format it so that it’s easier to see what it does.

► Change the indentation of the lines relating to the button so that the `VStack` code looks like the code below:

```
VStack {  
    Text("Welcome to my first app!")  
        .fontWeight(Font.Weight.black)  
        .color(Color.green)  
    Button(action: {}) {  
        Text("Button")  
    }  
}
```

With the code formatted this way, it’s easy to see that the `VStack` contains two views: a `Text` and a `Button`. Since the `Text` appears first, it is positioned at the top of the `VStack`, with the `Button` just below it.

The diagram below shows the relationship between the `VStack` code and what you see in the Canvas:



VStack, in code and on the screen

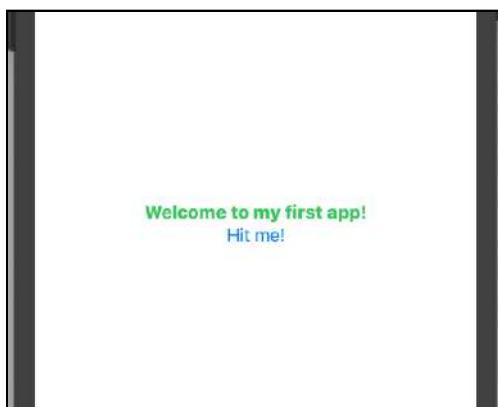
According to the to-do list, the Button should say “Hit me!” and not “Button”, so let’s fix that.

- Change the line defining the Text inside Button so that it reads like this:

```
Button(action: {}) {  
    Text("Hit me!")  
}
```

- Click the **Run** button.

You should see the following in the Simulator:



The app, with the Hit me! button added

- In the app, click the **Hit me!** button.

Nothing happens when you click the button. That’s to be expected, because you haven’t yet defined what should happen when the button is clicked.

Responding to button clicks

Let’s go back to the code that defines the button:

```
Button(action: {}) {  
    Text("Hit me!")  
}
```

As mentioned earlier, any code that should be executed when Button being is clicked should go inside the braces that follow `action:`. Let’s try some quick experimentation as a first step towards the goal of creating a pop-up when then user presses the button.

- Edit the code for the Button so that it looks like this:

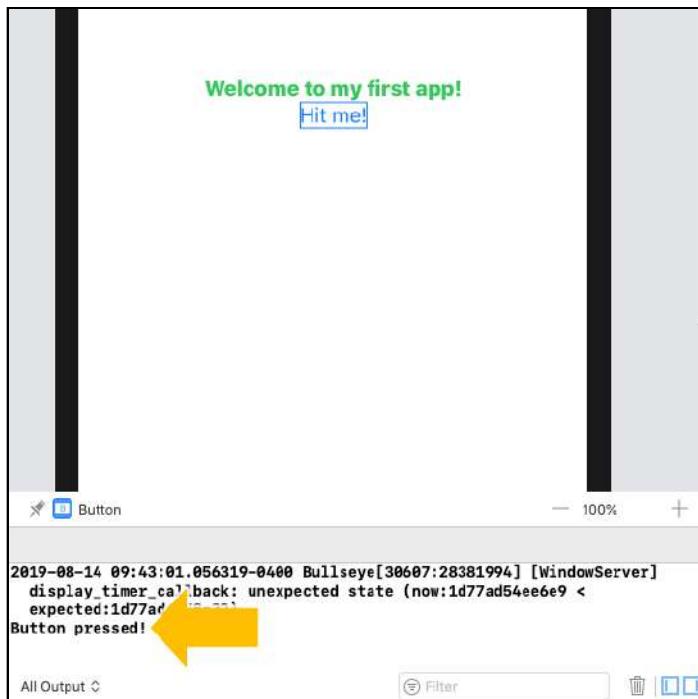
```
Button(action: {  
    print("Button pressed!")  
}) {  
    Text("Hit me!")  
}
```

You'll see what `print()` does in a moment.

Harnessing the power of `print()`

- Click the **Run** button, and when the app starts up in the Simulator, click the **Hit me!** button a couple of times.

You may initially be disappointed when clicking **Hit me!** seems to do nothing. But if you take a look at Xcode, you'll see that a pane has opened at the bottom of its window, and every time you click **Hit me!** in the Simulator, a new "Button pressed!" appears in the pane:



The debug pane, displaying 'button pressed'

`print()` is a *function*, which is a kind of method, except that it's not attached to any object. Not being attached to an object means that you can use it without having to name an object first.

Hint: In Swift, any time you see a name that starts with a lowercase letter that's immediately followed by parentheses, such `fontWeight()` or `print()` — you're probably looking at the name of a function or method. The difference is that methods are preceded by the object they belong to (for example, `Text("Hello").fontWeight(.black)`), while functions don't belong to objects and simply appear on their own (for example, `print("Hi")`).

The `print()` function takes some text and then *prints* it in Xcode's Console, which is the pane that appeared when you pressed **Hit me!** in the Simulator. The Console is a read-only text area that displays messages from the compiler and other Xcode systems, as well as messages from your own apps via `print()`.

`print()` is a *debugging tool*. This means that its purpose is to help the programmer figure out what's going on in their program, and it's often used to determine the cause of bugs (hence the name). You only see its results in Xcode, and only in apps that are run from Xcode in the Simulator or on devices that are connected to your computer. In apps that have been installed on a device and run on their own instead of Xcode, `print()` has no effect. It's there only for the benefit of you, the programmer.

As you continue programming, you'll find yourself using `print()` as an indicator to check that specific pieces of code are being executed, or to check on some value that your program has stored. In the code you just wrote, you're using `print()` as a signal that control of the program had reached a specific point: the `action: code` for the `Button`.

The fact clicking **Hit me!** in the app causes `print()` to print "Button pressed!" in the Console is a good sign. It means that you can respond to a button being clicked. Congratulations — you've just written some interactive code!

You're not done yet. Since `print()` is a debugging tool, users *never* see their results. From their point of view, clicking **Hit me!** still does nothing. You still have to make the button provide a response that the user can see. In order to do that, we need to go over the concept of *state*.

State and SwiftUI

A key part of programming SwiftUI is *state*. Rather than start with the computer science definition of state, let's go with something that might be a little more familiar: the dashboard of a car.



The dashboard of a car

The gauges and odometers are usually the most noticeable parts of a dashboard. They show the car's current speed, fuel level, engine temperature and distance traveled, each of which is some kind of numerical quantity.

Dashboards also have warning lights, such as the “check engine” light, the low oil pressure warning light, the “it’s time to take the car to the shop for overpriced regular maintenance” light and the “someone’s not wearing their seat belt and will be very sorry if there’s an accident” light. Each of these lights is either on, indicating that there’s a problem that needs the driver’s attention, or off. This “on/off”, “yes/no” information can be described as *binary*.

The information on a car’s dashboard — speed, fuel level, whether or not someone in the car likes to live dangerously without a seat belt and so on — taken all together, is that car’s *state*.

The driver’s actions can change the car’s state, and the new state is immediately shown in the dashboard. For example, if the driver presses on the accelerator pedal, the car’s speed increases, which in turn causes the speedometer to display the car’s new speed. If the driver then presses on the brake pedal, the car slows down, and the speedometer automatically and immediately shows the new, slower speed.

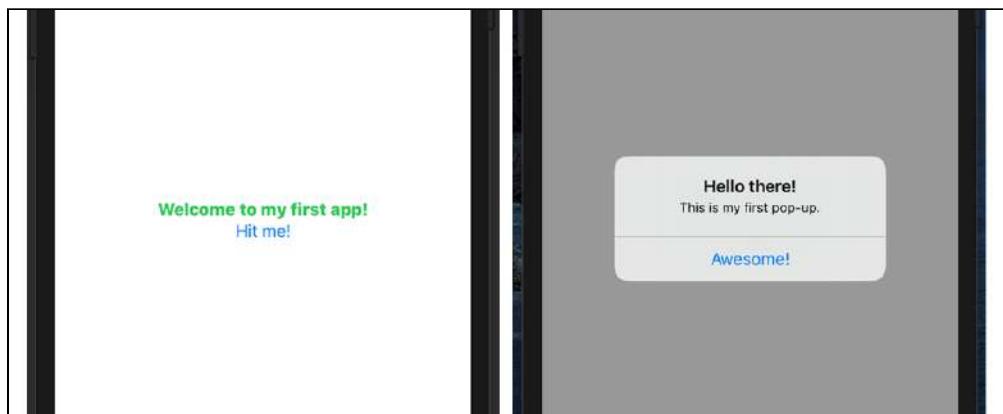
Internal circumstances can also change the car’s state, and once again, this new state is shown in the dashboard instantly. As the car uses up fuel, the fuel gauge moves away from F and towards E. When the driver fills the tank, the fuel gauge immediately goes back to F.

Another example is the “maintenance” light. When a pre-determined amount of time passes or the car has traveled a pre-determined distance since it was last brought in for maintenance, the car’s state changes from not needing maintenance to needing it, and the “maintenance” light turns on. Once the car has been brought to the dealership or a mechanic for maintenance, the car’s state changes back to not needing maintenance and the light turns off.

There’s a term for every possible combination of every possible value of those things that make up the car’s state: the *state space*. With all the possible combinations of things that make up the car’s state — speed, fuel level, engine temperature, distance traveled, and so on — a car has a really large state space.

The one-button app’s state space

Unlike our car example, the one-button app you’re building has a much smaller state space. In case you’ve forgotten, let’s take a second look at what the app will look like by the end of this chapter:

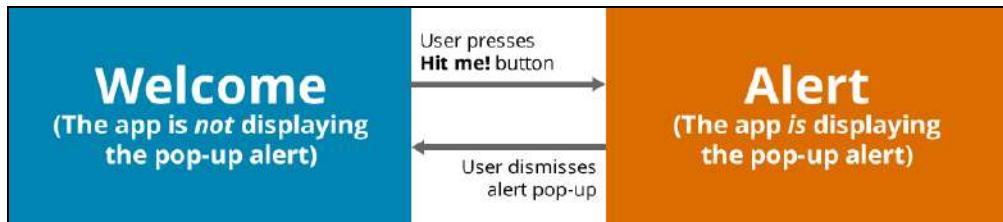


What the app will look like by the end of this chapter

It turns out that the app has only two states:

1. **Welcome:** Simply displaying the “Welcome to my first app!” screen that you’ve already seen. This is what the user should see when they haven’t pressed the **Hit me!** button.
2. **Alert:** Displaying the alert pop-up. This is what the user should see when they press the button.

The app's state space is so small that it's possible to draw it out in a *state diagram*:



State diagram for the one-button app

The rectangles represent the app's two states. The arrows are *transitions*, which are the ways to get from one state to another. Written beside each transition arrow is the reason for the change in state, formally known as a *transition condition*. The app has two transitions:

1. From **Welcome** to **Alert**. This transition happens when the user presses **Hit me!**.
2. From **Alert** to **Welcome**. This transition happens when the user dismisses the alert pop-up.

SwiftUI and state space

Coding a user interface in SwiftUI is similar to drawing a state diagram. Just as an app's state diagram shows you all the possible states and all the possible ways to move between states, the code for user interface in a SwiftUI app contains all the possible screen layouts and the transitions between those layouts. It's the state diagram for the user interface, in code form.

Let's review the code for the user interface as it is right now:

```
struct ContentView : View {
    var body: some View {
        VStack {
            Text("Welcome to my first app!")
                .fontWeight(.black)
                .color(.green)
            Button(action: {
                print("Button pressed!")
            }) {
                Text("Hit me!")
            }
        }
    }
}
```

Remember, the `body` property of `ContentView` defines the layout of the screen. At the moment, `body` contains a `VStack`, which arranges a `Text` object and a `Button` object in a vertical stack, in that order. This covers the **Welcome** state. We're missing the layout for the **Alert** state.

We'll define that layout soon, but there's something we need to take care of first: the app's state.

Representing state in the app with a variable

Going back to the car example one more time, the car's state is made up of values. Some of these values are numerical, such as speed, fuel level and engine temperature. Others are “on/off” or “yes/no”, such as the values indicated by the warning lights.

A program's state is also made up of values, and these values are stored in variables. Variables allows the app to remember things. Think of them as temporary storage containers, each one storing a single piece of data. A variable — or more often, many variables — is the perfect place to store an application's state.

You've already seen a variable in action: `body`, which contains everything that your app's single screen displays — the “Welcome to my first app” text, and the **Hit me!** button. Now you're going to create a variable to represent the app's state.

The variable that represents the app's state needs to store two possible values:

1. A value representing the state where the app *is not* displaying the alert pop-up.
2. A value representing the state where the app *is* displaying the alert pop-up.

This sounds like an “on/off”, “yes/no” value. In Swift and many other programming languages, such values are expressed as `true` and `false` and are called *Boolean values*, after English mathematician and logician George Boole.

Let's create a variable that holds a Boolean value to keep track of whether or not the alert pop-up should be visible. We'll give it the name `alertViewIsVisible`. When the app starts, the alert pop-up should *not* be displayed, which means that `alertViewIsVisible`'s initial value should be `false`.

► Add a line to the code for `ContentView` so that it looks like the following:

```
struct ContentView : View {  
    @State var alertisVisible: Bool = false  
  
    var body: some View {
```

```
    VStack {
        Text("Welcome to my first app!")
            .fontWeight(.black)
            .color(.green)
        Button(action: {
            print("Button pressed!")
        }) {
            Text("Hit me!")
        }
    }
}
```

Let's look at the newly-added line, `@State var alertisVisible: Bool = false`.

- You've already seen the keyword `var`, which means "This is a variable named..." It's followed by the name of the variable `alertViewable`.
- The variable `alertViewable` is followed by `:`, which means "is a", and `Bool`, which is Swift for "Boolean". This means that `alertViewable` can contain one of two possible values, `true` or `false`.
- Finally, the `= false` part means that `alertViewable` is initially set to the value `false`. Being a variable, you can expect this value to change at some point.

That unfamiliar new keyword, `@State`, needs a little more explaining. You've probably figured out that it marks `alertViewable` as a variable that stores information about the app's state. It also tells Swift to watch for any changes to the contents of this variable and be ready to take action when that happens.

When the app starts, `alertViewable`'s value is `false`. It will stay that way forever unless we add code to change its value to `true`. We want that to happen when the user presses **Hit me!**.

► Change the code for `ContentView` so that it reads as follows:

```
struct ContentView : View {
    @State var alertisVisible: Bool = false

    var body: some View {
        VStack {
            Text("Welcome to my first app!")
                .fontWeight(.black)
                .color(.green)
            Button(action: {
                print("Button pressed!")
                self.alertisVisible = true
            }) {
                Text("Hit me!")
            }
        }
    }
}
```

```
        }
    }
}
```

You've just added a line that sets `alertVisible` to `true`, and you've done it inside `Button`'s block of code marked `action:`, which gets executed when the user presses the button.

You might be wondering about the keyword `self`, which got tacked on to the beginning of `alertVisible.alertVisible`. `alertVisible` is a feature of the `ContentView` object, and in code *outside* `ContentView`, you access it with the code `ContentView.alertVisible`. However, you're *inside* `ContentView`, so you access it with the code `self.alertVisible` instead.

Defining the layout for the other state

It's worth repeating: In SwiftUI, you define the layout for all possible states. The layout for the **Welcome** state is already in the code; it's now time to define the layout for the other state — the **Alert** state, which is the app's state when the variable `alertVisible` contains the value `true`.

The state of the app is now updated whenever the user presses the button. It's time to write code to respond to that change in state.

The alert pop-up should appear when the user clicks the button. Fortunately, one of the methods that comes built into the `Button` object is called `alert()`, and it makes an alert pop-up appear if a given state variable's value is `true`.

Rather than *tell* you how it works, it's simpler (and more fun!) to *show* you:

- Change the code for `ContentView` so that it reads as follows:

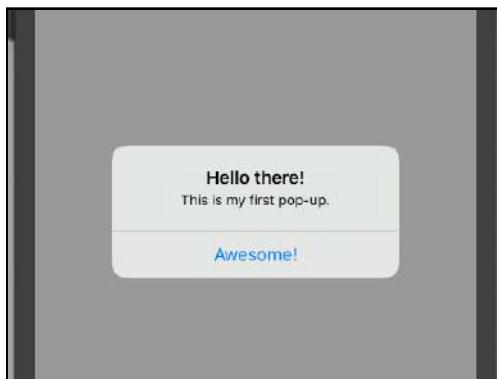
```
struct ContentView : View {
    @State var alertVisible: Bool = false

    var body: some View {
        VStack {
            Text("Welcome to my first app!")
                .fontWeight(.black)
                .color(.green)
            Button(action: {
                print("Button pressed!")
                self.alertVisible = true
            }) {
                Text("Hit me!")
            }
        }
    }
}
```

```
        }
        .alert(isPresented: self.$alertIsVisible) {
            Alert(title: Text("Hello there!"),
                  message: Text("This is my first pop-up."),
                  dismissButton: .default(Text("Awesome!")))
        }
    }
}
```

- Press the **Run** button, and when the app starts up in the Simulator, press the **Hit me!** button.

You should see the following:



The alert pop-up

Let's take a closer look at the code you just added:

```
.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text("Hello there!"),
          message: Text("This is my first pop-up."),
          dismissButton: .default(Text("Awesome!")))
}
```

`alert()` is a method on `Button` that presents an alert to the user. It requires two *arguments*, which is just computer science talk for “things that you provide to a method or function when you call it so that it can do its job.” Arguments can be either data or instructions (or in other words, code). In this case, you’re providing two pieces of data:

1. **A binding, or two-way connection to a Boolean state variable.** In this case, that variable is `alertIsVisible`, and the two-way connection means that `alertIsVisible`’s value and the alert pop-up are tied together. Changing `alertIsVisible` to `true` causes the alert pop-up to appear, and the user

dimissing the pop-up causes `alertIsVisible`'s value to be set to `false`. The `$` in `$alertIsVisible` tells Swift that this is a *two-way binding*: changes to `alertIsVisible` affect the alert pop-up, and changes to the alert pop-up — affect `alertIsVisible`.

2. **An Alert object.** This defines what the alert pop-up should contain.

The `Alert` object needs to be provided wth three things:

1. **title:** This is the bold text at the top of the alert pop-up. You typically want this to be short — no more than a handful of words. The argument is a `Text` object, which you initialize with another argument: “Hello there!”
2. **message:** This is the text below the title. You can make this a little longer than the title, but you shouldn’t make it longer than a sentence or two. As with the `title`, the argument is also a `Text` object that you initialize with “This is my first pop-up.” (Feel free to change any of these.)
3. **dismissButton:** This is the button at the bottom of pop-up. It dismisses the pop-up when the user presses it. The argument a method that creates a button with the text “Awesome!”

Congratulations, you’ve just written your first iOS app! What you just did may have seemed like gibberish to you, but that shouldn’t matter. We’ll take it one small step at a time.

You can strike off the first two items from the to-do list already: Putting a button on the screen and showing an alert when the user taps the button.

Take a little break, let it all sink in and come back when you’re ready for more! You’re only just getting started...

Note: Just in case you get stuck, the complete Xcode projects have been included in the files that come with this book. They’re snapshots of the project as they would appear at the beginning and end of each chapter. That way, you can compare your version of the app to the book’s, or — if you really make a mess of things — continue from a known working version.

You can find the project files for each chapter in the corresponding folder.

Dealing with error messages

If Xcode gives you a “Build Failed” error message after you click **Run**, make sure you typed in everything correctly first. Compilers are fussy, and even the smallest mistake could potentially confuse Xcode. It can be quite overwhelming at first to make sense of the error messages that Xcode spits out. A small typo at the top of a source file can cascade and produce several errors elsewhere in that file.

One common mistake is differences in capitalization. The Swift programming language is case-sensitive, which means it sees `Alert` and `alert` as two different names. Xcode complains about this with a “`<something>` undeclared” or “Use of unresolved identifier” error. When Xcode says things like “Parse Issue” or “Expected `<something>`” then you probably forgot a brace `}` or parenthesis `)` somewhere. Not matching up opening and closing brackets is a common error.

Tip: In Xcode, there are multiple ways to find matching braces to see if they line up. If you move the editing cursor past a closing brace, Xcode will highlight the corresponding opening brace, or vice versa. You could also hold down the `⌘` key and move your mouse cursor over a line with a brace and Xcode will highlight the full block from the opening curly brace to the closing brace (or vice versa) — nifty!

You can see the block, here:



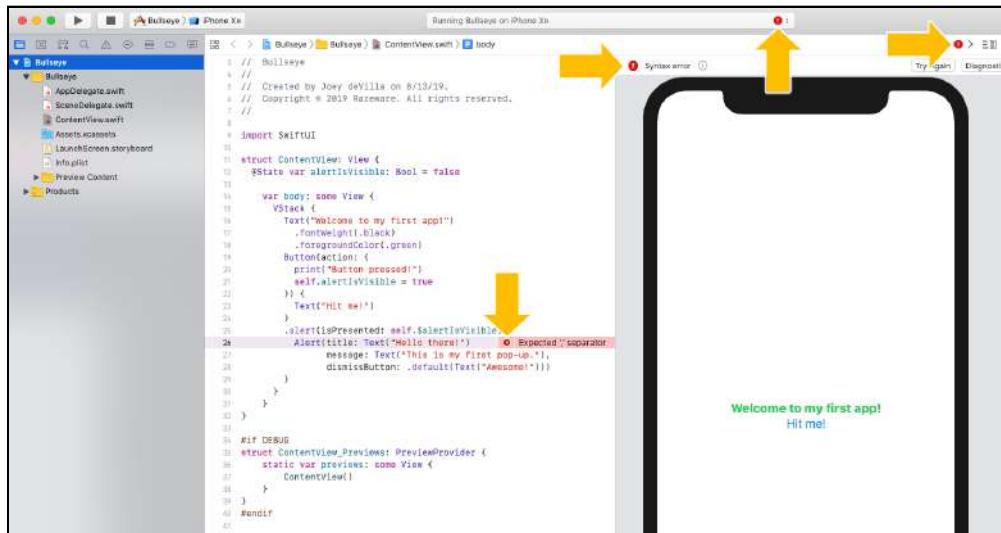
A screenshot of Xcode showing a Swift code block. The code defines a variable `body` as a `someView` type, containing a `VStack` block. Inside the `VStack` block, there is a `Text` element with the text "Hello me to my first app!" and some styling. Below the `VStack` is a `Button` block with an action that prints "Button pressed!" and sets `self.alertVisible` to `true`. The code then continues with another `Text` element and an `.alert` modifier. Two large yellow arrows point from the left margin towards the closing brace of the `body` variable, highlighting the entire scope of the variable definition.

```
var body: someView {
    VStack {
        Text("Hello me to my first app!")
            .fontWeight(.black)
            .foregroundColor(.green)
        Button(action: {
            print("Button pressed!")
            self.alertVisible = true
        }) {
            Text("Hit me!")
        }
        .alert(isPresented: self.$alertVisible) {
            Alert(title: Text("Hello there!"),
                  message: Text("This is my first pop-up."),
                  dismissButton: .default(Text("Awesome!")))
        }
    }
}
```

Xcode shows you the complete block for braces

Tiny details are very important when you’re programming. Even one single misplaced character can prevent the Swift compiler from building your app.

Fortunately, such mistakes are easy to find:

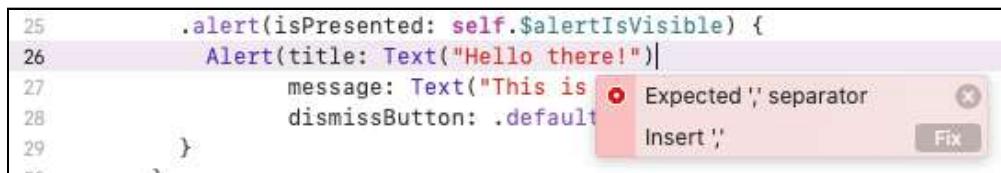


Xcode makes sure you can't miss errors

When Xcode detects an error, it switches the pane on the left from the Project navigator, to the **Issue navigator**, which shows all the errors and warnings that Xcode has found. (You can go back to the project navigator using the small icons along the top.)

The screenshot above shows an intentional error: a missing comma between the title and message parameters of the `Alert()` view that should appear whenever `alertIsVisible` is true.

Click on the error message in the Issue navigator and Xcode takes you to the line in the source code with the error. It's often not clear what went wrong when your build fails. For certain kinds of errors, Xcode lends a helping hand, suggests a fix and even offers to make the fix for you:



Fix-it suggests a solution to the problem

In this case, clicking the **Fix** button has Xcode follow its suggestion and insert the missing comma...

```
25     .alert(isPresented: self.$alertIsVisible) {  
26         Alert(title: Text("Hello there!"),  
27             message: Text("This is my first pop-up."),  
28             dismissButton: .default(Text("Awesome!")))  
29     }
```

Fix-it implements its solution

...and with the fix made, all the error indicators vanish. You have a working app again!

Errors and warnings

Xcode makes a distinction between errors (red) and warnings (yellow). Errors are fatal. If you get one, you cannot run the app until the error is fixed.

Warnings are informative. Xcode just says, “You probably didn’t mean to do this, but go ahead anyway.”

In the previous screenshot showing all the error locations via arrows, you’ll notice that there is a warning (a yellow triangle) in the Issue navigator. We’ll discuss this particular warning and how to fix it later on.

Generally though, it is best to treat all warnings as if they were errors. Fix the warning before you continue and only run your app when there are zero errors and zero warnings. That doesn’t guarantee the app won’t have any bugs, but at least they won’t be silly ones!

The anatomy of your app

Let's finish this chapter by looking at what goes on behind the scenes of your app.

You've already been introduced to the concept of *objects*, which are the building blocks of the app. The key objects in the app so far are:

- **ContentView**: An object representing the app's main screen.

You've also been introduced to the idea of *state*, which you can think of as a snapshot of the app's current circumstances. Right now, the app has two possible states, which are represented by the contents of the `alertVisible` variable, which holds a Boolean value (that is, it contains either `true` or `false`):

1. The **Welcome** state, where the user sees the main screen, with the "Welcome to my first app!" message. This is the app's state when `alertVisible` is set to `false`.
2. The **Alert** state, where the main screen is obscured by the alert pop-up. This is the app's state when `alertVisible` is set to `true`.

In SwiftUI, apps work by having its objects and state affect and be affected by each other. An object can react to some external stimulus — which could come from the user, the operating system, another object, or a change in state — which then causes the object to change the app's state.

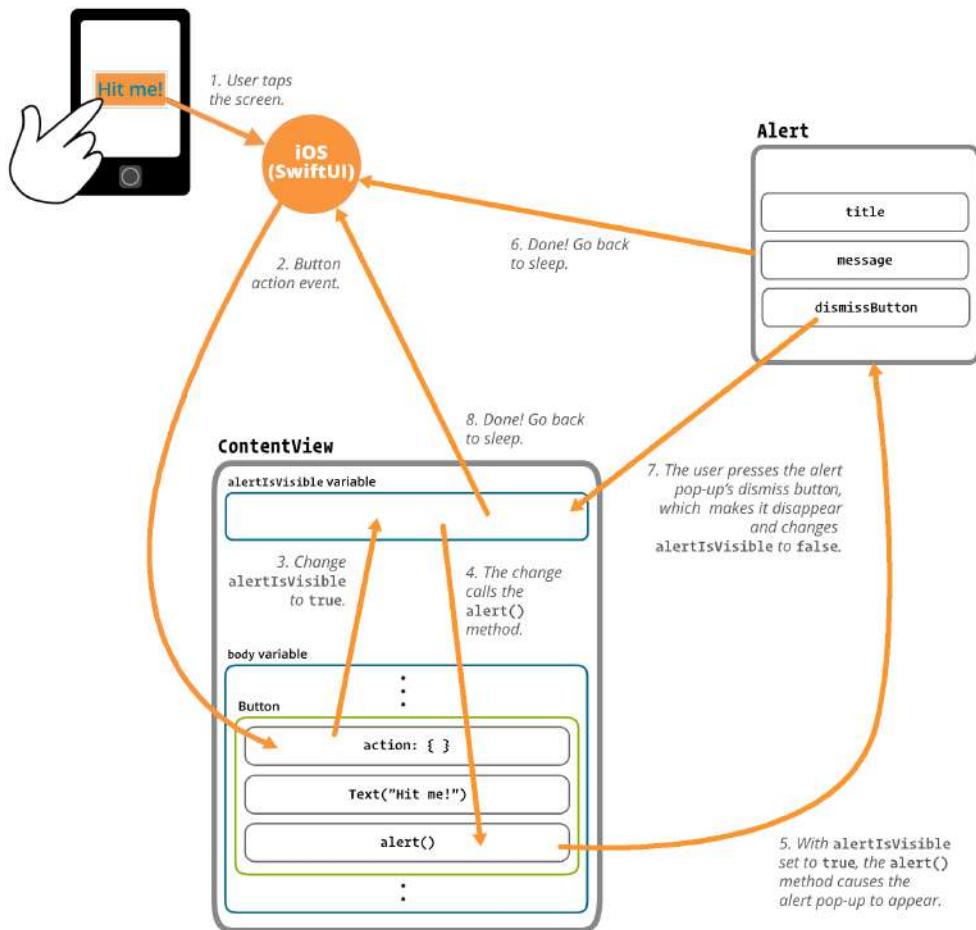
Changing the state can affect objects, which perform some action or change some data in response, which in turn can affect the state.

As strange as it may sound, an app spends most of its time doing...absolutely nothing. It just sits there waiting for something — an *event* — to happen. We say that iOS apps are *event-driven*, which means that apps' objects constantly listen for certain events to occur, and when they do, the objects respond as programmed.

When the user taps the screen, the app springs into action for a few milliseconds and then it goes back to sleep until the next event arrives.

Your part in this scheme is that you're writing the source code that ties SwiftUI objects and state together, so that objects respond to changes in state, and make the appropriate changes to the state, which in turn causes the objects to respond, and so on.

The diagram below shows how the app that you've just completed works “under the hood”:



The anatomy of your app

When the user presses the **Hit me!** button on the screen, its corresponding **Button** object executes the code in its `action: {}` method, which sets the contents of the `alertIsVisible` variable to `true`. `alertIsVisible` is a state variable, which means that it can affect objects in the app.

The **Button** object also calls its `alert()` method, which has a two-way binding to `alertIsVisible`. This means that this method is called every time `alertIsVisible` changes. The `alert()` method also defines an **Alert** view that should appear if `alertIsVisible` is set to `true`, which is the case when the button is pressed.

With the alert pop-up now displayed onscreen, the app returns back to what it does most of the time: waiting for the next user interaction. Since the alert is *modal* — that's user interface jargon for “completely blocking everything else on the screen until the user deals with it” — the only possible user interaction within the app is to dismiss the alert by pressing its button.

When the user presses the **Awesome!** button on the alert to dismiss it, the two-way connection between the visibility of the alert and `alertViewIsVisible` causes `alertViewIsVisible` to be set to `false`. The user is now back on the main screen, and the app goes back to waiting for the next user interaction.

This app will spend more than 99% of its time waiting for input events — in this case, button presses — from the user. It will spend a very small amount of time, measured in milliseconds, on responding to those events.

While this app responds only to touch events from the user, the user can provide other events as well, such as shaking the device or speaking. The operating system can also provide events that notify apps when things happen, such as the user receiving an incoming phone call, or when the app has received incoming data from the internet, and so on. Everything apps do is triggered by events.

You can find the project files for the app up to this point under **02 - The One-Button App** in the **Source Code** folder.

Chapter 3: Building User Interfaces

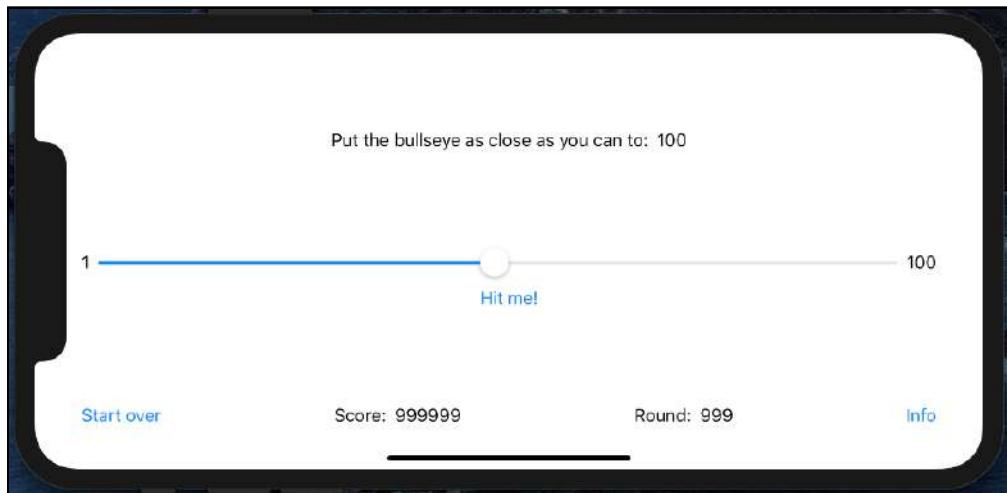
Joey deVilla

Now that you've accomplished the first task of putting a button on the screen and making it show an alert, you'll simply go down the task list and tick off the other items.

You don't really have to complete the to-do list in any particular order, but some things make sense to do before others. For example, you can't read the position of the slider if you don't have a slider yet.

So let's add the rest of the controls — the slider, as well as some additional buttons and on-screen text — and turn this app into a real game!

When you've finished this chapter, the app will look like this:



The game screen with standard SwiftUI controls



Hey, wait a minute... that doesn't look nearly as pretty as the game I promised you! The difference is that these are the standard controls. This is what they look like straight out of the box.

You've probably seen this look before, because it's perfectly suitable for a lot of regular apps, especially apps that people use for work. However, the default look is a little boring for a game. That's why you'll put some special sauce on top later, to spiff things up.

In this chapter, you'll cover the following:

- **Portrait vs. landscape:** Switch your app to landscape mode.
- **Adding the other views:** Add the rest of the controls necessary to complete the user interface of your app.
- **Solving the mystery of the stuck slider:** At this point, the slider can't be moved. Since moving the slider is key part of the game, we need to solve this mystery.
- **Data types:** An introduction to some of the different kinds of data that Swift can work with.
- **Making the slider less annoyingly precise:** We don't need the slider to report its position with six-decimal precision, but to the nearest whole number.
- **Key points:** A quick review of what you learned in this chapter.

Portrait vs. landscape

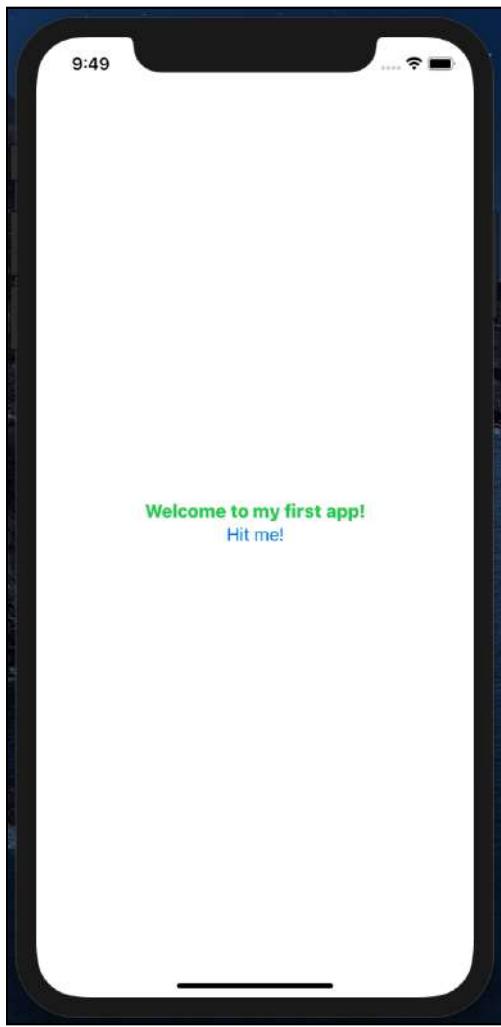
Notice that in the previous screenshot, the **aspect ratio** — the ratio of width to height — of the app has changed. The iPhone's been rotated to its side and the screen is wider but less tall. This is called **landscape** orientation.

Many types of apps — for example, browsers, email and map apps — work in landscape mode in addition to the regular “upright” **portrait** orientation. Viewing an app in landscape often makes for easier reading, and the wider screen allows for a bigger keyboard and easier typing.

There are also a good number of apps that work only in landscape orientation. Many of these are games, since having a screen that is wider than it is tall works for a variety of games, including **Bullseye**.

Right now, the app works in both portrait and landscape orientations. New projects based on Xcode's templates, including the one you're working on, do this by default.

- Build and run the app. If you've been following the steps in this book up to this point, it should look like this in the simulator:

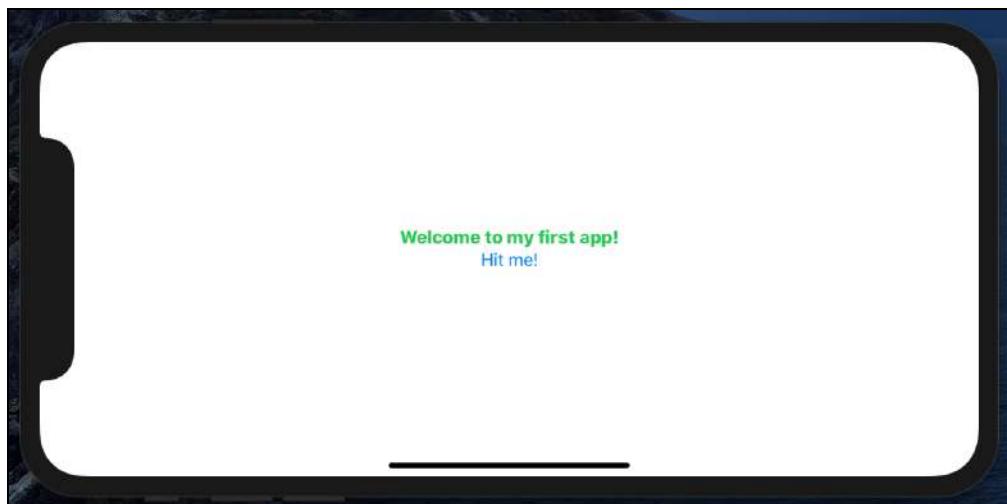


The app so far, in portrait orientation

The simulator defaults to portrait orientation, right side up, since this is the usual way people hold their phones. You can simulate the action of turning your phone to its side — or even upside down — in a couple of different ways:

- You can change the simulator's orientation by opening its **Hardware** menu and using the **Rotate Left** and **Rotate Right** options in that menu to rotate the simulator 90 degrees left or right.

- You can also use keyboard shortcuts. Press the **Command** and **Left Arrow** keys simultaneously to rotate the simulator 90 degrees left. Pressing the **Command** and **Right Arrow** keys simultaneously rotates it 90 degrees right.
 - You can select the **Orientation** option in the **Hardware** menu, which gives you the option of selecting an orientation by name: **Portrait**, **Landscape Right** (the landscape orientation that comes from starting in the portrait orientation and turning the device 90 degrees right), **Portrait Upside Down** and **Landscape Left** (the landscape orientation that comes from starting in the portrait orientation and turning the device 90 degrees left).
- While in the simulator, press the **Command** and **Left Arrow** keys simultaneously. You should see this:



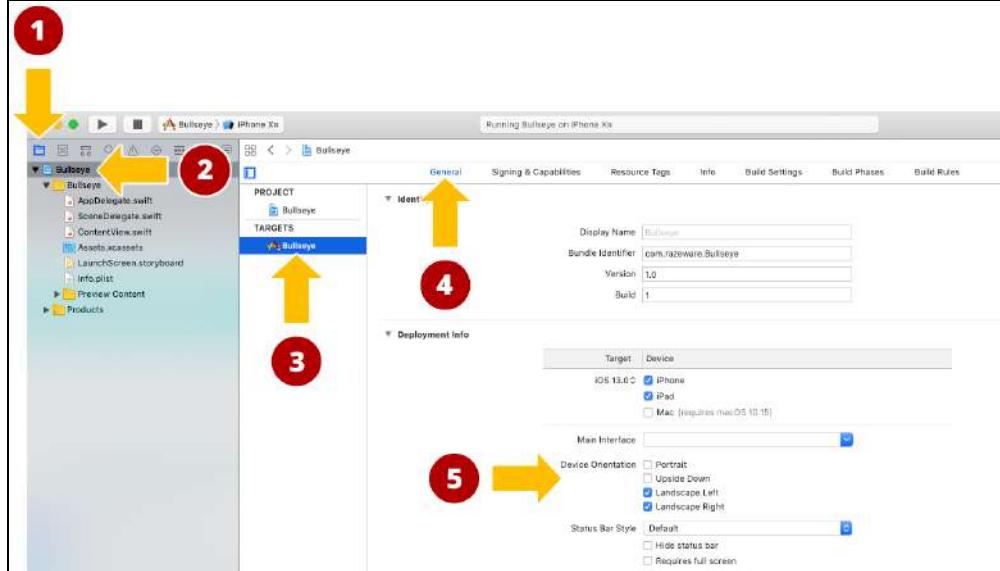
The app so far, in landscape orientation

One of the advantages that SwiftUI has over the old way of building iOS user interfaces — UIKit — is that it adjusts automatically to changes in orientation without requiring much work from the programmer. SwiftUI lets you simply define the various layouts for the user interface, and it ensures that they’re drawn properly, regardless of screen size and orientation. Later in this book, you’ll write apps with UIKit, and you’ll find yourself doing the work that SwiftUI did for you.

Converting the app to landscape

The **Bullseye** game works best in landscape orientation, since landscape allows for the widest slider possible. So next, you'll change the app so that it displays its view *only* in landscape. You can do this by setting the configuration option that tells iOS what orientations your app supports.

- In the Navigator section of Xcode, which is the leftmost column in the Xcode window, make sure that you've selected the Project navigator, whose icon looks like a file folder. Click the blue **Bullseye** project icon at the top of the Project navigator's list. The Editor and Canvas will disappear and panes that let you change your project's configuration will replace them.
- Make sure that you've selected the **General** tab:



The settings for the project

In the **Deployment Info** section, there are a number of checkboxes in an area marked **Device Orientation**.

- Make sure that you've unchecked **Portrait** and **Upside Down**, and that you've checked **Landscape Left** and **Landscape Right**.

- Build and run the app. You'll see that no matter which way you rotate the simulator, the app always stays in landscape orientation:

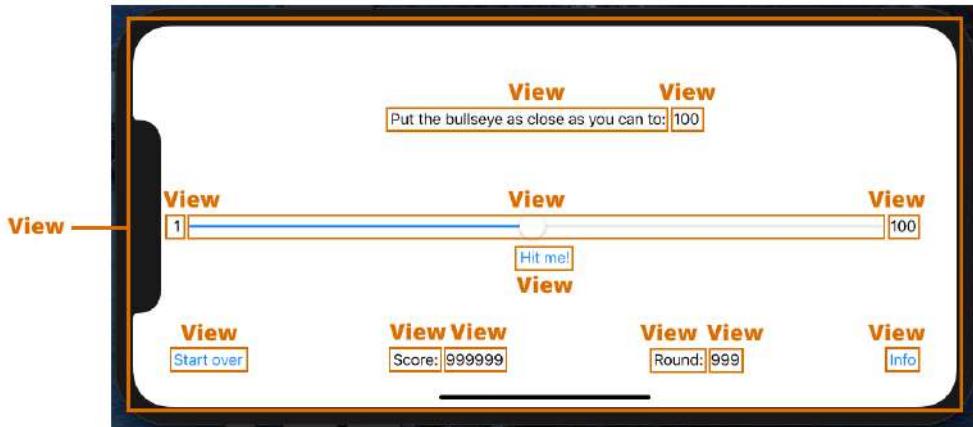


The app, set to landscape only, with the simulator in portrait orientation

Adding the other views

You're going to see the word "view" a lot in this book, so take a moment to quickly go over what "view" means. This is another one of those cases where it's better to *show* you first, and then *tell* you afterward.

Once again, here's what the **Bullseye** screen will look like at the end of the chapter, this time with all the *apparent* views highlighted and labeled:



The game screen, with all the views highlighted

Some of the views are invisible; you'll learn more about them soon.

A view is *anything* that gets drawn on the screen. In the screenshot above, it seems that everything is a view: The text items, the buttons and the slider are all views. In fact, every user interface control is a view.

Some views can act as containers for other views. The biggest view in the screenshot is one of these: It's the view representing the screen, and it contains all the other views on the screen: The text items, the buttons and the slider.

Different types of views

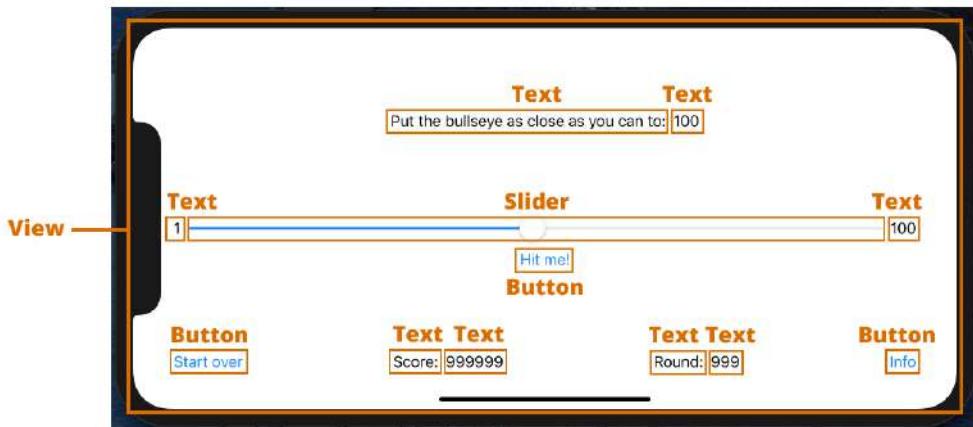
There are different types of views. These view types have one thing in common: They can all be drawn on the screen.

What makes each type different is a combination of *what they look like* and *what they do*. So far, you've worked with a few of them:

- **Text:** A view that displays one or more lines of read-only text. The "Welcome to my first app!" message in the app you made in Chapter 2, "Getting Started with SwiftUI" is a **Text** view.

- **Button:** A view that performs an action when triggered. In iOS, a user triggers a button when they complete a button press by releasing the button after pressing down on it. “Hit me!” in the app you made in Chapter 2, “Getting Started with SwiftUI” is a **Button** view.
- **VStack:** A view that acts as a container for other views and arranges them into a vertical stack. You used this to arrange the screen so that “Welcome to my first app!” is above “Hit me!”. Unlike the **Text** and **Button** views, the **VStack** view is invisible.
- **View:** A view that represents the entire screen and acts as a container for all the other views on the screen. I wouldn’t call this view “invisible”, but it’s something that the user generally doesn’t notice.

Take a look at those **Bullseye** screen views again, but with the specific types of views called out this time:

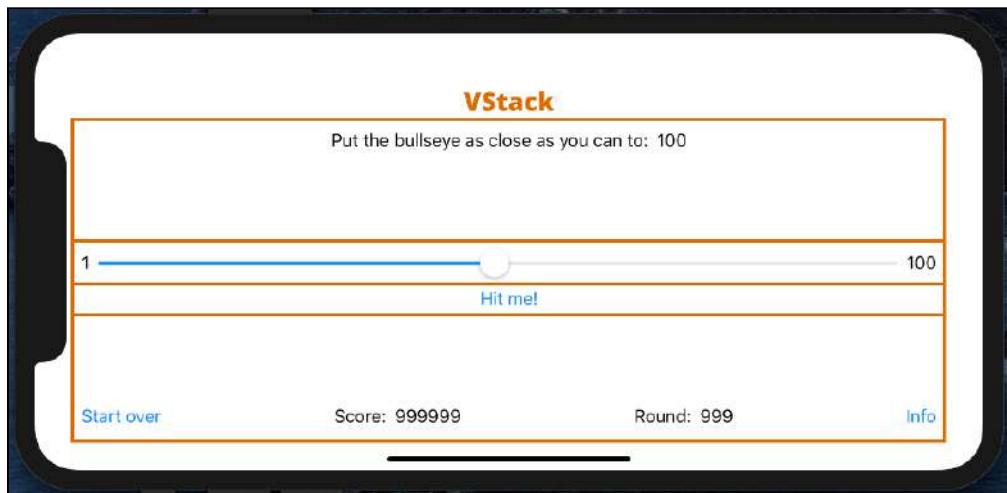


The different kinds of views in the game screen

You may have noticed a control you haven’t worked with before: The **Slider**. This control lets a user enter a number by sliding a control, which is called a **thumb**, along a straight track where one end represents a minimum value and the other end represents a maximum value.

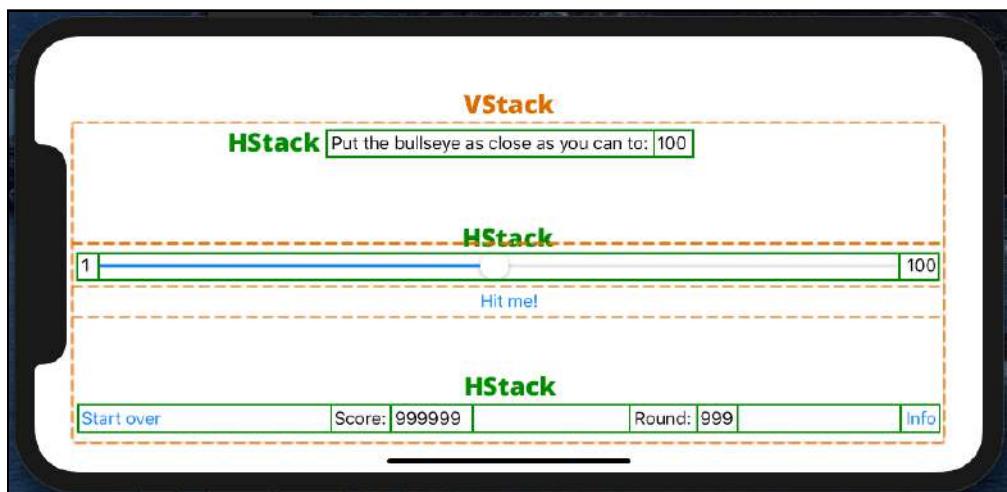
Note: In most apps, you wouldn’t make the user enter a precise number value using a slider. However, for a game like **Bullseye**, the slider makes the game challenging. After all, we don’t want to make it too easy for the player!

As I mentioned earlier, some of the views that will go on the game screen are invisible. One of them is a **VStack**, which you've already used. You'll continue to use it to arrange the views into rows. The screenshot below shows the rows that you'll create using the **VStack**:



The VStack in the game screen

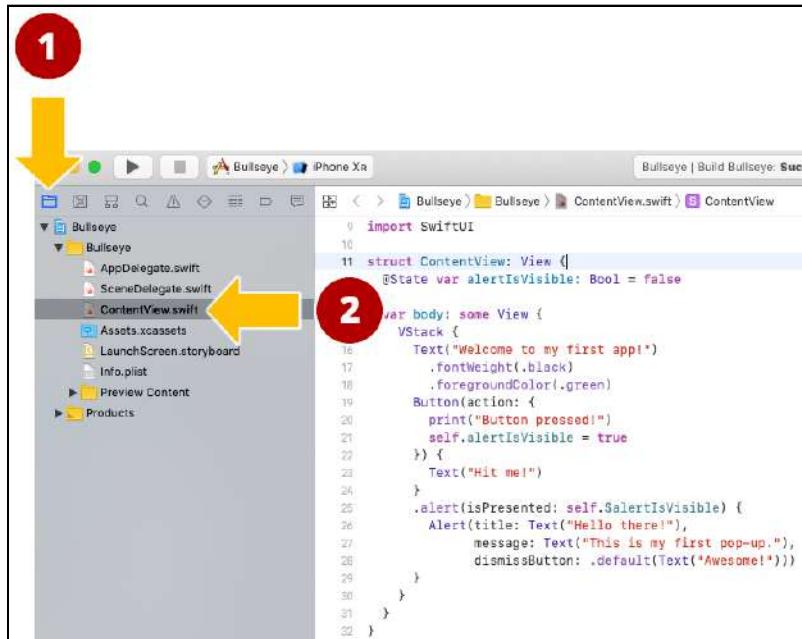
You'll also need to arrange some rows side by side. To do this, you'll use a new control, **HStack**, which acts as a container for other views and arranges them into a horizontal stack (hence the name). You'll use three **HStack** views, and you'll put each one inside a **VStack** cell, as shown below:



The HStacks in the game screen

Reviewing what you've built so far

Let's look at the code for the app as it is right now. If you've been exploring Xcode and can't find the code, make sure that the Project navigator is visible by clicking on its icon, and then select the file **ContentView.swift**. This is the file that contains the code that defines the game's screen:



Getting back to the code

Here's the part of the code that you've been working with to define the game screen:

```
struct ContentView : View {
    @State var alertIsVisible = false

    var body: some View {
        VStack {
            Text("Welcome to my first app!")
                .fontWeight(.black)
                .color(.green)
            Button(action: {
                print("Button pressed!")
                self.alertIsVisible = true
            }) {
                Text("Hit me!")
            }
        }
    }
}
```

```
        Alert(title: Text("Hello there!"),
              message: Text("This is my first pop-up."),
              dismissButton: .default(Text("Awesome!")))
    }
}
```

We've gone over this code before, but here's a quick review of how this all works:

ContentView is the view representing the screen. The first line of the code begins with `struct ContentView : View`, and it tells you that `ContentView` is a **View**. Remember, any time you see a `:` character in Swift, you should read it as "is a".

`ContentView` is an object, and objects can have **properties**, which are things that an object *knows* and **methods**, which are things that an object *does*. Right now, `ContentView` doesn't have any methods, but it does have two properties. Each of these properties is a `var`, which means "variable":

1. **alertIsVisible**: This property is `true` if the app is currently displaying the alert pop-up, and `false` otherwise. It's `false` by default, but changes to `true` when the user presses the **Hit me!** button, and then changes back to `false` when the user dismisses the alert pop-up. `@State` marks it as a variable that Swift should watch, and tells Swift that it should be ready to take action if its contents change.
2. **body**: This property defines the content, layout and behavior of the contents of `ContentView`. Right now, `ContentView` contains a **VStack** of two views: The "Welcome to my first app!" **Text** view and the **Hit me! Button** view.

Notice that the definition of `body` starts with the line `body: some View`. You should read this as "body is a `some View`". The grammar of that sentence is a little strange, but it means that the `body` property can hold either a plain **View** or some other type of **View**, such as **Text**, **Button** or in this case, a **VStack**.

The definition of `body` implies that it can hold only one view at a time. That would normally make for very simple, very boring apps except for the fact that there are some views that can hold other views. **VStack**, **HStack** and even **View** are examples of these. For **Bullseye**, we'll fill `body` with a **VStack**, and fill that **VStack** with the other views that make up the game.

Formatting the code to be a little more readable

In order to make the code easier to work with, you're next going to space it out and add some comments. That will make it easier to add the code for each section of the user interface in the right spots.

► Edit the code in **ContentView.swift** so that it looks like the code shown below. You won't be deleting anything or changing any existing lines. You'll be simply be adding lines — some of which are blank, and some of which begin with the `//` characters:

```
import SwiftUI

struct ContentView: View {

    // Properties
    // =====

    // User interface views
    @State var alertIsVisible: Bool = false

    // User interface content and layout
    var body: some View {
        VStack {

            // Target row
            Text("Welcome to my first app!")
                .fontWeight(.black)
                .foregroundColor(.green)

            // Slider row
            // TODO: Add views for the slider row here.

            // Button row
            Button(action: {
                print("Button pressed!")
                self.alertIsVisible = true
            }) {
                Text("Hit me!")
            }
            .alert(isPresented: self.$alertIsVisible) {
                Alert(title: Text("Hello there!"),
                      message: Text("This is my first pop-up."),
                      dismissButton: .default(Text("Awesome!")))
            }

            // Score row
            // TODO: Add views for the score, rounds, and start and
        }
    }
}
```



```
info buttons here.  
}  
  
    // Methods  
    // =====  
}  
  
// Preview  
// =====  
  
#if DEBUG  
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
    }  
}  
#endif
```

- Run the app. You shouldn't notice any changes.

The changes you made don't affect the way the program works, and as a result, they won't make a difference to the user. They *will* make a difference to *you*, because they affect the way the code *reads*. Even with relatively simple apps like **Bullseye**, it code can quickly get complex. Anything you do to make the code easier to read and understand will help you write better, more error-free code.

The lines beginning with // are *comments*. Like blank lines, they also don't perform any action or make any changes. Unlike blank lines, they contain text, but anything after the // characters and up to the end of the line is ignored. Comments are notes that programmers add to code to provide additional information about it.

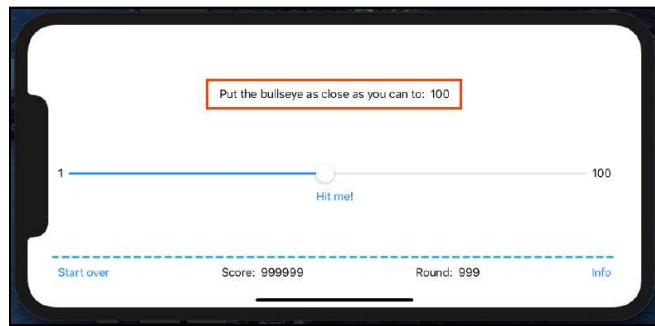
Programmers use comments for a number of purposes, including:

- Indicating who wrote the code and when. That's what the comments at the start of the **ContentView.swift** are. Comments like this are often put at the start of the file, and Xcode automatically does this with all its source code files.
- Explaining what sections of code are for. That's what you're doing with these comments. They mark the different sections of the code, such as where the properties and methods of the **ContentView** object go, the individual rows in the user interface, and so on.
- Providing a summary of what the code does, especially in cases where it might otherwise be difficult to understand.
- Giving additional background information that's not made clear in the code.

- Acting as a reminder to either fix something broken in the code or to add something missing to the code. In this sort of comment, many developers add words like TODO or HACK so that they can find these reminders again easily with a “search” function. You added a couple of TODO comments in the body variable to note that you need to add code to define the slider and score rows in the user interface.

Laying out the target row

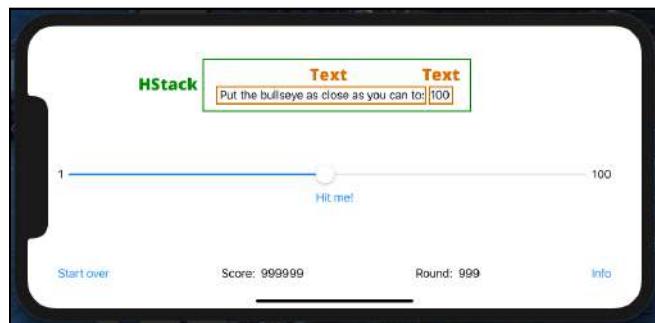
Let’s start with the text at the top of **Bullseye**’s screen (highlighted below), which tells the user the target value they’re aiming for:



The target text

This text challenges the user to move the slider to a specific value. You could use a single **Text** object with both the challenge and the target value, but we’ll go with **two**: One for the challenge and one for the target value.

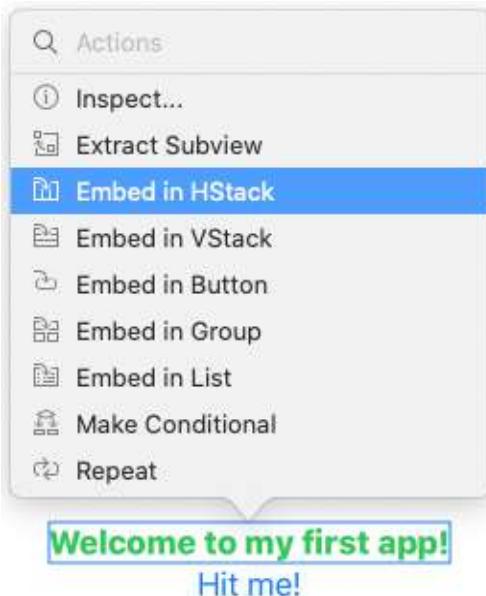
You want to lay these two **Text** objects out side by side, which sounds like an opportunity to use an **HStack**. Here’s a screenshot with some extra graphics showing how the **Text** objects and **HStack** fit together:



The HStack containing the target text’s Text views

There are a couple of ways to create this **HStack**. One way is to type the necessary code into the Editor. But instead, you'll do it another way: By using the Canvas. You'll embed the existing “Welcome to my first app!” text into an **HStack**, and then you'll change its text.

- If the **Resume** button is visible in the upper-right corner of the Canvas, press it.
- In the Canvas, command-click on **Welcome to my first app!**. Select **Embed in HStack** from the menu that appears:



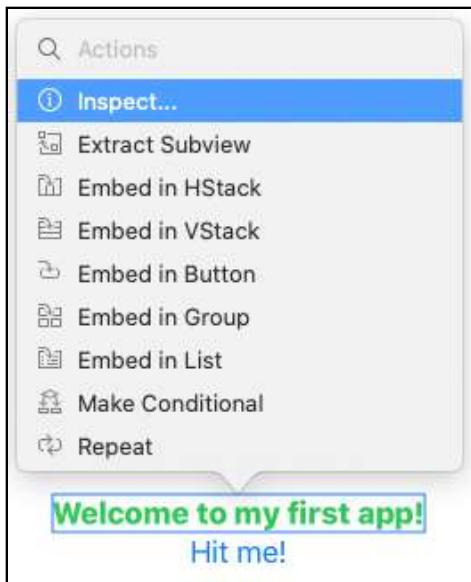
Embedding the ‘Welcome to my first app!’ Text view into an HStack

This embeds the “Welcome to my first app!” view into an **HStack**. If you look at the Editor, you'll see that the code in the **Target row** section has been updated to reflect what you did on the Canvas:

```
// Target row
HStack {
    Text("Welcome to my first app!")
        .fontWeight(.black)
        .foregroundColor(.green)
}
```

Next, you'll do even more with the Canvas. It's time to change the text.

- In the Canvas, command-click on **Welcome to my first app!**. Select **Inspect...** from the menu that appears:



Inspecting the 'Welcome to my first app!' view

- Use the inspector to change the text to “Put the bullseye as close as you can to:”:



Editing the 'Welcome to my first app!' view

This changes the **Text** view, and will also update the code in the editor:

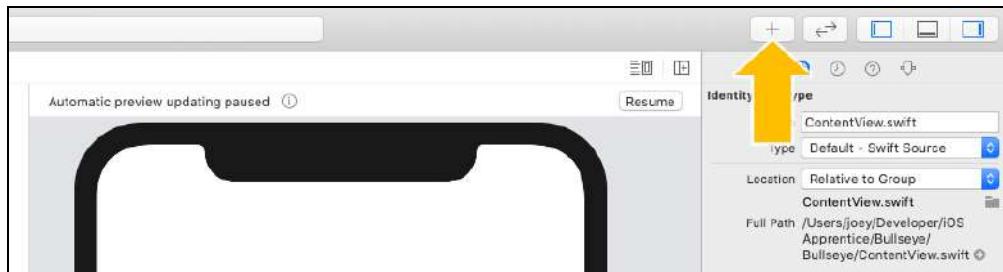
```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
        .fontWeight(.black)
        .foregroundColor(.green)
}
```

- We don't want the "Put the bullseye as close as you can to:" text to be green and bold, so remove the calls to the **Text** view's `fontWeight()` and `foregroundColor()` methods so that the code in the *Target row* section looks like this:

```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
}
```

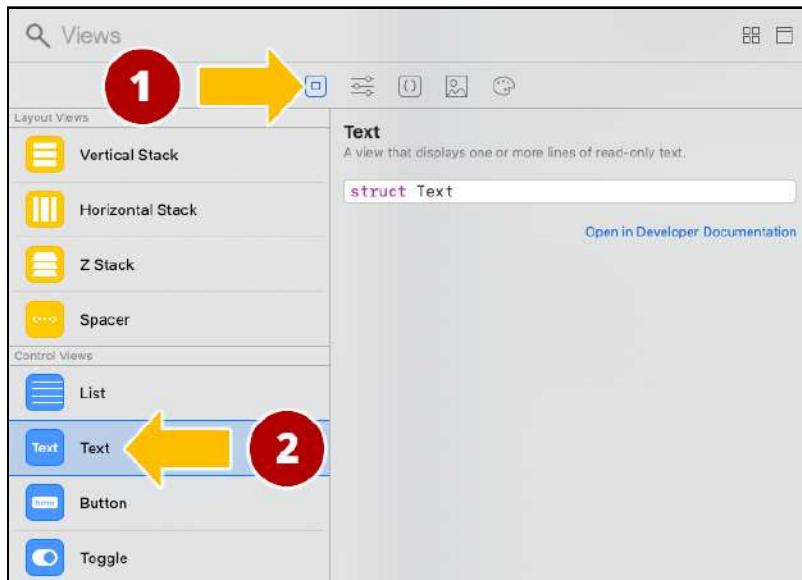
The next step is to add a new **Text** view to the **HStack**, to the right of the "Put the bullseye as close as you can to:" view. Use the library and the editor to do this.

- Open the library — remember, you do this by pressing the **Library** button, which is the + button located near the upper-right corner of the Xcode window:



The Library button

- The Library window will appear. Make sure that you've selected the **Views** tab, which is the leftmost one, with the square-within-a-square icon, then highlight the **Text** item in the Library list:



The library window, with the Text view selected

- Click and start dragging the **Text** item from the library list and onto the editor as shown below:

```
23 // Target row
24 HStack {
25     Text("Put the bullseye as close as you can to:")
26 }
27
```

Text Text

Dragging a text view from the library into the editor

- As you drag the item onto the editor, a blank line should appear below the `Text("Put the bullseye as close as you can to:")` line. When this blank line appears, drop the item. A new **Text** object will appear in the code, which will now look like this:

```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
    Text("Placeholder")
}
```

- Change the placeholder text (literally “Placeholder”) in the newly-added **Text** view to **100**. The code for the Target row should now look like this:

```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
    Text("100")
}
```

- Build and run the app. You’ll see that the two **Text** views have replaced the “Welcome to my first app!” message:

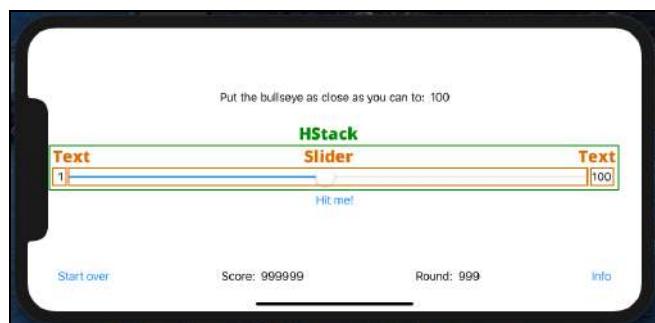


The app with the Target row added

The target value in the second **Text** view, **100**, is a placeholder. You’re using 100 because this text view will eventually contain a random number between 1 and 100. 100 is the largest – and more importantly, *widest* – text that will go into this view.

Laying out the slider row

Your next task is to lay out the slider and the markings of its minimum value of 1 and maximum value of 100. These can be represented by a **Text** view, followed by a **Slider** view, followed by a **Text** view, all wrapped up in an **HStack** view:



The slider and accompanying text, with the Slider and Text views and HStack pointed out

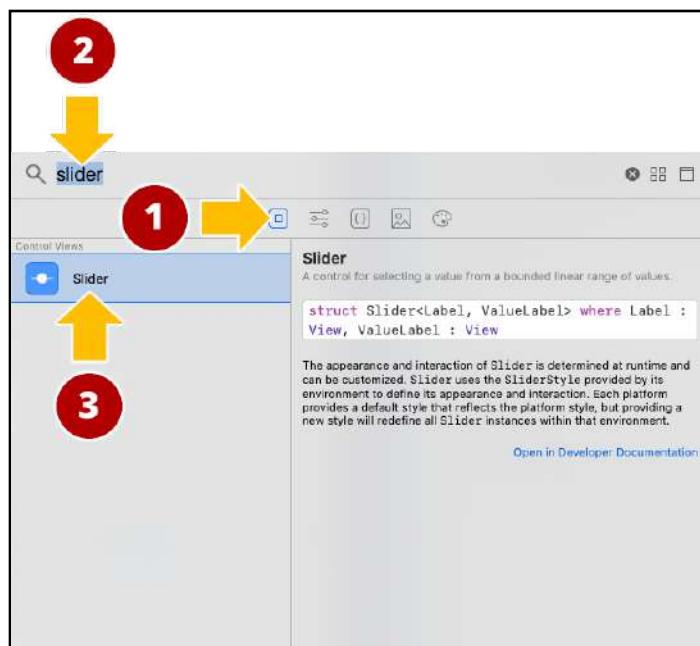
This time, try setting this up by writing some code. After all, you have some examples of using `HStack` and `Text` in code already!

- In the `Slider row` section, replace the `// TODO: Add views for the slider row here.` line so that code looks like this:

```
// Slider row
HStack {
    Text("1")
    Text("100")
}
```

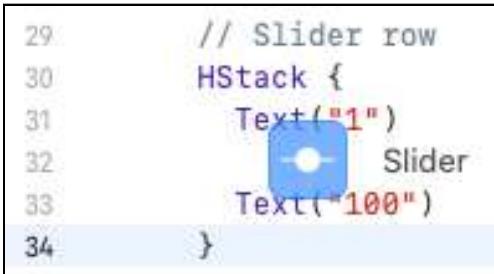
This defines the `1` and `100` `Text` views that are on the left and right sides of the slider, respectively. You could also type in the code to create the slider, but first, take a look at one more feature of the library.

- Open the library (press the + button near the upper right corner of the Xcode window). Make sure that you've selected the **Views** tab, which is the leftmost one, with the square-within-a-square icon, then type `slider` into the library's search text field, which is just to the right of the magnifying glass icon. As you type, the library's views in the library's list will disappear until only the slider remains.



The `Slider` in the library, after using the `Search` text field

- Drag the **Slider** view from the library list onto the editor, and drop it in the empty line in the `// Slider row` section of the code between the `Text("1")` line and the `Text("100")` line:

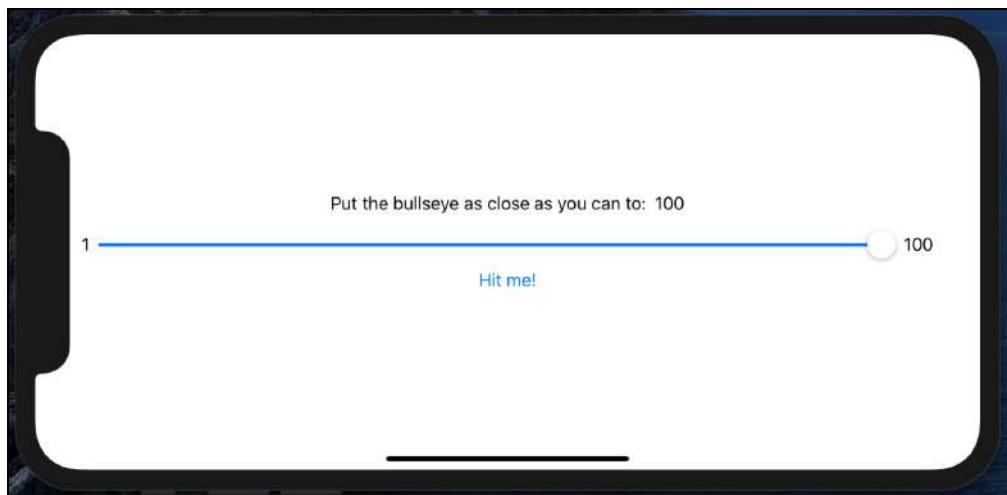


Dragging the Slider from the library and onto the code

The code for the Slider row should now look like this:

```
// Slider row
HStack {
    Text("1")
    Slider(value: .constant(10))
    Text("100")
}
```

- Build and run the app. The Slider row is now visible:



The app with the Slider row added

If you tried to move the slider, you probably noticed that it's stuck on the right side. This has something to do with the `.constant(10)` that you gave as the **Slider**'s `value:` argument.

This is a good time to look at a useful Xcode feature that allows you to find out more about just about anything in the code that has a name.

- While holding down the **option** key, move the cursor over the `.constant` in the line `Slider(value: .constant(10))`. The word `constant` should change color and the cursor's shape should change to a question mark (?). Click on `constant` and a pop-up window will appear with more details about it:



The app with the Slider row added

The summary text — “Creates a binding with an immutable value” — may sound like meaningless techno-babble to you now, but the words **binding** and **immutable** should be hints that it has something to do with **state**.

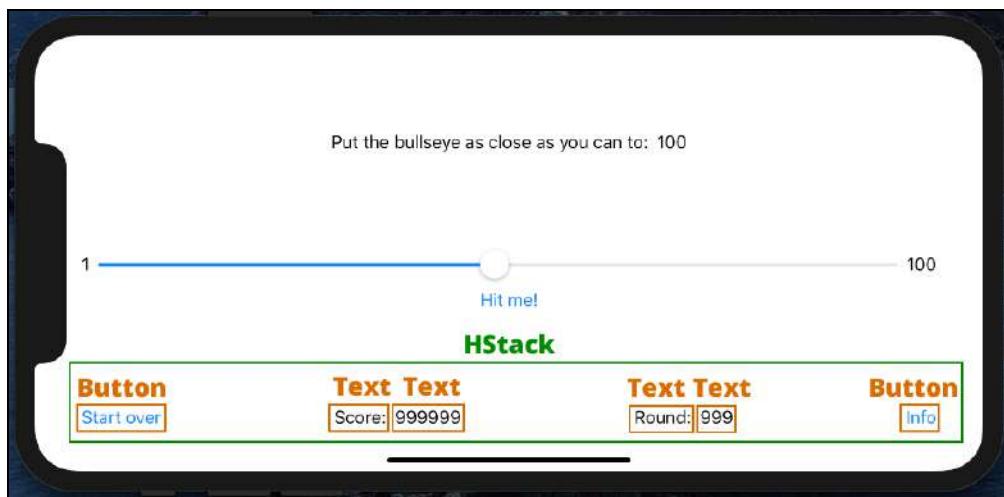
You’ll deal with the mystery of the stuck slider soon enough, but you’ll finish setting up the controls first.

Laying out the Button row

Here’s a little gift for you: The Button row’s already done!

Laying out the Score row

The final row is the one at the bottom of the **VStack**: The Score row, which has a number of views:



The Score row, with the Button and Text views pointed out

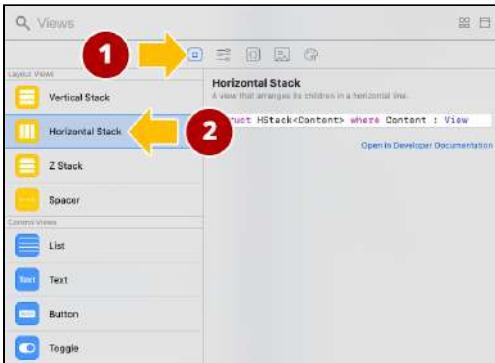
These views are:

- A **Button** view labeled **Start over**: The user will press this button to start a new game. It will reset the score to 0 and the round to 1.
- Two **Text** views for the score: One containing the text “Score” and one containing the score value. For now, the score will be set to a placeholder value of 999999.
- Two **Round** views: One containing the text “Round”, and one containing the number of the current round. For now, this number will be set to a placeholder value of 999.
- A **Button** view labeled **Info**: The user will press this button to get more information about the game. It will take the user to another screen, where they’ll see the additional information.

Don’t forget that these should be all in a row, which means that you need an **HStack**, so start with that.

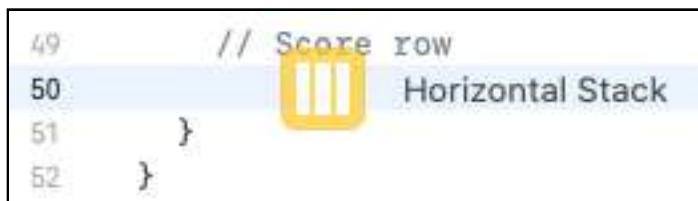
► In the *Score row* section, replace the `// TODO: Add views for the score, rounds, and start and info buttons here.` with a blank line.

- Open the library (once again, press the + button near the upper right corner of the Xcode window). Make sure that you've selected the **Views** tab (the leftmost one, with the square-within-a-square icon), then find the **Horizontal Stack** view:



HStack in the Library

Drag this view onto the editor and drop it into the blank line after `// Score row`:



Dragging the HStack from the library onto the code

The code starting at `// Score row` should now look like this:

```
// Score row
HStack {
    Text("Placeholder")
}
```

Notice that Xcode didn't just give you an **HStack**; it also included a **Text** view. You didn't ask for it, so try removing it by deleting the `Text("Placeholder")` line.

Here's what happens:



Xcode showing error indicators everywhere after being presented with an empty HStack

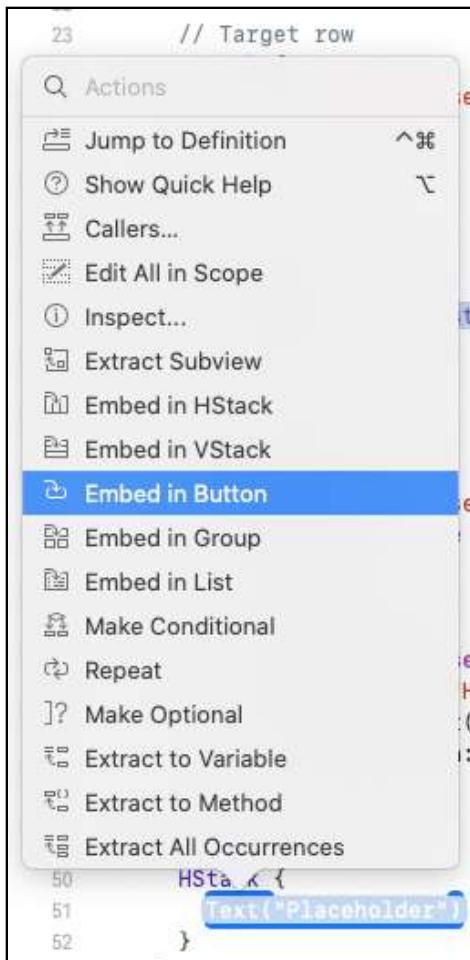
Xcode is diligent about alerting you to errors in your code, and it turns out that an empty **HStack** is an error. An **HStack** must contain at least one view, so Xcode did a little preemptive error prevention by throwing in a “free” **Text** view when creating the **HStack**. You’ve probably figured out that the same rule applies to **VStack** views.

- If you got experimental and tried removing the **Text** view from the **HStack**, try undoing the change. If that doesn't work, simply type in code so that the code starting at `// Score` row looks like this:

```
// Score row
HStack {
    Text("Placeholder")
}
```

Since **Button** views contain **Text** views, Xcode has included a useful feature that takes advantage of this.

- Command-click on the `Text` keyword in the `// Score row` section of the code. You should see this pop-up menu:



Command-clicking on `Text` to reveal the pop-up menu and selecting 'Embed in Button'

- Scroll down the menu and select **Embed in Button**. Xcode will embed the `Text` view inside a `Button` view and the code will now be:

```
// Score row  
HStack {  
    Button(action: {}) {  
        Text("Placeholder")  
    }  
}
```

- Copy the Button code and paste it so that the code becomes the following:

```
// Score row
HStack {
    Button(action: {}) {
        Text("Placeholder")
    }
    Button(action: {}) {
        Text("Placeholder")
    }
}
```

- Change the **Text** view in each **Button** view so that the first one becomes the **Start over** button and the second one becomes the **Info** button. The code should now look like this:

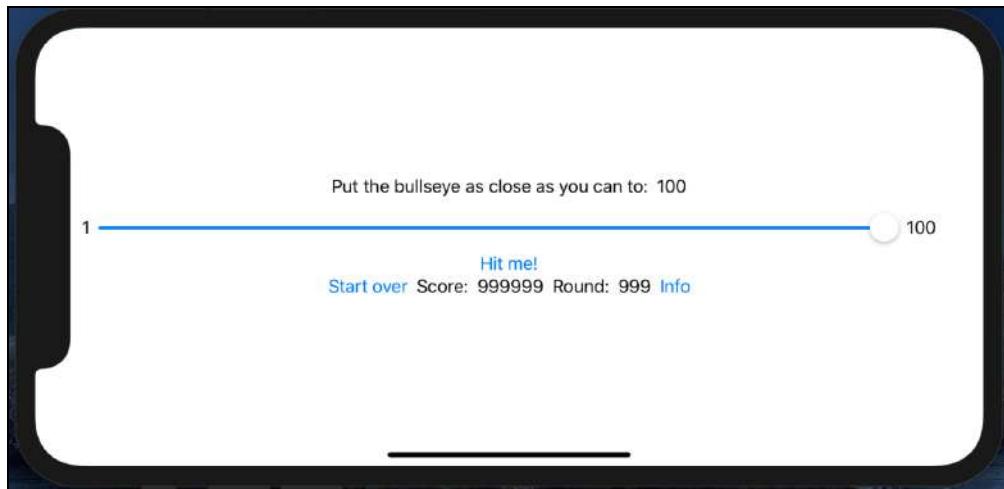
```
// Score row
HStack {
    Button(action: {}) {
        Text("Start over")
    }
    Button(action: {}) {
        Text("Info")
    }
}
```

By now, you're probably beginning to get the hang of creating user interfaces the SwiftUI way. Next, add the remaining **Text** views between the **Button** views.

- Add **Text** views so that the // Score row code becomes this:

```
// Score row
HStack {
    Button(action: {}) {
        Text("Start over")
    }
    Text("Score:")
    Text("999999")
    Text("Round:")
    Text("999")
    Button(action: {}) {
        Text("Info")
    }
}
```

- Build and run the app. The Simulator should display this:



The app with all the views, running in the Simulator and looking compressed

All the controls are there, but the app looks somewhat compressed. In the next section, you'll fix that.

Introducing spacers

It's time to bring some **Spacer** views into your app. As their name implies, these views are designed to fill up space.

When you put a **Spacer** view into an **HStack**, it expands to fill up the remaining horizontal space.



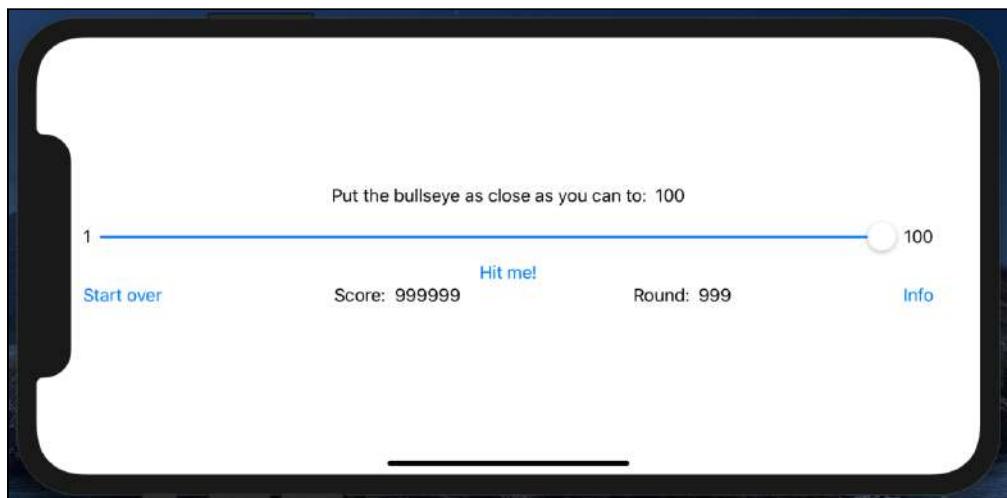
A spacer in an HStack, sandwiched between two views

Next, see what happens when you use a spacer in the *Score row*.

- Add **Spacer** views to the *Score row* code so that it looks like this:

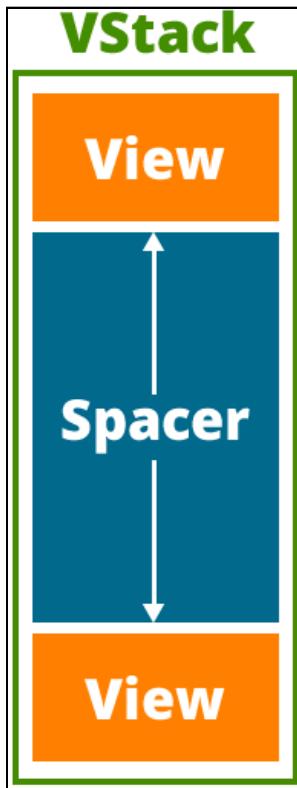
```
// Score row
HStack {
    Button(action: {}) {
        Text("Start over")
    }
    Spacer()
    Text("Score:")
    Text("999999")
    Spacer()
    Text("Round:")
    Text("999")
    Spacer()
    Button(action: {}) {
        Text("Info")
    }
}
```

- Build and run the app. The Score row should look a lot less compressed:



The app with *Spacer* views added to the Score row

Spacer views also work in **VStack** views. There, they expand to fill up the remaining vertical space.



A spacer in a VStack, sandwiched between two views

Let's put some **Spacer** views in your app's **VStack** in the following places:

- Above the Target row.
 - Between the Target row and the Slider row.
 - Between the Slider row and the Button row.
 - Between the button row and the Score row.
- Change the code for **ContentView** so that it looks like this:

```
struct ContentView: View {  
    // Properties  
    // ======
```

```
// User interface views
@State var alertIsVisible: Bool = false

// User interface content and layout
var body: some View {
    VStack {
        Spacer()

        // Target row
        HStack {
            Text("Put the bullseye as close as you can to:")
            Text("100")
        }

        Spacer()

        // Slider row
        HStack {
            Text("1")
            Slider(value: /*@START_MENU_TOKEN@*/.constant(10)/*
 *@END_MENU_TOKEN@*/
            Text("100")
        }

        Spacer()

        // Button row
        Button(action: {
            print("Button pressed!")
            self.alertIsVisible = true
        }) {
            Text("Hit me!")
        }
        .alert(isPresented: self.$alertIsVisible) {
            Alert(title: Text("Hello there!"),
                  message: Text("This is my first pop-up."),
                  dismissButton: .default(Text("Awesome!")))
        }

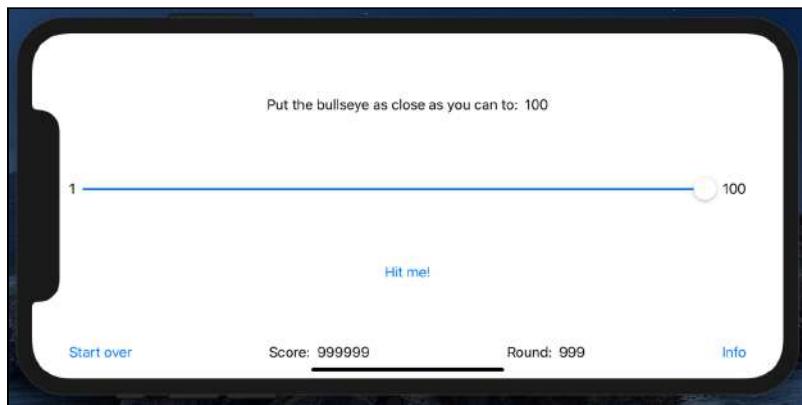
        Spacer()

        // Score row
        HStack {
            Button(action: {}) {
                Text("Start over")
            }
            Spacer()
            Text("Score:")
            Text("999999")
            Spacer()
            Text("Round:")
            Text("999")
            Spacer()
        }
    }
}
```

```
        Button(action: {}) {
            Text("Info")
        }
    }

    // Methods
    // =====
}
```

► Run the app. It's looking a whole lot better!



The app with Spacer views added to the Score row and between rows

There's just one last little change you need to make to the app's layout: The Score row is a little too close to the bottom. In the next section, you'll find out how to fix that.

Adding padding

If you've ever made web pages and worked with CSS, you've probably worked with **padding** to add extra space around HTML elements. SwiftUI views can also have padding, which you can set using one of the `padding()` methods¹, which all views have.

In this case, we want to add some padding to the bottom of the Score row. You can do this by calling the `padding()` method for the **HStack** containing the Score row.

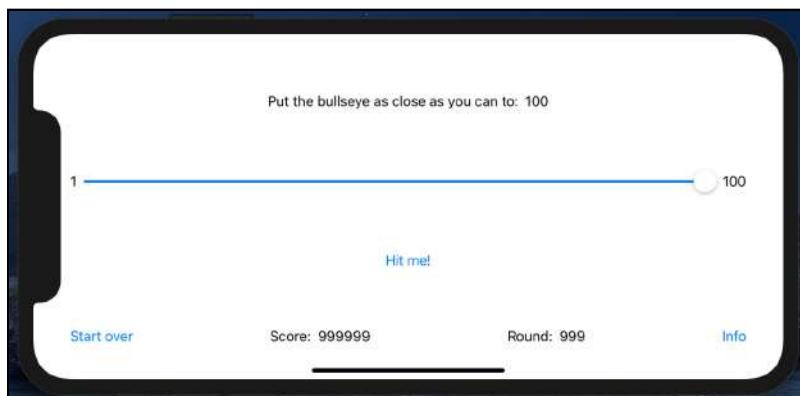
- Add the following to the end of the *Score row* code, so that it ends up looking like this:

```
// Score row
HStack {
    Button(action: {}) {
        Text("Start over")
    }
    Spacer()
    Text("Score:")
    Text("999999")
    Spacer()
    Text("Round:")
    Text("999")
    Spacer()
    Button(action: {}) {
        Text("Info")
    }
}
.padding(.bottom, 20)
```

The `padding()` method takes two arguments:

1. The set of edges to pad. There are a number of options for this argument including `.bottom` (which you just used), `.leading` and `trailing` (for the leading and trailing edges, respectively), `.top`, `.horizontal` (which pads both leading and trailing edges), `.vertical` (which pads both top and bottom edges) and `.all`.
2. The amount of padding. This is a number of screen distance units called **points**, which you'll cover a little later on.

- Build and run the app. It looks a whole lot better!



The app with all the spacers and padding on the Score row

Solving the mystery of the stuck slider

Let's get back to why the slider doesn't work. As mentioned earlier, it has to do with **state**.



Store with two signs: 'Open' and 'Sorry we're closed'. Creative Commons photo by "cogdogblog" — Source: <https://www.flickr.com/photos/cogdog/7155294657/>

If you've ever gone to a restaurant where the sign said they were open, only to find that they were closed when you tried to enter, you've experienced what happens when a user interface, in this case the “Open” sign, doesn't match the state (the place was actually closed).

This kind of problem happens when the user interface and state aren't connected. In the restaurant example, keeping the “open/closed” sign in sync with the restaurant's actual open/closed state means that someone has to make sure that the sign is always providing the right information. If you've ever worked in the food service industry, you know that the kind of dedication and reliability needed to make sure that the sign is always right is rare.

You may have seen this sort of thing in software as well. One example is the user interface of an email app that tells you that you have a new message, but when you check, it turns out that you'd already read it. You've probably seen other examples of user interfaces that were wrong about their application's state. As apps grow, their state becomes more complex, and it's all too easy to forget to update some part of the user interface when some state detail changes.

SwiftUI solves the problem of the mismatch between user interface and application state by creating bindings between them. In a SwiftUI application, when you update

some property that's part of its state, any user interface elements bound to that property automatically update to reflect the change.

You can also choose to make two-way bindings, where if the user changes the value of some user interface element that's bound to some state property by pressing a button, entering a value into a text field or moving a slider, that property is automatically updated.

This means that in SwiftUI, user interface controls have to be connected to some kind of value. Sometimes, that value is a constant, which is often the case with **Text** views:

```
Text("This is a constant value")
```

The code that currently sets up the slider in **Bullseye** is similar:

```
Slider(value: .constant(10))
```

In this case, the slider is bound to a state property that is set to 10 and can't be changed; therefore the slider's position can't be changed, either.

Making the slider movable

The solution to the mystery of the stuck slider is to connect it to a state variable, whose value *can* change. So now, declare one. You'll call it `sliderValue` and set its initial value to 50.

► Add a declaration for the `sliderValue` variable to the *Properties* section of `ContentView`, so that the lines starting with the `// User interface views` comment look like this:

```
// User interface views
@State var alert isVisible: Bool = false
@State var sliderValue: Double = 50.0
```

Remember, `@State` marks the variable as part of the application's state and tells Swift to watch it for changes to its value. The `var sliderValue: Double = 50.0` part is Swift for "The variable `sliderValue` is a `Double`, and its value is 50.0"

Note: A **Double** is a Swift data type that represents numbers with decimal points really, really, *really* precisely. We're using it because that's the kind of value that sliders work with. We'll talk more about data types soon.

Now that there's a state variable for the slider, it's time to connect the two together!

► Change the line where the slider is set up to the following:

```
Slider(value: self.$sliderValue, in: 1...100)
```

This code creates — or in programmer-speak, **instantiates** — a **Slider** view. Here's what the three parameters do:

- **value**: Specifies the *binding* that connects a state variable to the slider. If you think of `sliderValue` as the value of the slider's current position, `$sliderValue` (note the \$) is the two-way connection between the slider and the `sliderValue`. Changing the value of `sliderValue` affects the position of the slider's thumb (the thing on the slider that moves), and moving the thumb on the slider changes the value in `sliderValue`. Since `sliderValue` is a property of the object that you're in, you precede it with `self..`.
- **in**: Specifies the range of values that the slider covers. `1...100` represents the range of values starting at 1 at its minimum and ending and including 100 at the maximum.

► Build and run the app. The initial value of `sliderValue` is 50.0, which means that the slider's initial position is midway between the left and right ends. You can also move the slider now!

Reading the slider's value

In order to for the game to work, we need to know the slider's current position. Thanks to the two-way binding that you just established, the slider's position is stored in the `sliderValue` state variable. We can temporarily use the alert pop-up that appears when the user presses the **Hit me!** button to display this value.

Here's the code for the Button row:

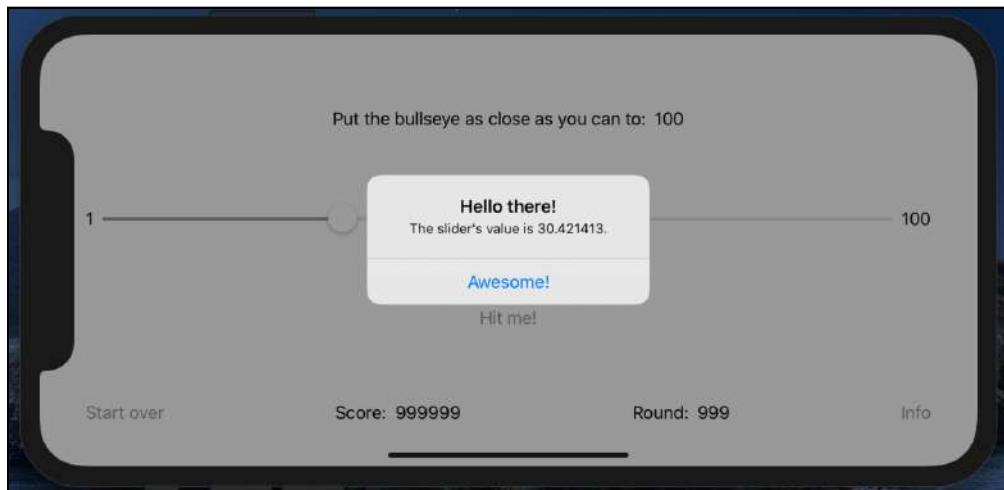
```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertVisible) {
    Alert(title: Text("Hello there!"),
        message: Text("This is my first pop-up."),
        dismissButton: .default(Text("Awesome!")))
}
```

Now, change the argument that you provide to the `message:` parameter so that it displays the slider's current value.

► Change the call to the **Button**'s `presentation()` method to:

```
.alert(isPresented: self.$alertIsVisible) {  
    Alert(title: Text("Hello there!"),  
          message: Text("The slider's value is \  
(\(sliderValue))."),  
          dismissButton: .default(Text("Awesome!")))  
}
```

► Build and run the app, move the slider anywhere you like, then press the **Hit me!** button. The alert pop-up will appear, and it'll give you a painfully precise readout of the slider's value:



The app displays a painfully precise slider value

Take a closer look at the `Text` view that was passed to the alert pop-up's `message:` parameter:

```
Text("The slider's value is \((sliderValue).")
```

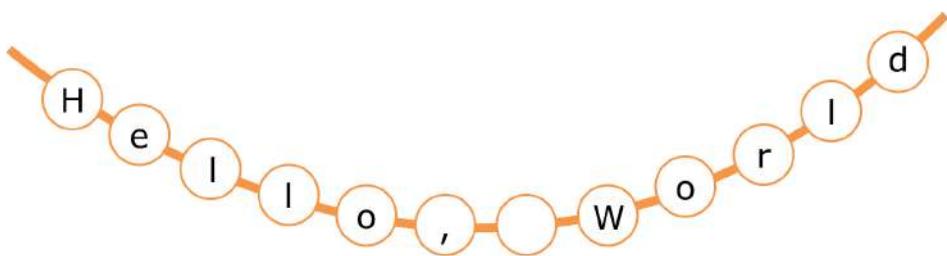
The alert pop-up uses the text before and after `\(sliderValue)` — “The slider's value is” and “”. On the other hand, `\(sliderValue)` is replaced by the value in the `sliderValue` variable. Think of the `\(...)` as a placeholder: “The slider's value is X”, where X will be replaced by the value of the slider.

Data types

Before doing more work on the app, take a moment to consider **data types** in Swift. These classify the different kinds of data that Swift can work with.

Strings

You've already done a fair bit of work with **strings**, which represent text information. Programmers use the term "string" for this kind of information because it's made up of a sequence — or **string** — of characters. Think of characters in a string as being like pearls on a necklace:



A string of characters

Here are some the strings you've used so far:

- "Welcome to my first app!"
- "Hit me!"
- "Awesome!"
- "The slider's value is \((sliderValue)."

You used the first three when making **Text** views, and the last one to display the slider's value in the alert pop-up.

Creating a string is simple in Swift: Just put text between a pair of "double quote" characters (""). Other languages let you create strings by using either "single quote" characters ('') or double quotes, but Swift doesn't. Strings should be **delimited** — that's computer-science fancy-talk for "started with and ended with" — by double quotes. And they must be plain double quotes, not typographic "smart quotes".

To summarize, this is the proper way to make a Swift string...

```
"I am a good string"
```

...and these are wrong:

- 'I should have double quotes'
- ''Two single quotes do not make a double quote''
- "My quotes are too fancy"

Inserting variables' values into strings

Anything between the characters `\(` and `)` inside a string is special — instead of taking that information literally, Swift evaluates whatever is between those characters and turns the result into a string.

You used this in the alert pop-up to display the value of the slider's current position, which is stored in the state variable `sliderValue`. You did it by using this string to create the alert pop-up:

```
"The slider's value is \(sliderValue)."
```

If you run the app and press the **Hit me!** button without moving the slider (which means its value will be 50), the pop-up *doesn't* display this text:

The slider's value is \(sliderValue).

Instead, it says:

The slider's value is 50.000000.

Inside a string, Swift treats the `\(` and `)` as markers for the beginning and ending of something it should evaluate, converts that thing into a string, and then inserts it into the rest of the string.

This feature isn't limited to numerical variables — you can also put string variables between `\(` and `)`. Suppose a variable named `weather` contains the value **sunny**. Swift interprets the string "We expect the weather to be `\(weather)` today." as **We expect the weather to be sunny today.**.

You can even put calculations between `\(` and `)`. Swift interprets the string "Your answer, `\(1 + 2)`, is correct." as **Your answer, 3, is correct.**

Filling in the blanks this way is a very common way to build strings in Swift and is known as *string interpolation*.



Numbers

Swift has a number of ways to represent numerical values. The two that you'll probably use the most are:

- **Double**: This is short for **double-precision floating-point number**, which is a fancy computer-science way of saying “painfully precise number.” It’s accurate to 15 or 16 digits, which should be more than enough precision for most calculations. It also has a large range, being able to represent numbers as small as 10⁻³⁴⁵ and as large as 10³⁰⁸. You’ll use these when you need to store and work with numbers with decimal points.
- **Int**: This is short for **integer**, which simply means “whole number”. You’ll use these when you need to store and work with numbers without decimal points.

You’ve already worked with a **Double** when you created the `sliderValue` variable to store the position of the slider. The slider reports its position as a **Double**, so that’s what we use to store its value.

Booleans

You’ll often have to store values of the “yes/no” or “on/off” kind. That’s what **Bool** variables — short for “Boolean” — are for. They can store only two values: `true` and `false`.

Boolean values, which are often referred to simply as **Booleans**, are often used in programs to make decisions. You’ll soon learn about the `if` statement, which performs a set of instructions if some condition is true, and another set of instructions (or no instructions) if the condition is false.

Variables

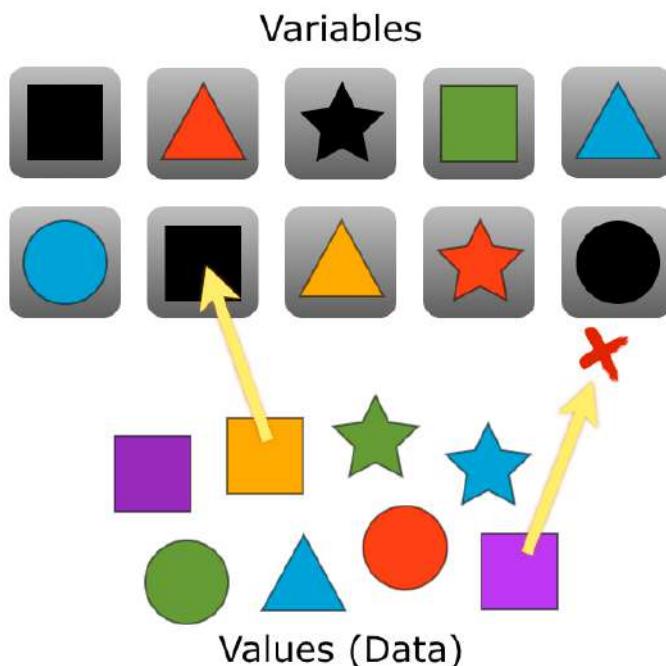
If you’re new to programming, it’s important to remember that programs are really made of just two things:

1. **Data**, which is information that we want to manage, process and perform calculations with.
2. **Instructions**, which tell the computer to how to manage, process and perform calculations with the data.

Programs store their data in variables. So far, **Bullseye** stores its data in just two variables — one to keep track of the slider’s position, and one to keep track of

whether or not the alert pop-up is visible. The previous chapter told you to think of variables as temporary storage containers, each one storing a single piece of data. Just as there are containers of all shapes and sizes, data also comes in all kinds of shapes and sizes, which we call **data types**. You've just looked at a few data types: Strings, Ints, Doubles and Bools.

The idea is to put the right shape in the right container. The container is the variable and its type determines what “shape” fits. The shapes are the possible values that you can put into the variables. You might want to think of variables as being like children's toy blocks:



Variables are containers that hold values

You won't just put stuff in the container and then forget about it. You'll often replace the contents with a new value. When the thing that your app needs to remember changes, you take the old value out of the box and put in the new value. That's the whole point behind variables: They can *vary*.

For example, `sliderValue` will change every time the user moves the slider, and you'll change the value of `alertVisible` to `true` every time the user presses the

Hit me! button. The size of the storage container and the sort of values the variable can remember are determined by its **data type**, or just **type**.

You specified the Double type `sliderValue` variable, which means this container can hold very precise numbers. Double is one of the most common data types. There are many others though, and you can even make your own.

The idea is to put the right shape in the right container. The container is the variable and its type determines what “shape” fits. The shapes are the possible values that you can put into the variables.

You can change the contents of each box later, as long as the shape fits. For example, you can take out a blue square from a square box and put in a red square — the only thing you have to make sure of is that both are squares.

But you can’t put a square in a round hole: The data type of the value and the data type of the variable have to match. You can’t put a string value into an integer variable, and you can’t put an integer value into a string variable. The type of the value has to match the variable.

How long do variables last?

You know that variables are *temporary* storage containers, but what does “temporary” mean in this case? How long does a variable keep its contents?

Unlike meat or vegetables, variables won’t spoil if you keep them for too long. A variable will hold onto its value indefinitely, until you put in a new value or destroy the container altogether.

Each variable has a certain lifetime, also known as its **scope**, which depends on exactly where in your program you defined that variable. In the case of **Bullseye**, both `alertIsVisible` and `sliderValue` stick around for just as long as their owner, `ContentView`, does. Their fates are intertwined.

`ContentView` exists for the duration of the app, and therefore so do `alertIsVisible` and `sliderValue`. They don’t get destroyed until the app quits. Soon, you’ll also see variables that are short-lived (also known as **local variables**).

Making the slider less annoyingly precise

There is such a thing as too much precision. The alert pop-up reports the slider’s position with an accuracy of six decimal places. We want the game to be challenging,



but not *that* challenging! The app should report the position of the slider as a number between 1 and 100 inclusive, with no decimal points.

The `Slider` reports its position as a `Double`, which is a very precise number with a decimal point. Right now, the app simply takes this number and displays it in the alert pop-up. We want the pop-up to round the position to the closest whole number and report the position as an `Int`.

Rounding a Double to the nearest whole number

Every Swift data type comes with a set of methods to act on that data. Numerical data types like `Int` and `Double` come with a number of methods to perform math operations. `Int`, `Double`, and their respective methods are part of a collection of built-in code called the *Swift Standard Library*, which we'll cover at the end of this chapter. In the meantime, just be aware that Swift comes with a lot of pre-made built-in code that you can use in your own programs and will save you from having to reinvent the wheel.

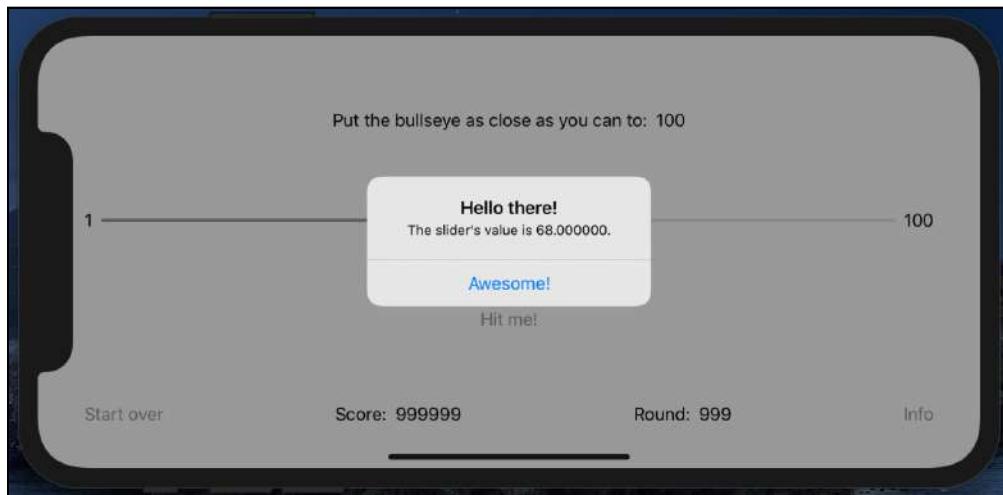
The `Double` type has methods that are useful for working with numbers with decimal points. One of these is `rounded()`, which takes the original value and gives back — or as we say in programming, **returns** — a rounded version of the value. For example, if you call `rounded()` on a `Double` variable containing the value `4.2`, it will return the value `4.0`.

`rounded()` uses “schoolbook rounding,” which means that if the fractional part of the number is **0.5** or larger, it rounds up to the higher value. It rounds **4.5** up to **5.0**, **4.4999** to **4.0**, and **-4.5** to **-5.0**.

► Update the code for the Button row so that it looks like the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertIsVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text("Hello there!"),
          message: Text("The slider's value is \
(\(self.sliderValue.rounded()))"),
          dismissButton: .default(Text("Awesome!")))
}
```

- Build and run the app. Move the slider wherever you like, and then press **Hit me!**. You should see something like this:



The app displays a rounded, but still painfully precise slider value

This is an improvement, but there's still the matter of those trailing zeros. There are a couple of ways to remove them:

- One way is to format the number to show only the digits before the decimal point.
- Another way would be to convert the number into an integer (an `Int`) and display that value.

Later on in the development of **Bullseye**, we're going to generate a target number, which will be a whole number that the user will have to match by positioning the slider. This means that the target number will be an `Int`. We'll need to compare the target number to the value of the slider's position, so the second approach sounds like the better one.

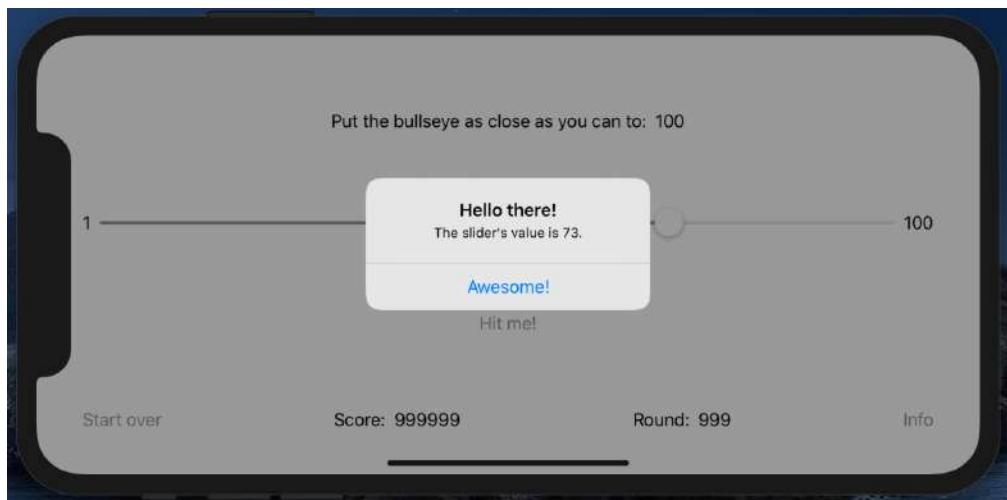
The simplest way to convert a `Double` value into an `Int` value is to create a new `Int` value and use the `Double`'s value to **initialize** it.

- Update the code for the Button row so that it looks like the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertIsVisible = true
}) {
    Text("Hit me!")
}
```

```
.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text("Hello there!"),
          message: Text("The slider's value is \
(Int(sliderValue.rounded()))."),
          dismissButton: .default(Text("Awesome!")))
}
```

- Build and run the app. Move the slider wherever you like, then press **Hit me!**. You should see something like this:



The app displays a whole number slider value

Here's the part that's changed:

```
message: Text("The slider's value is \
(Int(sliderValue.rounded()))."),
```

In this code, `sliderValue.rounded()` is being fed into `Int()`. This tells Swift to create a new `Int` using `sliderValue.rounded()`, which is the value of the slider rounded to the nearest whole number.

Any time you see a capitalized word followed by parentheses ((and)), braces ({ and }) or both, it usually means that something new is being created — or in programming terms, **instantiated** — using the things between the parentheses and braces. You've already seen many examples of this. Here's one:

```
Text("Here is some text.")
```

This instantiates a new `Text` view containing the text “Here is some text.”

Here's a more complex example:

```
 VStack {  
     Text("Here is some text.")  
     Text("And here's more text!")  
 }
```

This instantiates a new **VStack** view, and inside it, two **Text** views are also instantiated.

The Swift Standard Library

You could've written a method to round a **Double** to the nearest whole number, but you didn't have to. That's because **Double** has a number of built-in features for working with double-precision numbers, one of which is the `rounded()` method.

Double, `rounded()`, and many other similar goodies are part of a large collection of code that make up the Swift Standard Library. It contains a lot of useful pre-made functionality that you can use in your own apps, including:

- Data types, including the ones you worked with in this chapter: **Int**, **Double**, **String**, and **Bool**.
- Methods that go with each of those data types, which let you do more things with them. You've already used one of them: **Double**'s `rounded()` method. In the next chapter, you'll use a couple more Standard Library methods to generate a random number and remove the “minus sign” from a negative number.
- Functions, such as `print()`.
- Other more advanced features, all of which can serve as the building blocks for your apps that you don't need to write yourself.

It's pretty much impossible to do any Swift programming without making use of something in the Swift Standard Library — that how useful it is. It does more than just provide useful features. It will also save you so much time because it contains all sorts of things that you'd otherwise have to code yourself. Better yet, all the features it provides have the advantage of having been thoroughly tested — and not just by Apple's quality assurance team, but by the entire Swift developer community who use it regularly (and who complain loudly when it's not working as they expect).

You'll use the Swift Standard Library as often as you use Xcode, so it's worth reviewing it regularly as you learn Swift and iOS programming. It evolves with each version of Swift, so even experienced developers, looking at its online documentation from time to time to see what changed.



To see everything that the Swift Standard Library has to offer, visit the Swift Standard Library home page, located in Apple's developer documentation site at https://developer.apple.com/documentation/swift/swift_standard_library.

Key points

So far, you've done the following:

- Set up all of **Bullseye**'s basic user interface elements.
- Made the slider work
- Converted the original alert pop-up to report the slider's position value.

Along the way, you've learned a lot, including:

- What portrait and landscape orientations are and how to make an app landscape-only.
- Different types of views.
- A little more about state and variables.
- Data types.

In the next chapter, you'll take the app and turn it into a functioning game!

You can find the project files for the app up to this point under **03 - Slider** in the **Source Code** folder.



Chapter 4: Swift Basics

Joey deVilla

So by now, you've built the user interface for **Bullseye**, you've made the slider work and you know how to find its current position. That already knocks quite a few items off your to-do list. In this chapter, you'll take care of a few more items on that list. Here's what this chapter will cover:

- **Generating and displaying the target value:** Select the random number that the player will try to match using the slider and display it onscreen.
- **Calculating the points scored:** Determine how many points to award to the player based on how close they came to positioning the slider at the target value.
- **Writing methods:** You've used some built-in methods so far, but built-in methods can't cover everything. It's time to write your own!
- **Improving the code:** Make the code more readable so that it's easier to maintain and improve and less error-prone.
- **Key points:** A quick review of what you learned in this chapter.



Generating and displaying the target value

First, you need to come up with the random number that the user will try to match using the slider. Where can you get a random number for each game's target value?

Generating (sort of) random numbers

Random numbers come up a lot when you're making games because games need to have an element of unpredictability. You can't get a computer to generate numbers that are truly random and unpredictable, but you can employ a **pseudo-random number generator** to spit out numbers that at least appear to be random.

To make random numbers, pseudo-random generators typically start with a **seed value**, a number derived from an event that isn't easy to predict. Some examples include the number of milliseconds the system has been running, the user's most recent keyboard input, mouse clicks and other events. They feed this seed value into a mathematical formula that creates a list of wildly different numbers that *appear* random.

If you were to run a pseudo-random number generator that always started with the same seed value, it would always generate the same set of numbers in the same order. But it's pretty unlikely that the events Swift uses for its seed values will be exactly the same each time, so the random number generators built into Swift are good enough for everyday applications like games.

For the curious: macOS and iOS constantly update a file named **/dev/random** with hard-to-predict values from the system's device drivers, and you can use that file as a source of seed values for pseudo-random number generators. You can see what it contains by opening the **Terminal** app and entering `od -d /dev/random` on the command line. You'll see a stream of numbers, which you can stop by typing **control+c**.

Generating a random target number

Swift's data types for numbers, which include `Int` and `Double` numeric types, have a method that lets you generate random numbers in a given range.

- Add the following line to the start of **ContentView.swift**:

```
@State var target: Int = Int.random(in: 1...100)
```

The set of @State variables should now look like this:

```
@State var alert isVisible: Bool = false
@State var sliderValue: Double = 50.0
@State var target: Int = Int.random(in: 1...100)
```

Take a closer look at the line you just added.

The first half of the new line, `@State var target: Int`, doesn't include anything that you haven't already seen. It says that you're declaring a variable named `target` that holds `Int` (i.e., integer, or whole number) values, and that `target` makes up part of the state for `ContentView`. As a state variable, Swift will watch `target` for changes to its value, then take any necessary action when that value changes.

The second half of the line, `= Int.random(in: 1...100)`, might be new to you. It assigns an initial value to `target`, and that value is a random number between 1 and 100 inclusive. The random number comes from the `random()` function built into the `Int` data type to get a pseudo-random integer between 1 and 100. The `1...100` part is a **closed range**, which you should read as “all the numbers between 1 and 100, including 1 and 100.” The `...` part indicates that you want the range to include the last number (100) as part of the range.

You could also use a **half-open range**, which you specify with these characters: `..<`. `1..<100` is an example of a half-open range, and you should read it as “all numbers between 1 and 100, including 1, *excluding* 100.” If you wanted to specify a range of numbers from 1 to 100 inclusive using a half-open range, you'd do it with `1..<101`.

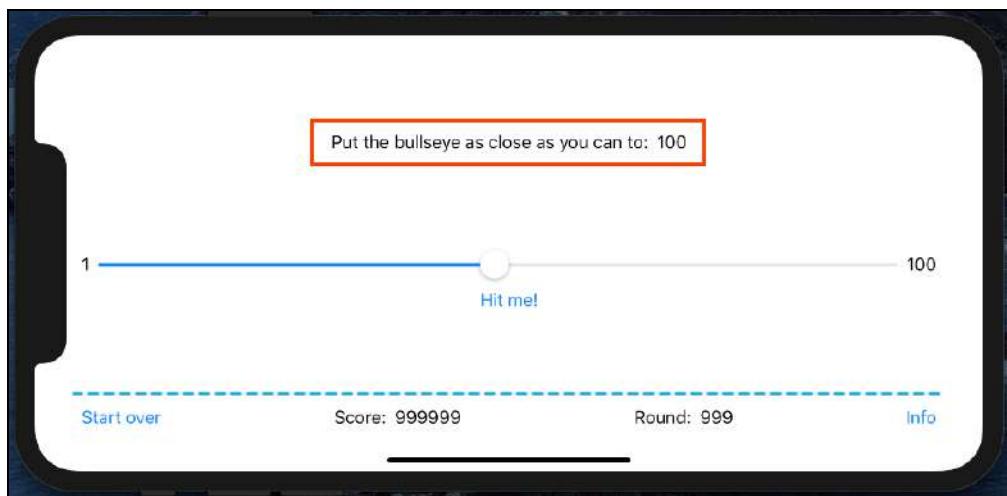
With this single line, the app now generates a new random target value every time it starts. Your next step is to make that target value visible to the user.

Displaying the target value

- Scroll down to the part of the body variable that begins with the comment line `// Target row:`

```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
    Text("100")
}
```

This code defines the text near the top of the screen, which tells the user what the target value is:



The app screen, with the target text highlighted

Right now, the text that displays the score value holds the placeholder text “100”:

```
Text("100")
```

- ▶ Change it so that it displays the value inside `target`, the state variable you just created:

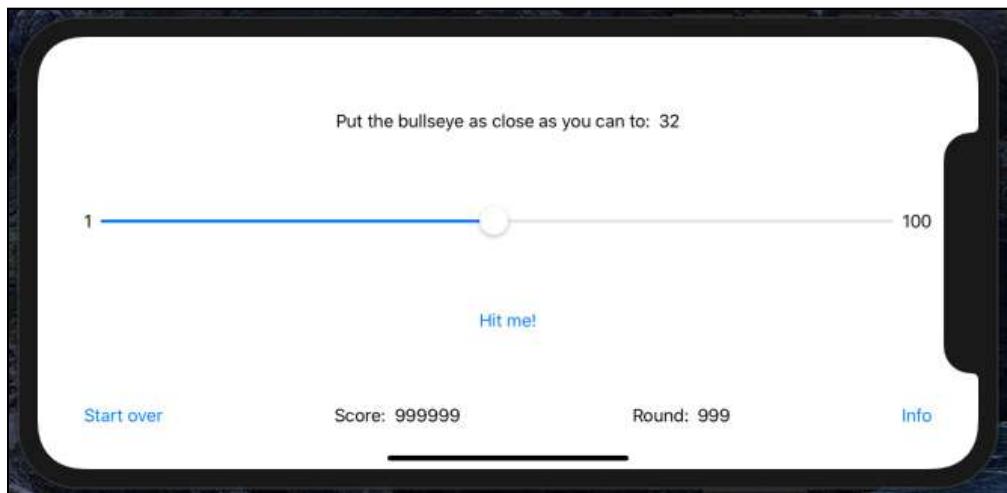
```
Text("\(self.target)")
```

You’ve just replaced the `100` with `\(self.target)`. As you learned in the previous chapter, the characters `\(` and `)` have a special meaning when used inside a string. They mark the beginning and end of something that Swift should evaluate, convert into a string and then insert into the rest of the string.

In this case, the object between `\(` and `)` that Swift needs to evaluate is `self.target`. Remember that any time you see code in the form of `object.feature`, you should read it as code that makes use of an object’s feature. In this case, the object is `self`, which is Swift for “the object that this code lives in.” In this case, `self` refers to `ContentView`. The feature is `target`, which is one of `ContentView`’s variables... the one you just created.

Since target is a state variable (because you marked it with the keyword `@State`), the Text object will always display the current value of target, even when target changes. You'll see this in action in the next chapter, when you incorporate multiple rounds into the game.

- Build and run the app. There's only a 1 in 100 chance that your target will be to put the bullseye as close as possible to 100:



The app screen, now with a random target value

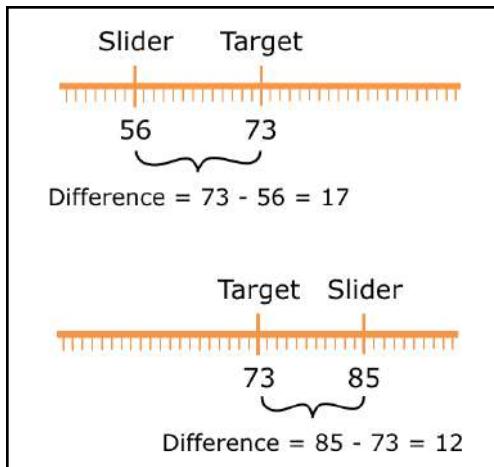
- Stop the app and run it again, then do that again a few more times. 99% of the time when you restart the app, the target value will be different from the previous one.

Calculating and displaying the points scored

Now that you have both the target value *and* a way to read the slider's position, as you learned from the previous chapter, you can calculate how many points the player scored.

How close is the slider to the target?

The closer the slider is to the target when the player presses the **Hit me!**, the more points they should receive. To calculate the score for each round, you look at how far the slider's value is from the target:



Calculating the difference between the slider position and the target value

A simple approach to finding the distance between the target and the slider is to subtract `sliderValue` from `target`.

Unfortunately, that gives a negative value if the slider is to the right of the target because now `sliderValue` is greater than `target`.

You need some way to turn that negative value into a positive value — or you end up subtracting points from the player's score (unfair!).

Doing the subtraction the other way around — `sliderValue` minus `target` — won't always solve things either because, then, the difference will be negative if the slider is to the left of the target instead of the right.

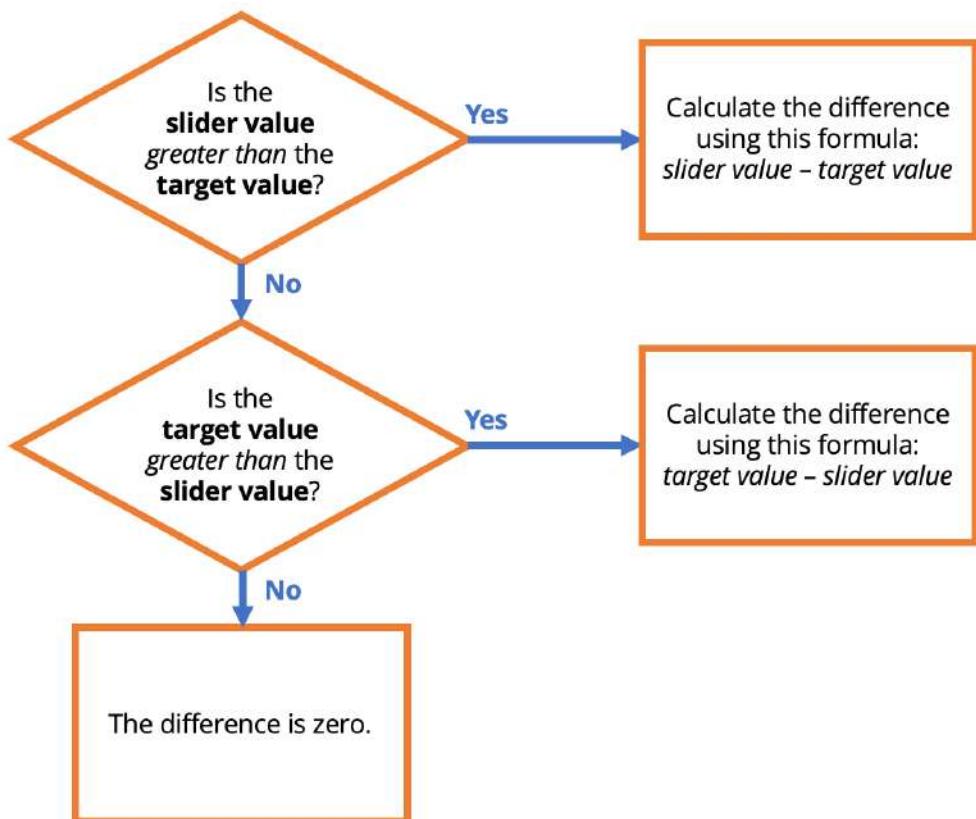
Hmm, it looks like you're in trouble here...

Exercise: How would you frame the solution to this problem if you wanted to solve it in natural language? Don't worry about how to express it in code for now. Just think it through in plain language.

I came up with something like this:

- If the slider's value is greater than the target value, then the difference is: Slider value minus the target value.
- However, if the target value is greater than the slider value, then the difference is: Target value minus the slider value.
- Otherwise, both values must be equal, and the difference is zero.

If you prefer to think in pictures, here's the solution in flowchart form:



Calculating the difference, in flowchart form

This will always lead to a difference that is a positive number, because you always subtract the smaller number from the larger one. Do the math to test it out:

- If the slider is at position 60 and the target value is 40, then the slider is to the right of the target and has a larger value. The difference is $60 - 40 = 20$.

- However, if the slider is at position 10 and the target is 30, then the slider is to the left of the target and has a smaller value. The difference is $30 - 10 = 20$.

Calculating the points scored

The number of points the player receives should depend on the difference between the slider value and the target value:

- When the slider is right on top of the target, the difference between the slider value and the target value is 0. In this case, the player should receive the maximum number of points for getting a bullseye.
- When the slider is as far as possible from the target, it means that the slider is at one end and the target is at the opposite end. In this case, the player should receive the minimum number of points for being way off.

For now, we'll use a simple formula:

points = 100 - difference between slider value and target value

With this formula, the player earns 100 points for placing the slider right at the target value. In the case where the slider is at one end and the slider is at the opposite end, the player scores one point, just for showing up.

Algorithms

In coming up with a way to calculate the score, you've come up with an **algorithm**. That's a fancy term for a process or series of steps to follow to perform a calculation or solve a problem. This algorithm is very simple, but it's an algorithm nonetheless.

There are many algorithms that you can adapt for use in your own programs. As you gain more experience programming, you might run into well-known ones such as **quicksort**, for sorting a list of items, and **binary search**, for quickly searching a sorted list. The academic field of computer science centers around studying algorithms and finding better ones. You can find these algorithms in books or online and you can use them in your own programs, saving you from having to reinvent the wheel.

Although there are many published algorithms, you'll still have to come up with your own algorithms to fit the specific needs of the program you're writing. Some will be as simple as the one above; others will be complex and might cause you to throw up your hands in despair. That's part of the fun of programming.

You can describe any algorithm using plain language or diagrams — use whatever method works better with the way you think. Remember that an algorithm, no matter how complex it is or how fancy a result it produces, is just a set of steps to follow.

If you ever get stuck and you don't know how to make your program calculate something, step away from the computer and think the steps through. Take out a piece of paper — still one of the best software engineering tools out there — and try to write or draw out the steps. Ask yourself how would you perform the calculation or solve the problem by hand. Once you know how to do that, converting the algorithm to code should be a piece of cake.

Writing your own methods

Back near the start of Chapter 2, you read about the concept of functional decomposition, which is the process of tackling a large project by breaking it into sub-projects, and possibly breaking the sub-projects into even smaller sub-projects until they are manageable.

Whenever you find yourself thinking something along the lines of, “At this point in the app, I need to tackle this sub-project,” that’s a sign that you need to create a method to perform that task. Once you have a method, you can simply activate it by calling it by name.

You’ve already made use of a pre-defined method: `rounded()`, which comes built-in with the `Double` data type. You used it to round the slider’s current value to the nearest whole number:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertVisible = true
}) {
    Text("Hit me!")
}
.presentation(self.$alertVisible) {
    Alert(title: Text("Hello there!"),
        message: Text("The slider's value is \
(Int(self.sliderValue.rounded()))."),
        dismissButton: .default(Text("Awesome!")))
}
```

There couldn’t possibly be a built-in method for every purpose, but that’s not a problem because you can write your own. Start by writing a method to calculate how many points the player should receive.



Implementing a basic method for calculating the points to award the player

In building a method to calculate how many points to award to the player, you’re going to use an approach called **stepwise refinement**. This means starting by building the simplest thing that could possibly work and then refining it over a number of steps until you get the desired result.

The first version of this method will simply award 100 points to the player, no matter where the player positioned the slider. At this point, you want it to simply report a number.

- Add the following to the end of `ContentView`, after the **Methods** comments and before `ContentView`’s closing brace `()`:

```
func pointsForCurrentRound() -> Int {  
    return 100  
}
```

Now, take a look at this new method, starting with the first line:

```
func pointsForCurrentRound() -> Int {
```

This line is the method’s **signature**, which specifies three things:

1. The name of the method.
2. The information that the method must receive.
3. The information that the method provides as a result.

Just like `structs` and `vars`, methods start with a keyword that specifies what kind of thing you’re defining. For methods, this keyword might surprise you: it’s `func`, which is short for **function**. You use this keyword because methods are a kind of function, which is a general term for a block of code that you can call by name and which may or may not return a value at the end.

The name of the method follows the `func` keyword. In this case, it’s `pointsForCurrentRound()`.

You’ve probably noticed that method names end with parentheses (the `()` characters). That’s a convention borrowed from the way you write mathematical functions. That’s how you can tell whether something’s a variable or a function: Function names end with parentheses, while variable names don’t.



Some methods require additional information before they can perform their task. You've already used such a method, `rounded()`, which requires a rounded number. If you were defining a method that required additional information, you would put that information within the parentheses. Since `pointsForCurrentRound()` doesn't require additional information, you don't have to put anything between the parentheses.

After the parentheses comes this symbol: `->`. It doesn't mean "minus" followed by "greater than." Rather, you should interpret it as an arrow pointing rightward, and should read it as "returns a value of the following data type." This is immediately followed by `Int`. This means that when `pointsForCurrentRound()` has completed its task, it should give back — or as we say in programming, **return** — an integer value.

After the method signature comes the **body** of the method, which goes between braces (the `{` and `}` characters). The body of the method specifies what the method does.

The body of the `pointsForCurrentRound()` method is a single line:

```
return 100
```

The `return` keyword defines the result that the method provides. `return 100` makes the method provide a result of 100 when called.

So you've completed your first step, reporting a number that you've set! Now that you have a basic method for calculating the points to award the player, you can see it in action.

Calling the method and viewing its result

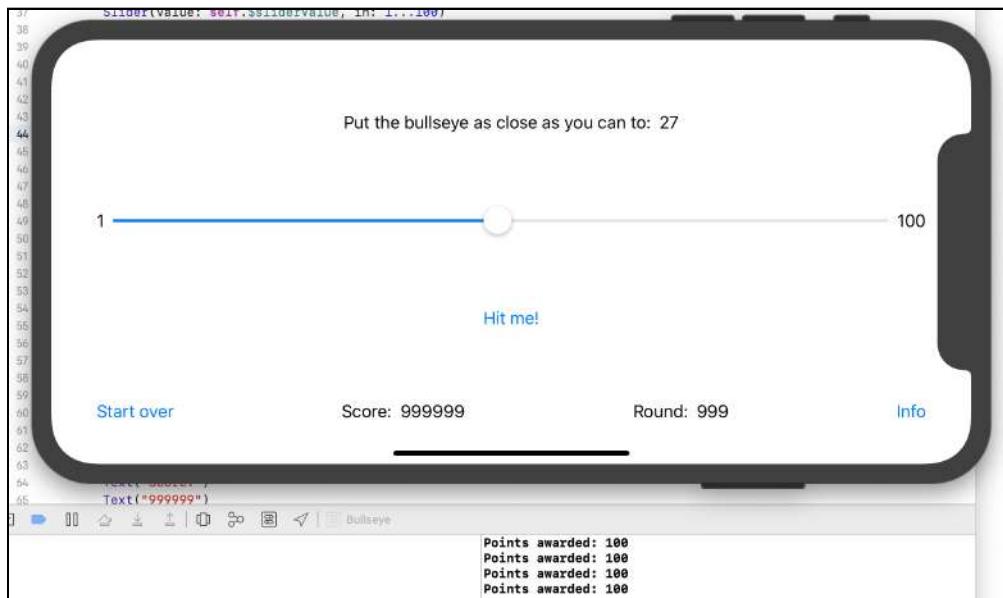
To see how it works, use `print` to show what `pointsForCurrentRound()` returns.

- Change the `print` statement in the **Button row** section so that it displays the results of `pointsForCurrentRound()`. The result should look like this:

```
// Button row
Button(action: {
    print("Points awarded: \(self.pointsForCurrentRound())")
    self.alertVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertVisible) {
    Alert(title: Text("Hello there!"),
        message: Text("You hit the button!"))
}
```

```
        message: Text("The slider's value is \  
        (Int(self.sliderValue.rounded()))."),  
        dismissButton: .default(Text("Awesome!"))  
    }
```

- Build and run the app and press the **Hit me!** button a few times, keeping an eye on the Xcode’s debug console. You should see a line that says **Points awarded: 100** for each press of the **Hit me!** button:



Seeing the points awarded in the debug console

`pointsForCurrentRound()` returns the number that you set, but that’s not the *correct* one. Next, you’ll fix that so the game awards the right number of points.

Making `pointsForCurrentRound()` actually calculate points

Now that `pointsForCurrentRound()` returns a value and the alert pop-up displays that value, it’s time to change the value to the actual number of points that the player should receive.

- Replace the code in `pointsForCurrentRound()` so that the method looks like this:

```
func pointsForCurrentRound() -> Int {  
    var difference: Int  
    if self.sliderValue.rounded() > self.target {  
        difference = self.sliderValue.rounded() - self.target  
    } else if self.target > self.sliderValue.rounded() {  
        difference = self.target - self.sliderValue.rounded()  
    } else {  
        difference = 0  
    }  
    return 100 - difference  
}
```

You've now replaced the code that simply awards the player 100 points no matter how well they played with code that implements the algorithm you created earlier.

The code may *look* right, but Xcode will take exception:

```
func pointsForCurrentRound() -> Int {  
    var difference: Int  
    if self.sliderValue > self.target {  
        difference = self.sliderValue - self.target  
    } else if self.target > self.sliderValue {  
        difference = self.target - self.sliderValue  
    } else {  
        difference = 0  
    }  
    var awardedPoints: Int = 100 - difference  
    return 99  
}
```

● Binary operator '>' cannot be applied to operands of type 'Double' and 'Int'
● Binary operator '-' cannot be applied to operands of type 'Double' and 'Int'
● Binary operator '>' cannot be applied to operands of type 'Int' and 'Double'
● Binary operator '-' cannot be applied to operands of type 'Int' and 'Double'

Xcode complaining loudly

The error messages show what the problem is: You're attempting to compare and subtract two different kinds of things:

- `sliderValue.rounded()`, which is a `Double`.
- `target` which is an `Int`.

To us humans, both `Double` and `Int` values are just numbers, and differentiating between them seems like nitpicking. However, computers represent `Double` and `Int` values very differently, and the compiler treats them as very different things. A computer can't make sense of it.

To compare `sliderValue` and `target` and perform arithmetic on them, you'll need to convert one of them to the same type as the other. Since you're only using a `Double` because sliders use that data type to report their values, we'll convert `sliderValue.rounded()` into an `Int`.

► Update `pointsForCurrentRound()` so it looks like this:

```
func pointsForCurrentRound() -> Int {  
    var difference: Int  
    if Int(self.sliderValue.rounded()) > self.target {  
        difference = Int(self.sliderValue.rounded()) - self.target  
    } else if self.target > Int(self.sliderValue.rounded()) {  
        difference = self.target - Int(self.sliderValue.rounded())  
    } else {  
        difference = 0  
    }  
    return 100 - difference  
}
```

The error messages should be gone now. Before running the app, take a moment to review this new code.

The first line should be familiar:

```
var difference: Int
```

This declares a new variable, `difference`, which you'll need to store the difference between the slider's current position and the target value. Since the difference will be a whole number, it's an `Int`.

Note that you haven't assigned a value to `difference`, you've simply declared it's an `Int`. That's because you'll assign a value to it in the lines of code that follow.

What follows the first line is new:

```
if Int(self.sliderValue.rounded()) > self.target {  
    difference = Int(self.sliderValue.rounded()) - self.target  
} else if self.target > Int(self.sliderValue.rounded()) {  
    difference = self.target - Int(self.sliderValue.rounded())  
} else {  
    difference = 0  
}
```

The `if` construct allows your code to make decisions, and it works much as you expect:

```
if something is true {  
    then do this  
} else if something else is true {  
    then do that instead  
} else {  
    do something when neither of the above are true  
}
```

Basically, you put a **logical condition** after the `if` keyword. If that condition turns out to be true, like if `sliderValue` is greater than `target`, then the code in the block between the `{ }` brackets executes.

However, if the condition isn't true, then the computer looks at the `else if` condition and evaluates that instead. There may be more than one `else if`, and code execution moves one by one from top to bottom until one condition proves to be true.

If none of the conditions are found to be valid, then the code in the final `else` block executes.

In the implementation of this little algorithm, you compare `sliderValue.rounded()` against the `target`. Remember that the slider (and therefore `sliderValue`) is precise to about 6 decimal places, so we're rounding its value to the nearest whole number.

First, you determine if `sliderValue.rounded()` is greater than `target`:

```
if self.sliderValue.rounded() > self.target {
```

The `>` is the **greater-than** operator. The condition `self.sliderValue.rounded() > self.target` is `true` if the value stored in `sliderValue` is at least one higher than the value stored in `target`. In that case, the following line of code executes:

```
difference = self.sliderValue.rounded() - self.target
```

Here, you subtract the smaller value, `target`, from the larger one, `sliderValue.rounded()`, and store the result in `difference`.

Notice how the variable names clearly describe what type of data they contain. Often, you'll see code that's harder to understand, like this:

```
a = b - c
```

It's not immediately clear what's happening here, except that some arithmetic is taking place. The variable names `a`, `b` and `c` don't give any clues as to their purpose or what kind of data they contain. That makes it harder to maintain your code in the future.

Now, go back to the `if` statement. If `sliderValue` is equal to or less than `target`, the condition is untrue (or `false` in computer-speak) and execution will move on to the next condition:

```
 } else if self.target > self.sliderValue.rounded() {
```

The same thing happens here as before, except now you've reversed the roles of `target` and `sliderValue`. The computer will only execute the following line when `target` is the greater of the two values:

```
 difference = self.target - self.sliderValue.rounded()
```

This time, you subtract `sliderValue.rounded()` from `target` and store the result in the `difference` variable.

There is only one situation you haven't handled yet: When `sliderValue.rounded()` and `target` are equal. If this happens, the player has put the slider exactly at the position of the target random number, a perfect score. In that case, the difference is 0:

```
 } else {  
     difference = 0  
 }
```

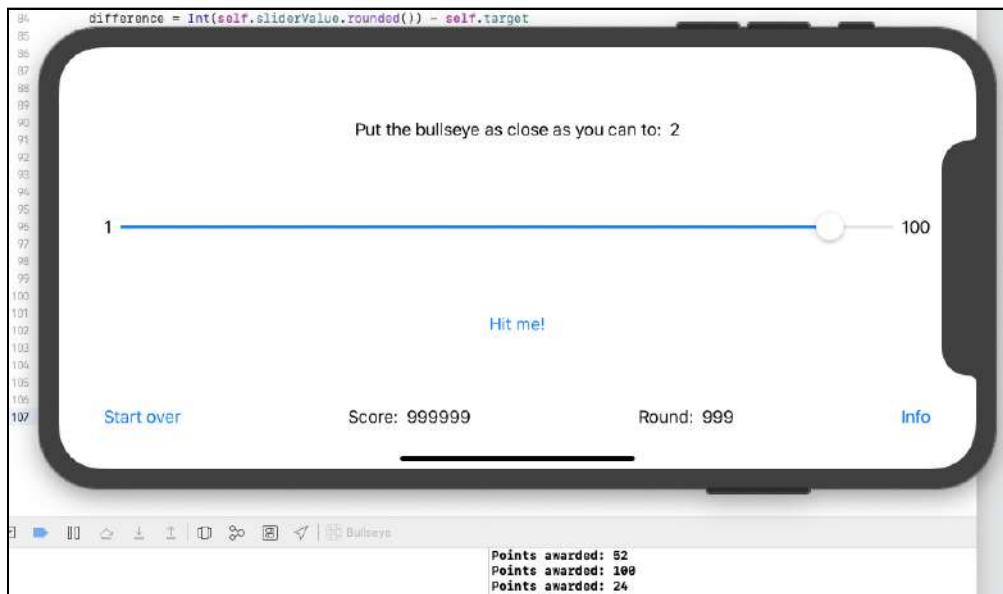
By now, you've already determined that one value is not greater than the other, nor is it smaller. You can only draw one conclusion: The numbers must be equal!

Once you know the difference between the slider and the target values, calculating the number of points to award to the player is simple. It's 100 minus the difference, and the method returns that value:

```
 return 100 - difference
```

Now that you've reviewed everything, you're probably eager to see the method in action!

- Build and run the app and play a few rounds: Move the slider, press **Hit me!**, and look at Xcode's debug console to see how you scored each time:



The first working points calculations

Displaying the points

Now that `pointsForCurrentRound()` properly calculates the points the player earned, it's time to display them. So next, you'll make a change to the alert pop-up so that it displays the results of `pointsForCurrentRound()`.

- Scroll up to the **Button row** section and change the `message:` parameter of the Alert so that the section looks like this:

```
// Button row
Button(action: {
    print("Points awarded: \(self.pointsForCurrentRound())")
    self.alertIsVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text("Hello there!"),
        message: Text("The slider's value is \
(Int(self.sliderValue.rounded())).\n" +
                    "The target value is \(self.target).\n" +
                    "You scored \n"))
```

```
(self.pointsForCurrentRound()) points this round.",  
        dismissButton: .default(Text("Awesome!")))  
}
```

Take a look at the message: parameter for the Alert view:

```
message: Text("The slider's value is \  
    (Int(self.sliderValue.rounded())).\n" +  
        "The target value is \(self.target).\n" +  
        "You scored \(pointsForCurrentRound()) points this  
    round."),
```

The first thing that you should notice is that you made a longer string by “adding” strings together with the + sign. Programmers call this **string concatenation**, and you’ll see it quite often in programming.

The next thing to note is that the first two lines end with \n, but they don’t seem to appear in the pop-up. That’s because inside a string, \n represents the **newline** character, which ends the current line and starts a new one. You’re using it to break a run-on string into three easy-to-read lines.

Note: In Swift, as in many other programming languages, the appearance of the \ character inside a string marks the start of an **escape sequence**, which is a sequence of characters that Swift should translate into another character or a sequence of characters.

You use escape sequences to represent things that would be hard or impossible to represent directly inside a string.

The final thing you should notice is that each of the strings that you concatenate uses the \ (and) characters to include values from a property or method:

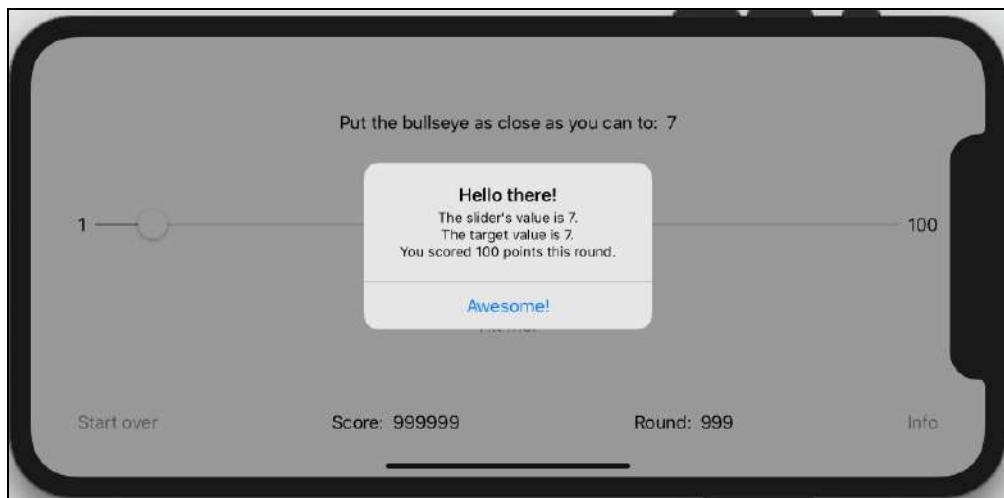
```
"The slider's value is \(Int(self.sliderValue.rounded())).\n"
```

```
"The target value is \(self.target).\n"
```

```
"You scored \(pointsForCurrentRound()) points this round."
```

Remember that you use the `(\ and)` characters for string interpolation, and you now know that the `\\` marks the beginning of an escape sequence. As an escape sequence, this code is not interpreted literally; it's a signal to Swift that anything between `\\` and `)` is first evaluated, then converted into a string, and then included along with the rest of the string.

- Build and run the app. Move the slider and press **Hit me!**, and you'll see the new and improved alert pop-up:



The alert pop-up displaying the slider value, target value and points earned

Improving the code

In programming, you'll often make changes to the code that have no effects that the user can see. These are changes to the app's internal structure, and they're visible only to the programmer — that is, *you*.

But that doesn't mean that they aren't worth doing. There are many reasons for making these changes, such as making the code easier to read, understand and maintain, and making it less likely that you'll introduce bugs into the code. Programmers call the process of making these kinds of changes **refactoring**. That's what you're going to do in this section.

Using a constant to DRY your code

Take a look at the `if` statement inside `pointsForCurrentRound()`:

```
if Int(self.sliderValue.rounded()) > self.target {  
    difference = Int(self.sliderValue.rounded()) - self.target  
} else if self.target > Int(self.sliderValue.rounded()) {  
    difference = self.target - Int(self.sliderValue.rounded())  
} else {  
    difference = 0  
}
```

The repeated use of `Int(self.sliderValue.rounded())` makes it more difficult to read. It's also a calculation that runs *four times*, even though the result is the same each time. That sort of repetition is nothing to a computer, but the programmer who has to read it — and that's probably you — has to take a moment to figure out what that code is trying to do... *four times*.

In case you've forgotten, the code takes the slider's current value, rounds it to the nearest whole number, and then uses that result to create a new `Int` value.

You can improve this code by performing the calculation once and storing its result for later use. You can even give the storage place a meaningful name that makes the code easier to read. To do this, create a new variable, `sliderValueRounded`, to hold the result of `Int(self.sliderValue.rounded())`. You can then use it to make the `if` statement much easier to read.

► Make changes to `pointsForCurrentRound()` so that its code reads like this:

```
func pointsForCurrentRound() -> Int {  
    var sliderValueRounded = Int(self.sliderValue.rounded())  
    var difference: Int  
    if sliderValueRounded > self.target {  
        difference = sliderValueRounded - self.target  
    } else if self.target > sliderValueRounded {  
        difference = self.target - sliderValueRounded  
    } else {  
        difference = 0  
    }  
    return 100 - difference  
}
```

The `if` statement is a lot easier to read now.

You've just changed code that performed a calculation that gave the same answer four times into code that performs the same calculation only once, stores its answer, and uses that answer when needed.



This is an approach that programmers call “Don’t Repeat Yourself,” or **DRY**.

The rationale behind DRY is that it makes code easier to read and to change, and makes it less prone to errors by reducing unnecessary redundancy. You just saw how much easier the code is to follow with the changes you made. If you decided later that you wanted to change the way that you convert the slider’s value, you’d have to do it only once with your new DRY code, instead of *four times* with the previous code.

- Build and run the app to confirm that it still works and that the changes you made aren’t visible to the player.

You may have noticed that even though the program works, Xcode has a suggestion:

```
func sliderValueForCurrentRound() -> Int {
    var sliderValueRounded = self.sliderValue.rounded(1) // Variable 'sliderValueRounded' was never mutated, consider changing its type.
    var difference: Int
    if sliderValueRounded > self.target {
        difference = sliderValueRounded - self.target
    } else if self.target > sliderValueRounded {
        difference = self.target - sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

Xcode says that *sliderValueRounded* was never mutated

“Variable ‘*sliderValueRounded*’ was never mutated?” Why would anyone want to mutate a variable?

Introducing `let` and `constants`

If you’re into science fiction, you probably read “mutated” and thought of it as meaning “exposed to radiation or chemicals and turned into a horrible monster.” However, in programming, “mutated” simply means “changed”.

Take a look at the `if` statement in `pointsForCurrentRound()`, where you use the variable `sliderValueRounded`:

```
if sliderValueRounded > self.target {
    difference = sliderValueRounded - self.target
} else if self.target > sliderValueRounded {
    difference = self.target - sliderValueRounded
} else {
    difference = 0
}
```

In this code, `sliderValueRounded` is compared to `self.target`, subtracted from `self.target`, or has `self.target` subtracted from it. At no point does the value of `sliderValueRounded` ever change once it’s set. It’s not really a variable because it doesn’t vary.

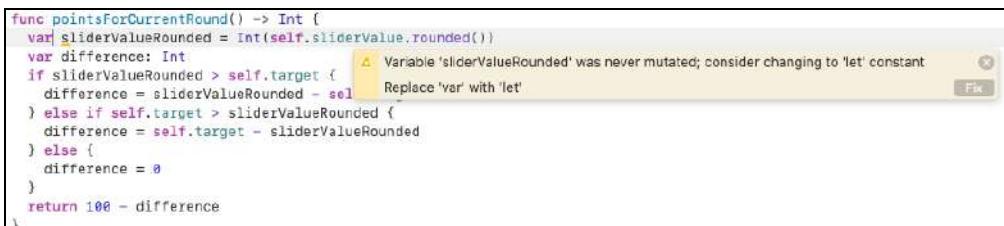
Xcode is suggesting that since `sliderValueRounded` never changes, you should change it from a variable into a constant by using the `let` keyword instead of `var`.



You use `let` to declare **constants**, which are like variables except that their values can be set only once; after that, they can't be changed. If you try to change the value of a constant after its value is set, it causes an error.

Take Xcode's suggestion but this time, instead of manually changing the code, make Xcode do the work for you.

- Click on the icon beside the suggestion, which Xcode calls “warnings”. The warning pop-up will expand and you'll see a suggested fix: Replace `var` with `let`.

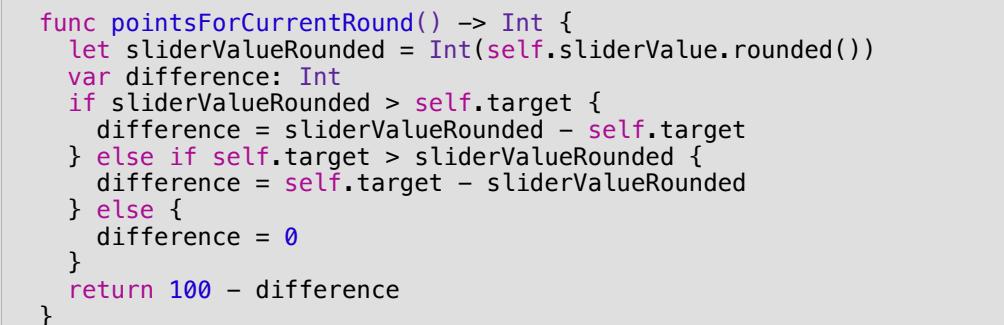


```
func pointsForCurrentRound() -> Int {
    var sliderValueRounded = Int(self.sliderValue.rounded())
    var difference: Int
    if sliderValueRounded > self.target {
        difference = sliderValueRounded - self.target
    } else if self.target > sliderValueRounded {
        difference = self.target - sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

A screenshot of Xcode showing a code editor with a warning for the variable `sliderValueRounded`. The warning message is: "Variable 'sliderValueRounded' was never mutated; consider changing to 'let' constant". Below the message is a "Fix" button. The code itself is a function named `pointsForCurrentRound` that calculates a difference based on the rounded slider value and the target value.

Expanding the warning reveals Xcode's suggested fix

- Click on the **Fix** button to let Xcode take this action and make the fix itself. The code will now look like this:



```
func pointsForCurrentRound() -> Int {
    let sliderValueRounded = Int(self.sliderValue.rounded())
    var difference: Int
    if sliderValueRounded > self.target {
        difference = sliderValueRounded - self.target
    } else if self.target > sliderValueRounded {
        difference = self.target - sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

The code has been modified by Xcode to replace all occurrences of `var` with `let`, specifically for the variable `sliderValueRounded`.

- Build and run the app again. You'll see that it still works just like before and that no changes are noticeable to the player.

Since changing `sliderValueRounded` from a variable to a constant had no noticeable effect on the app, you might be asking “What was the point? Why should Xcode care if a variable never changes and suggest that I use a constant instead?”

The answer is that using a constant here helps prevent bugs. Many coding errors happen because the programmer thinks a variable holds a certain value, only to discover that some other code has changed that value.

This is true in the simple sort of programming you're doing right now, where you only have to worry about one thing happening at a time. In **parallel programming**, where you have many pieces of code running at the same time, or **asynchronous programming**, where you have many pieces of code running independently at unpredictable times and all communicating with each other, it's even easier for code to change data that it shouldn't be changing.

This is why there's a general rule that says whenever possible, use a constant instead of a variable. If you need to store the result of a calculation for later use, you'll often find that the result never changes so you can store it as a constant.

Xcode is good at spotting opportunities to turn variables into constants, but it's not perfect. Take another look at the complete code for `pointsForCurrentRound()`, paying particular attention to how `difference` is set:

```
func pointsForCurrentRound() -> Int {
    let sliderValueRounded = Int(self.sliderValue.rounded())
    var difference: Int
    if sliderValueRounded > self.target {
        difference = sliderValueRounded - self.target
    } else if self.target > sliderValueRounded {
        difference = self.target - sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

The computer science term for the different actions taken as the result of things like the `if` statement is **branches**. In the code above, there are three branches and in each one, `difference` is set to a different value. However, no matter which branch the code takes, the value of `difference` never changes after it's set, so it should be turned into a constant. Remember that a constant is like a variable, except that its value can be set only once.

► Change the line:

```
var difference: Int
```

to:

```
let difference: Int
```

► Build and run the app. Once again, you'll see that it still works, with no noticeable changes.



Another attempt to DRY some code

Since you've defined the `sliderValueRounded` constant in `pointsForCurrentRound()`, try using it in `ContentView`'s body to make it more DRY. Scroll up to the **Button row** section and pay particular attention to `message::`:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertVisible = true
}) {
    Text("Hit me!")
}
alert(isPresented: self.$alertVisible) {
    Alert(title: Text("Hello there!"),
          message: Text("The slider's value is \
(Int(self.sliderValue.rounded())).\n" +
                      "The target value is \((self.target).\n" +
                      "You scored \((pointsForCurrentRound())
points this round."),
          dismissButton: .default(Text("Awesome!")))
}
```

That unwieldy `Int(self.sliderValue.rounded())` that you got rid of in `pointsForCurrentRound()` is in the string that displays the slider value. Since you've already defined `sliderValueRounded` in `pointsForCurrentRound()`, try using it there.

- Change the code that defines the alert pop-up so that it looks like this:

```
alert(isPresented: self.$alertVisible) {
    Alert(title: Text("Hello there!"),
          message: Text("The slider's value is \
(sliderValueRounded).\n" +
                      "The target value is \((self.target).\n" +
                      "You scored \((pointsForCurrentRound())
points this round."),
          dismissButton: .default(Text("Awesome!")))
}
```



While it seems as if this change would make your code more readable, Xcode has issues with it:

```
    .alert(isPresented: self.$alertIsVisible) {
        Alert(title: Text("Hello there!"),
              message: Text("The slider's value is \(sliderValueRounded).\n" +
                           "The target value is \(self.target).\n" +
                           "You scored \(self.pointsForCurrentRound()) points this round."),
              dismissButton: .default(Text("Awesome!")))
    }
```

Xcode displays an error message about `sliderValueRounded`

“Use of unresolved identifier ‘`sliderValueRounded`’” is Xcode’s robotic way of saying “I have no idea of what `sliderValueRounded` is.”

To figure out why you are getting this warning, you’ll change your code back and then you’ll look at why you got an error.

- Change the code that defines the alert pop-up back to this:

```
    .alert(isPresented: self.$alertIsVisible) {
        Alert(title: Text("Hello there!"),
              message: Text("The slider's value is \
(Int(self.sliderValue.rounded())).\n" +
                           "The target value is \(self.target).\n" +
                           "You scored \(pointsForCurrentRound())
              points this round."),
              dismissButton: .default(Text("Awesome!")))
    }
```

The lives and times of variables and constants

If you look at the code in `ContentView`, you’ll see that there are variables and constants in two places:

1. Inside `ContentView` but outside everything else in `ContentView`.
2. Inside `pointsForCurrentRound()`.

Take a look at the code for `pointsForCurrentRound()` again:

```
func pointsForCurrentRound() -> Int {
    let sliderValueRounded = Int(self.sliderValue.rounded())
    let difference: Int
    if sliderValueRounded > self.target {
        difference = sliderValueRounded - self.target
    } else if self.target > sliderValueRounded {
        difference = self.target - sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

`pointsForCurrentRound()` has two constants: `sliderValueRounded` and `difference`. They come into being the moment they are declared within the method (which happens at the `let` statement), and they vanish from existence at the end of the method. You can only refer to them after they've been declared, and they exist only inside `pointsForCurrentRound()`. This restriction would also apply if they were variables.

When you declare constants and variables inside a method, you can only refer to them within that method. This is why programmers call them **local** constants and variables.

Now look at the start of `ContentView`:

```
struct ContentView : View {
    @State var alert isVisible: Bool = false
    @State var sliderValue: Double = 50.0
    @State var target: Int = Int.random(in: 1...100)

    var body: some View {
        ...
    }
}
```

When the app starts, it creates an instance of `ContentView`. You'll learn more about what happens when an app starts later on in this book. Upon the creation of `ContentView`, its variables: `alertIsVisible`, `sliderValue`, `target` and `body` also come into being. Since these variables belong to the `ContentView` instance, programmers call them **instance variables**.

Instance variables are accessible from anywhere within the object they belong to. That's why `ContentView`'s methods (`pointsForCurrentRound()`), references (`sliderValue` and `target`) and variables (other instance variables) can reference them.



When a variable exists and you can reference it, it's **in scope**. A good general rule to follow is that a variable is in scope only inside the braces where you declared it. For example:

```
{  
    var a = 30 // a is in scope here  
  
    // (More code goes here)  
    {  
        var b = a + 1 // Both a and b are in scope here  
  
        // (More code goes here)  
    }  
    // b is no longer in scope.  
  
    print("The value of a is \(a).") // a is still in scope  
}  
  
// Both a and b are no longer in scope.
```

In case you'd forgotten: the // characters mark the start of a comment, which the compiler ignores. You can read about them in more detail below.

Comments

You've probably noticed the green text that begins with // a few times now. As I explained earlier, these are comments. You can write any text you want after the // symbol, and the compiler will ignore any text from the // to the end of the line.

```
// I am a comment! You can type anything here.
```

The best use for comment lines is to explain how your code works. You should try to write your code in a self-explanatory way, but sometimes a little extra explanation can go a long way.

Unless you have the memory of an elephant, you'll probably have forgotten exactly how your code works when you look at it six months later; this is where comments are useful. As I've said before, you want to write code so that the next programmer can easily understand it, because that next programmer might be *you!*

There's another style of comment that covers more than one line. Anything between the `/*` and `*/` markers is a comment:

```
/*
    I am also a comment!
    I can span multiple lines.
*/
```

The `/* */` comments are good for longer comments. They also have another common use: temporarily disable whole sections of source code, which is helpful when you're trying to hunt down a pesky bug.

Remember, the compiler ignores comments, so you can make it ignore one or more lines of code by putting them into a comment. This practice is known as **commenting out**.

Xcode makes it simple to comment out one or more lines of code. Use the **Command-**/ keyboard shortcut to comment/uncomment any currently-selected lines, or if you have nothing selected, the current line.

A second attempt at DRYing some code

Right now, two different places in `ContentView` make use of the same calculation to get the value of the slider: Round it to the nearest whole number and convert it into an `Int`.

You use that calculation in the part of the `body` instance variable that defines the alert pop-up:

```
Alert(title: Text("Hello there!"),
      message: Text("The slider's value is \
(Int(self.sliderValue.rounded())).\n" +
                  "The target value is \((self.target).\n" +
                  "You scored \((pointsForCurrentRound()) \
points this round."),
      dismissButton: .default(Text("Awesome!")))
}
```

It's also used in the `pointsForCurrentRound()` method:

```
func pointsForCurrentRound() -> Int {
    let sliderValueRounded = Int(self.sliderValue.rounded())
    let difference: Int
    if sliderValueRounded > self.target {
        difference = sliderValueRounded - self.target
    } else if self.target > sliderValueRounded {
```

```
        difference = self.target - sliderValueRounded
    } else {
        difference = 0
    }
```

Ideally, you'd like to have a "single source of truth" for the slider's current position as an integer value, and you'd like to make it accessible from anywhere inside `ContentView`.

There are a couple of ways you could make this happen. One way would be to define a new method. It would look like this (don't type this one in; just read it):

```
func sliderValueRounded() -> Int {
    return Int(sliderValue.rounded())
}
```

This new method would make a rounded, whole-number value for the slider position available from anywhere within `ContentView`. If a method were the only way you could do this, you'd use it.

However, there's another way to get this value: a **computed property**, which is a property that acts like a method. In cases where you need a simple calculation based on a property, it's often better to use a computed property instead of a method. We explain the differences between property types in Chapter 26 in Section 3 of the book.

► Add the following to the end of the **User interface views** part of `ContentView`'s **Properties** section, immediately after the line where you declare `target`:

```
var sliderValueRounded: Int {
    Int(self.sliderValue.rounded())
}
```

Declaring a computed property is like declaring an ordinary property in that it begins with `var` or `let` followed by the property name and the property's data type. The difference is that computed properties have a body which defines the value in the property. In the case of the `sliderValueRounded` property, the body contains `Int(self.sliderValue.rounded())`.

Now that you have the computed property, you can use it.

► Change the code that defines the alert pop-up to the following:

```
Alert(title: Text("Hello there!"),
      message: Text("The slider's value is \
(self.sliderValueRounded).\n" +
```

```
        "The target value is \(self.target).\n" +
        "You scored \(self.pointsForCurrentRound())\n"
    points this round."),
    dismissButton: .default(Text("Awesome!")))
```

- Change `pointsForCurrentRound()` to the following. Since `ContentView` now has a `sliderValueRounded` property, there's no longer a need for a `sliderValueRounded` constant within the method:

```
func pointsForCurrentRound() -> Int {
    let difference: Int
    if self.sliderValueRounded > self.target {
        difference = self.sliderValueRounded - self.target
    } else if self.target > self.sliderValueRounded {
        difference = self.target - self.sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

- Build and run the app. It still works in the same way, but underneath, the code is now easier to read and to maintain.

Simplifying the Alert code

`ContentView`'s body defines the user interface; as a result, it's big and can be unwieldy. For example, consider the code in body that generates the alert pop-up:

```
Alert(title: Text("Hello there!"),
      message: Text("The slider's value is \
(self.sliderValueRounded()).\n" +
                  "The target value is \(self.target).\n" +
                  "You scored \(pointsForCurrentRound())\n"
    points this round."),
      dismissButton: .default(Text("Awesome!")))
```

The `message:` parameter has a big chunk of text in it. This is a good place to make use of a method to simplify things.

- Add the following method in `ContentView`'s **Methods** section, just below the `pointsForCurrentRound()` method:

```
func scoringMessage() -> String {
    return "The slider's value is \(self.sliderValueRounded).\n" +
           "The target value is \(self.target).\n" +
           "You scored \(self.pointsForCurrentRound()) points this
```

```
    round."
}
```

You should note a couple of things about this method:

- Because this method returns a result, it has a name that describes that result.
 - This method returns a `String` instead of an `Int`. Methods aren't limited to only returning numbers; they can return all kinds of data, as you'll see in the exercises in this book.
- Incorporate the new `scoringMessage()` method into the alert pop-up code:

```
Alert(title: Text("Hello there!"),
      message: Text(self.scoringMessage()),
      dismissButton: .default(Text("Awesome!")))
```

That's a lot cleaner.

- Once again, build and run the app. Again, you won't see any noticeable changes, but you'll feel the warm glow of satisfaction that you've done a good job refactoring the code.

Removing redundancies

Take a look at the **Properties** section of `ContentView`, particularly the part marked **User interface views**:

```
@State var alertisVisible: Bool = false
@State var sliderValue: Double = 50.0
@State var target: Int = Int.random(in: 1...100)
var sliderValueRounded: Int {
    Int(self.sliderValue.rounded())
}
```

Remove the data types from the first three properties so that the code looks like this:

```
@State var alertisVisible = false
@State var sliderValue = 50.0
@State var target = Int.random(in: 1...100)
var sliderValueRounded: Int {
    Int(self.sliderValue.rounded())
}
```

Note that Xcode doesn't display any error messages after this change.

- Build and run the app to confirm that it still works, even though you've removed the information about those properties' data types.

Why does the code still work? It's because Swift is smart enough to deduce, or as we say in programming, **infer**, the type of a variable or constant based the value you assign to it.

Consider the first property:

```
@State var alert isVisible = false
```

By assigning the value `false` to `alertViewIsVisible`, Swift infers that `alertViewIsVisible`'s data type is `Bool`.

Here's the next property:

```
@State var sliderValue = 50.0
```

Assigning `sliderValue` with the value **50.0** — a number with a decimal point — causes Swift to infer that the variable's data type is `Double`, the preferred data type for numbers with decimal points.

And then comes this property:

```
@State var target = Int.random(in: 1...100)
```

`Int.random(in: 1...100)` is a method that returns a random integer between 1 and 100 inclusive. By assigning it to `target`, Swift infers that `target` is an `Int` property.

Whenever possible, you should let Swift infer the type of variables and constants. There *are* times when Swift can't do that, though. Here's an example, taken straight from `pointsForCurrentRound()`

```
let difference: Int
if self.sliderValueRounded > self.target {
    difference = self.sliderValueRounded - self.target
} else if self.target > self.sliderValueRounded {
    difference = self.target - self.sliderValueRounded
} else {
    difference = 0
}
return 100 - difference
```

`difference` isn't assigned a value until after it's declared, which means that Swift doesn't have anything to use to infer `difference`'s type. In circumstances like this,

Swift *has* to be told what data type difference is. Swift also has to be told the data type of computed properties. If you try to change the declaration of `sliderValueRounded` to the following...

```
var sliderValueRounded: {  
    Int(self.sliderValue.rounded())  
}
```

...Xcode will complain quickly by throwing a very short error message: “Expected type”. This is where computed properties are more like methods; they can get so complex that Swift can’t infer their data type.

Key points

In this chapter, you added the following features to your app:

- The game now displays the target value at the top of the screen.
- It also calculates the points that the player earned based on the difference between the slider and the target values, then displays those points.

You also learned about:

- Generating pseudo-random numbers
- Algorithms
- Writing your own methods
- Making decisions with the `if` statement
- Adding strings using concatenation and representing special characters with escape sequences.
- Refactoring
- The `let` statement and constants
- The scope of variables and constants
- Computed properties

You have the basic elements of a working game, but it can only play a single round right now. In the next chapter, you’ll make the game fully functional with rounds and scorekeeping. You can find the project files for the app up to this point under **04 - Outlets** in the **Source Code** folder.



Chapter 5: A Fully Working Game

Joey deVilla

You've made a lot of progress on the game, and the to-do list is getting shorter! You have a basic version of the game running, where you can generate and display the target value, and you can also calculate and show the player the number of points they've scored in the current round.

It's now time to make a fully-working game, where the player can play multiple rounds and the game keeps a running score. We'll also give the player the ability to start a new game.

This chapter covers the following:

- **Improving the `pointsForCurrentRound()` algorithm:** Simplifying how the the number of points awarded to the player is calculated.
- **What's the score?:** Calculate the player's total score over multiple rounds and display it onscreen.
- **One more round...:** Implement updating the round count and displaying the current round on screen.
- **Key points:** A quick review of what you learned in this chapter.



Improving the pointsForCurrentRound() algorithm

Let's do a little more refactoring of `pointsForCurrentRound()`, the method that calculates how many points to award to the player based on the difference between the target value and where they put the slider. Here's its code at the moment:

```
func pointsForCurrentRound() -> Int {
    let difference: Int
    if self.sliderValueRounded > self.target {
        difference = self.sliderValueRounded - self.target
    } else if self.target > self.sliderValueRounded {
        difference = self.target - self.sliderValueRounded
    } else {
        difference = 0
    }
    return 100 - difference
}
```

Most of the code in this method is devoted to making sure that `difference` — the difference between the slider value and the target value — is always positive. This is done by making sure that the smaller value is always subtracted from the larger value.

“Absolute” power

There's a simpler way to do this, and it comes from one of the many math functions built into the Swift Standard Library: The `abs()` function. Given a number, which can be an `Int`, a `Double` or any other Swift data type that represents a number, it returns the *absolute value* of that number, which is the value of that number, but ignoring the sign.

Here are some examples of `abs()` in action:

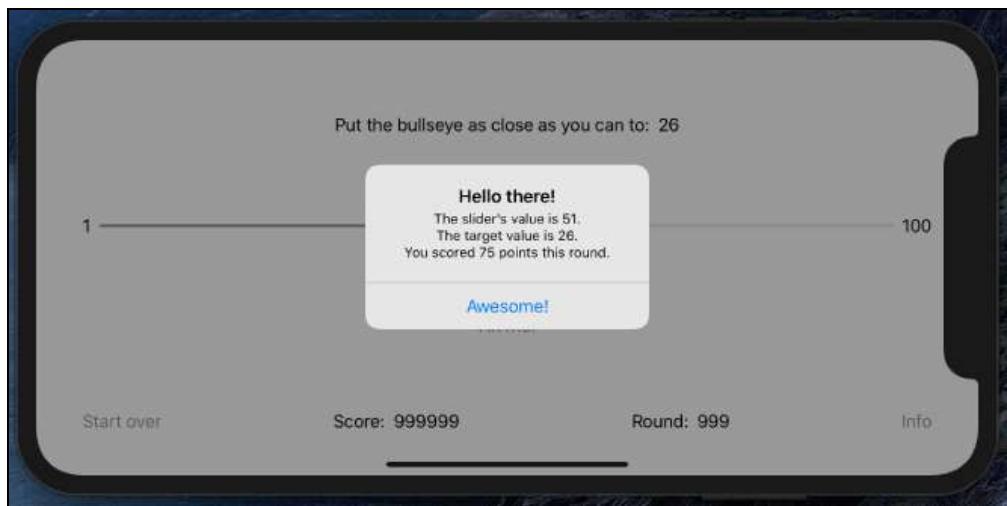
- `abs(5)` returns 5
- `abs(-5)` returns 5
- `abs(-5.25)` returns 5.25

- Let's use `abs()` to simplify the code in `pointsForCurrentRound()`. Change its code to the following:

```
func pointsForCurrentRound() -> Int {  
    let difference = abs(self.sliderValueRounded - self.target)  
    return 100 - difference  
}
```

Note that you didn't have to specify `difference`'s data type. That's because Swift can infer it from the code on the right side of the `=` sign: `self.sliderValueRounded` and `self.sliderValueRounded` are both `Ints`, subtracting the latter from the former also yields an `Int` and the absolute value of that result is also an `Int`. Based on this, Swift infers that `difference` is an `Int`.

- Run the app and click **Hit me!**. It works as before, without any changes that the player will notice, but with much less code:



Good code is simple and readable, and this often translates to less code. If you can get the same result using less code, you get not only the benefits of simplicity and readability, but fewer lines of code makes it less likely to introduce bugs.

Removing a “magic number”

Here's the current code for `pointsForCurrentRound()`:

```
func pointsForCurrentRound() -> Int {
    let difference = abs(self.sliderValueRounded - self.target)
    return 100 - difference
}
```

If you've worked on the code recently, it's probably quite obvious to you what the **100** in `return 100 - difference` is for. It's the maximum possible score, which happens when the player positions the slider right at the target value.

However, if you spend some time away from *Bullseye*'s code and then return to it, you might have forgotten where the **100** comes from. You might also decide to change this value at a later point.

There's a programming term for numbers like this that appear in code: *magic numbers*. They're called magic because they're just there, without any explanation or context; they just “magically” appear in the code. In programming, we strongly discourage the use of magic numbers, and recommend that you replace them with a constant with a name that explains what the number is for.

► Let's define a new constant, `maximumScore`, to replace the magic number. Change the code for `pointsForCurrentRound()` to this:

```
func pointsForCurrentRound() -> Int {
    let maximumScore = 100
    let difference = abs(self.sliderValueRounded - self.target)
    return maximumScore - difference
}
```

Once again, you don't have to specify `maximumScore`'s data type. Based on the value of **100** assigned to it, Swift will infer that `maximumScore` is an `Int`.

► Run the app. Once again, it works as it did before the change.

You've replaced a number without context — **100** — with the constant `maximumScore`, which both holds the value 100 *and* explains what it's for. Even with this additional line, you still have a `pointsForCurrentRound()` that's less than a third the size of the original.

What's the score?

Now that you have a lean, mean `pointsForCurrentRound()` and know how far off the slider is from the target, it's time to keep track of the player's score.

The first thing you'll need is a place to store the score. Think about the nature of the score:

- It should have a name that makes its use and purpose clear: `score`.
- It's a whole-number value. This means that it should be an `Int`. It should have an initial value of `0`.
- It's part of the state of the game. Thus means that it should be marked with the `@State` keyword.

► Add the new variable to the *User interface views* section of `ContentView`'s properties, just below the declaration for `sliderValueRounded`. The section should look like this at the end:

```
// User interface views
@State var alertVisible = false
@State var sliderValue = 50.0
@State var target = Int.random(in: 1...100)
var sliderValueRounded: Int {
    Int(self.sliderValue.rounded())
}
@State var score = 0
```

Now that there's a `score` variable, there needs to be code to add the points that the player earned to it. The player earns points when they tap the **Hit me!** button, so that seems like a logical place to calculate the total score.

The code for the **Hit me!** button is in the body variable, in the *Button row* section:

```
// Button row
Button(action: {
    print("Points awarded: \(self.pointsForCurrentRound())")
    self.alertVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertVisible) {
    Alert(title: Text("Hello there!"),
          message: Text(self.scoringMessage()),
          dismissButton: .default(Text("Awesome!")))
}
```

The code in the Button view's `action:` parameter is executed whenever it's tapped. Right now, that code is:

```
print("Points awarded: \(self.pointsForCurrentRound())")  
self.alertVisible = true
```

This code does the following:

- It outputs the points that the player has earned for the current attempt on Xcode's console. This only happens with the Simulator, or on a connected device that's running the app from Xcode, and will only be seen by the programmer. The user never sees this.
- It sets the `alertVisible` property to `true`, which causes the alert pop-up to appear.

Let's add to this code. We should add the points that are being awarded to the player for this round to the total score.

► Change the code for the Button view so that it looks like the following:

```
// Button row  
Button(action: {  
    print("Points awarded: \(self.pointsForCurrentRound())")  
    self.alertVisible = true  
    self.score = self.score + self.pointsForCurrentRound()  
}) {  
    Text("Hit me!")  
}  
.alert(isPresented: self.$alertVisible) {  
    Alert(title: Text("Hello there!"),  
          message: Text(self.scoringMessage()),  
          dismissButton: .default(Text("Awesome!")))  
}
```

There's now a place in which to store the score, and there's a way to add to the score when the player taps **Hit me!**. It's now time to display the score.

Since `score` is a `@State` variable, it means that Swift constantly watches it for changes, and immediately updates any user interface elements that make use of it when those changes happen. Let's set up that user interface element.

► Scroll down to the part of the body variable marked *Score row* and change it to the following:

```
// Score row  
HStack {  
    Button(action: {}) {
```

```
    Text("Start over")
}
Spacer()
Text("Score:")
Text("\(self.score)")
Spacer()
Text("Round:")
Text("999")
Spacer()
Button(action: {}) {
    Text("Info")
}
.padding(.bottom, 20)
```

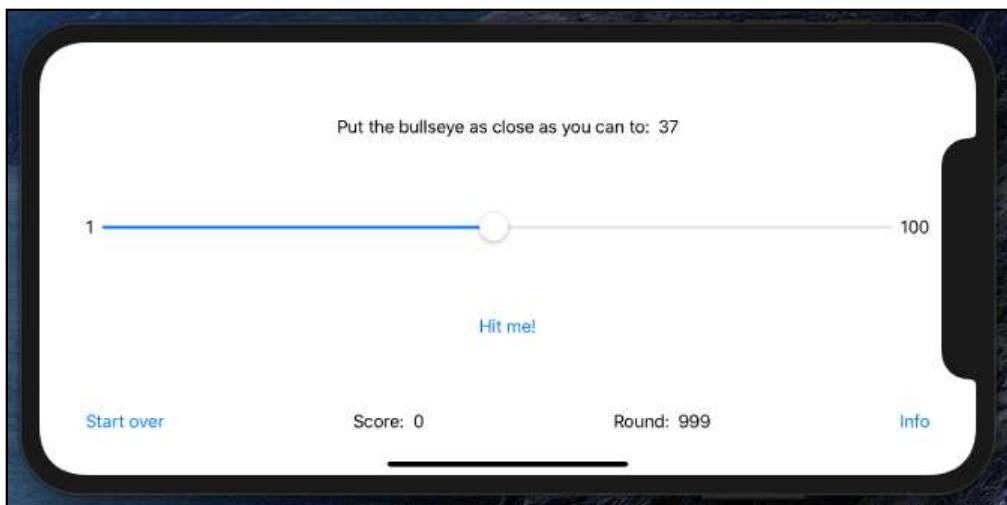
Note the change. You've replaced this hard-coded score:

```
Text("999999")
```

With:

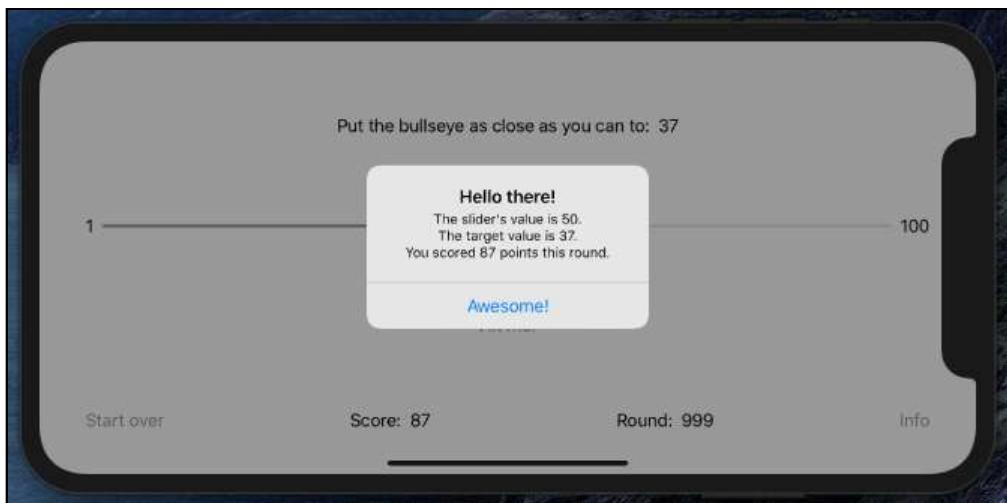
```
Text("\(self.score)")
```

- Run the app. When it starts, you'll see something like this:



Note that the player's score is no longer 999999, but 0, which is the initial value assigned to `score`. The Text view now displays the score, which is always up to date because `score` is a @State variable.

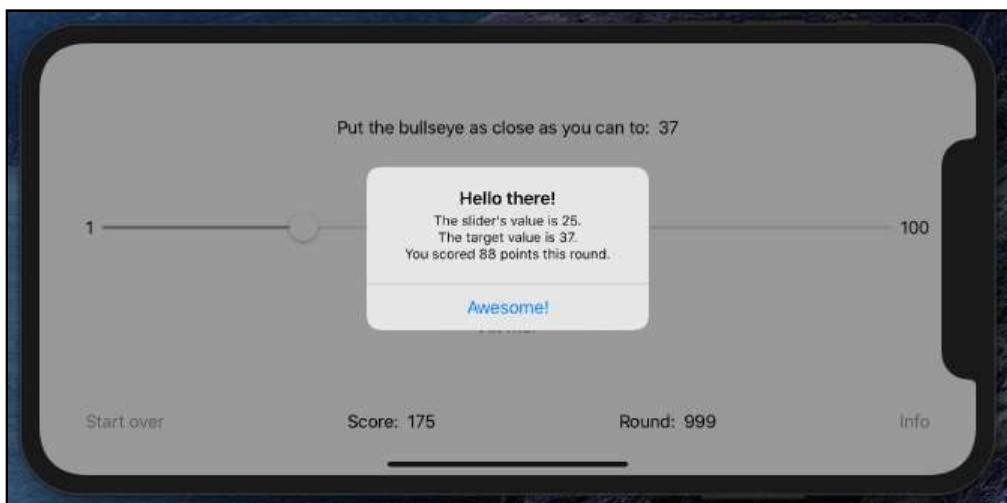
- Click **Hit me!** The pop-up will appear:



The pop-up acts as you'd expect, displaying the slider's value, the target value, and the number of points the player scored.

What's new is the score. If you look at the bottom of the screen, you'll see that the score has been updated, with the player's points have been added to it.

- Click **Awesome!** The pop-up will be dismissed, and you'll have another chance to position the slider. Move the slider, and click **Hit me!** again:



Once again, the pop-up appears, along with value, the target value, and the number of points the player scored. And once again, if you look at the bottom of the screen, you'll see that the score has been updated.

There's just one problem now: The target never changes. We'll fix that by starting a new round.

One more round...

Once the player has tapped **Hit me!** and been awarded their points, the game should present the player with a new target. This means coming up with a new random value for target. That part is easy:

```
self.target = Int.random(in: 1...100)
```

The trickier part is figuring out *where* to put this code.

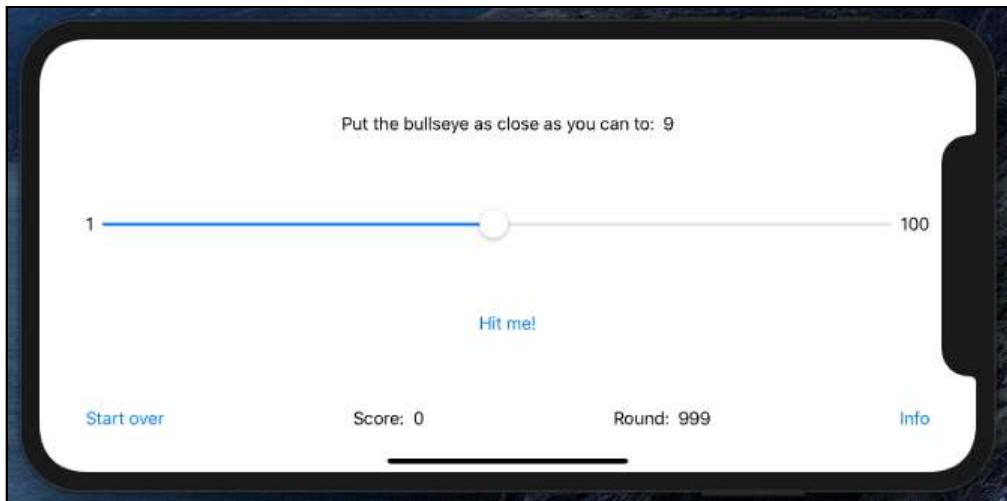
The most obvious place is inside the `action:` parameter of the `Button` view. Right now, code in this parameter causes the pop-up to appear and updates the score. It looks like a good place to generate a new target value.

► Change the code in body at the start of the *Button row* section so that it looks like the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertIsVisible = true
    self.score = self.score + self.pointsForCurrentRound()
    self.target = Int.random(in: 1...100)
}) {
```

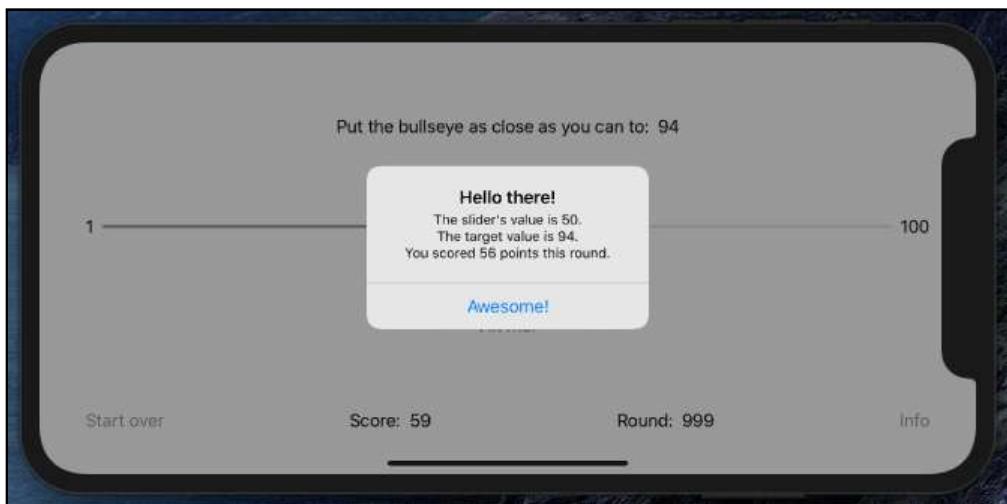


- Run the app and make a note of the target value — don't do anything else just yet:



In the example above, the target value is 9.

- Click **Hit me!**. 99 times out of 100, you'll see that the target value has changed!



The target value was 9 before you clicked **Hit me!**, and changed — both on the main screen and in the pop-up — to 94.

The player was awarded 56 points, and if you do the math:

- Take the maximum score of **100**,
- the new target value of **94** and the slider position of **50**, making a difference of **44**,
- which makes for **56** points, which comes from $100 - 44$.

How did this happen?

Asynchronous code execution

You probably know that computers — your iOS device included — can perform several tasks at the same time, either by actually performing tasks simultaneously, or combining careful scheduling with their millisecond speed to make it appear as if they're multitasking.

Let's look at the Button code again:

```
// Button row
Button(action: {
    print("Points awarded: \(self.pointsForCurrentRound())")
    self.alertVisible = true
    self.score = self.score + self.pointsForCurrentRound()
    self.target = Int.random(in: 1...100)
}) {
```

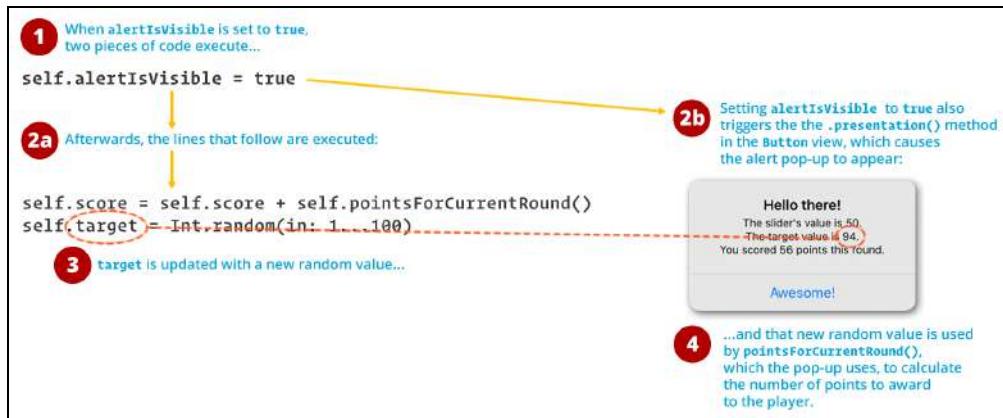
The multitasking starts when `alertVisible` is set to `true`. The program follows two paths:

1. The program continues to execute the rest of the code in Button's action parameter.
2. Setting the state variable `alertVisible` to `true` triggers Button's `.presentation()` method and causes the alert pop-up to appear.

These two paths of execution happen over a span of milliseconds, and practically simultaneously.



The diagram below might make it easier to understand what's happening:



The series of events that causes the alert pop-up to appear happens right away, but the code in the `action:` parameter has already updated the score and generated a new target number before the alert pop-up has even finished drawing itself onscreen.

In programmer-speak, alerts work *asynchronously*. We'll talk much more about that in a later chapter, but it means that you should keep in mind that a lot of code that updates the user interface based on changes to state variables often gets executed at the same time. Clearly, we'll need to take another approach.

- Change the code in body at the start of the `Button row` section so that the only code in the button's `action:` parameter sets `alertIsVisible` to `true`. The section should end up like this:

```
// Button row
Button(action: {
    self.alertIsVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text("Hello there!"),
        message: Text(self.scoringMessage()),
        dismissButton: .default(Text("Awesome!")))
}
```

Finding a better place to start a new round

The problem with our first approach is that it tried to start a new round in response to the player tapping **Hit me!**, which is *before* the alert pop-up gets displayed. The new round should start in response to the player dismissing the alert pop-up, which happens when they tap the pop-up's **Awesome!** button.

It turns out that the `Alert` object, which is initialized at the end of the button row section of code...

```
Alert(title: Text("Hello there!"),
      message: Text(self.scoringMessage()),
      dismissButton: .default(Text("Awesome!")))
```

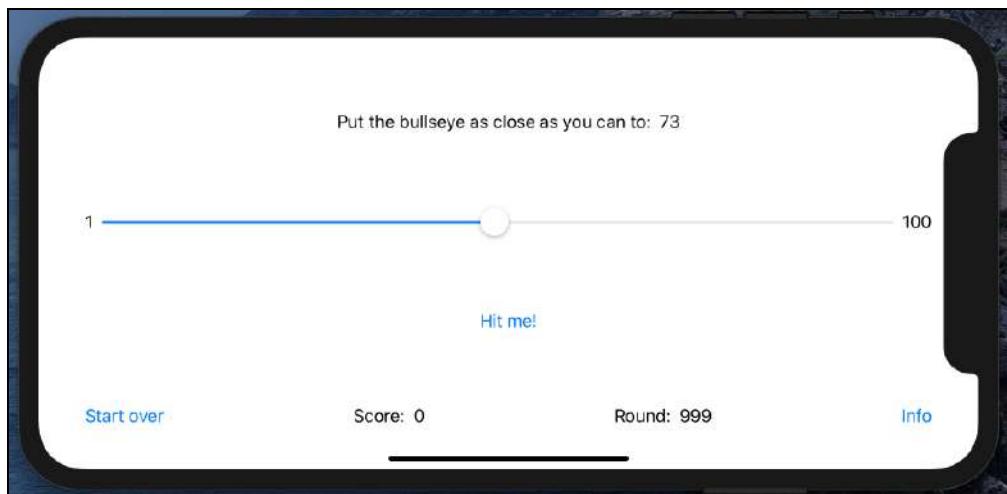
...accepts an optional extra parameter *after* `dismissButton:`, and that parameter is code to be executed when the alert pop-up is dismissed.

► Change the code in body's *Button row* section so that it looks like this:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertIsVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text("Hello there!"),
          message: Text(self.scoringMessage()),
          dismissButton: .default(Text("Awesome!")) {
              self.score = self.score + self.pointsForCurrentRound()
              self.target = Int.random(in: 1...100)
          }
    )
}
```

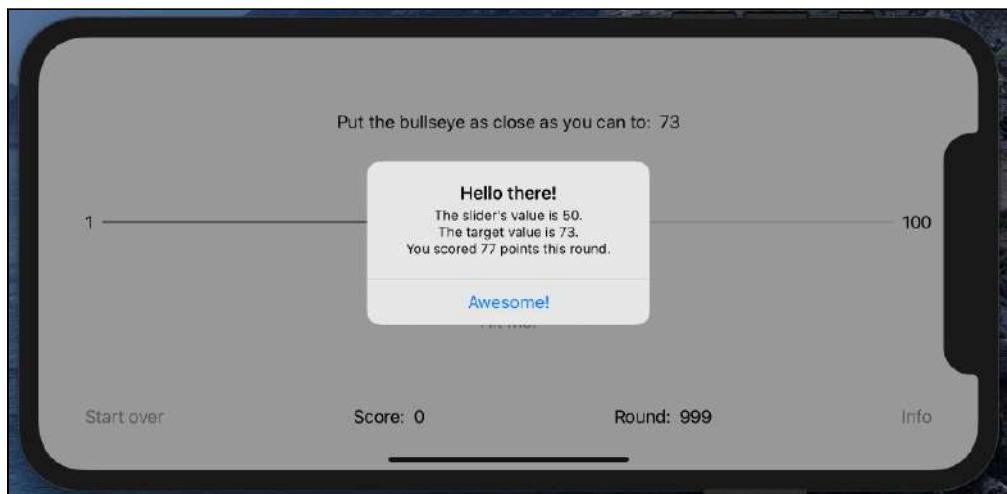
Note: You may have noticed that objects and methods seem to expect some of their parameters inside parentheses `()` and some of them inside braces `{ }`. We'll explain why this is so later in this book, and it will all make sense. For now, just trust in the code that we're showing you.

- Run the app, and, once again, make a note of the target value first:



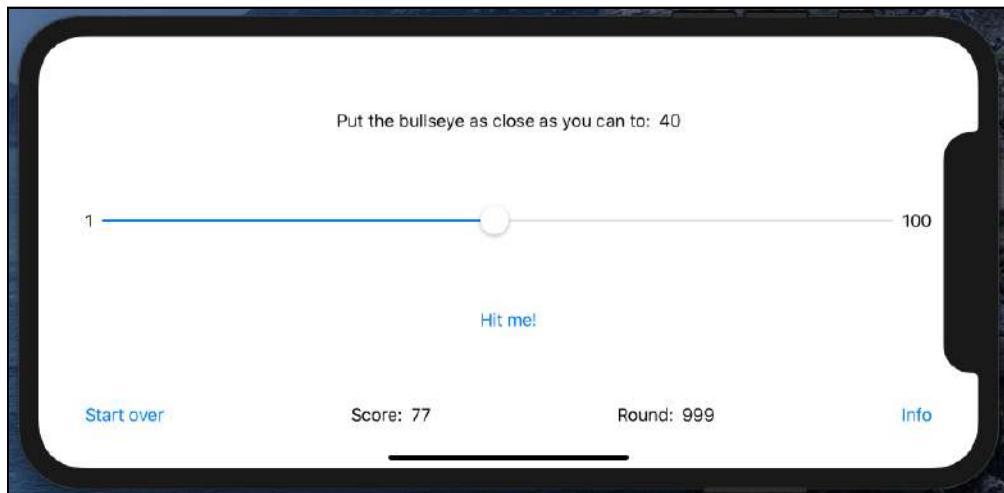
In the example above, the target value is 73.

- Click **Hit me!**. This time, you'll see that the target value is still 73, and that the points earned this round are based on that value:



You should also note that the score hasn't been updated yet — it's still 0.

- Dismiss the pop-up by pressing **Awesome!**, and make a note of the target value and score:



The target value is new, and the score has been updated. Now it's time to properly display the current round.

Showing the current round

Just as there's a designated place to store the score, there also needs to be a place to store the number of the current round.

Think about what this variable should be like:

- It should have a name that makes its use and purpose clear: `round`.
- It's a whole-number value. This means that it should be an `Int`. It should have an initial value of `1`.
- It's part of the state of the game. Thus means that it should be marked with the `@State` keyword.

- Add the new variable to the *User interface views* section of `ContentView`'s properties. The section should look like this at the end:

```
// User interface views
@State var alertIsVisible = false
@State var sliderValue = 50.0
@State var target = Int.random(in: 1...100)
var sliderValueRounded: Int {
    Int(self.sliderValue.rounded())
```

```
}
```

```
@State var score = 0
```

```
@State var round = 1
```

Now that we have the `round` variable, we need code to increase its value by 1 — or in programmer-speak; *increment* it — at the start of a new round. A new round starts when the player dismisses the alert pop-up, so that's where this code should go.

The code for the **Hit me!** button is in the body variable, in the button row section:

- Change the code at the end of bodys *Button row* section so that it looks like this:

```
Alert(title: Text("Hello there!"),
      message: Text(scoringMessage()),
      dismissButton: .default(Text("Awesome!")) {
        self.score = self.score + self.pointsForCurrentRound()
        self.target = Int.random(in: 1...100)
        self.round = self.round + 1
    }
)
```

There's now a place in which to store the number of the current round, and that number is incremented when the player dismissed the alert pop-up. It's now time to display it.

As with `score`, `round` is a state variable, it means that Swift constantly watches it for changes, and changes cause any user interface elements that make use of it to be updated. Let's set up that user interface element.

- Scroll to the part of the body variable marked *Score row* and change it to the following:

```
// Score row
HStack {
    Button(action: {}) {
        Text("Start over")
    }
    Spacer()
    Text("Score:")
    Text("\(self.score)")
    Spacer()
    Text("Round:")
    Text("\(self.round)")
    Spacer()
    Button(action: {}) {
        Text("Info")
    }
}
.padding(.bottom, 20)
```



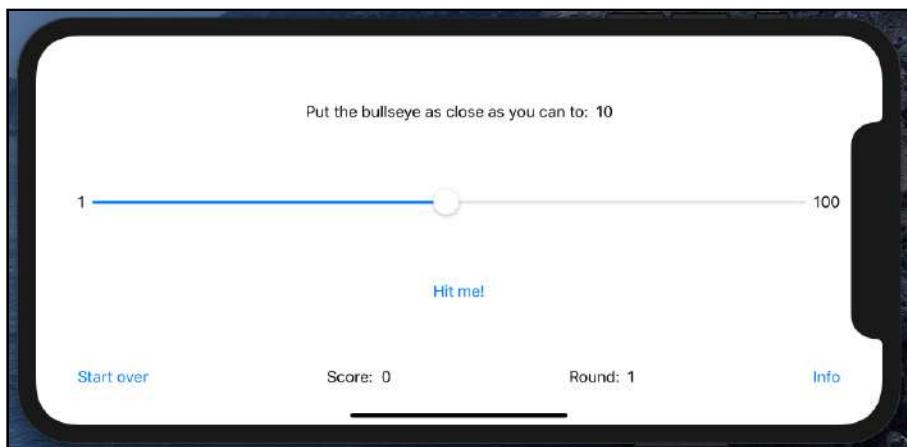
Note the change. You've replaced this hard-coded count of rounds:

```
Text("999")
```

With:

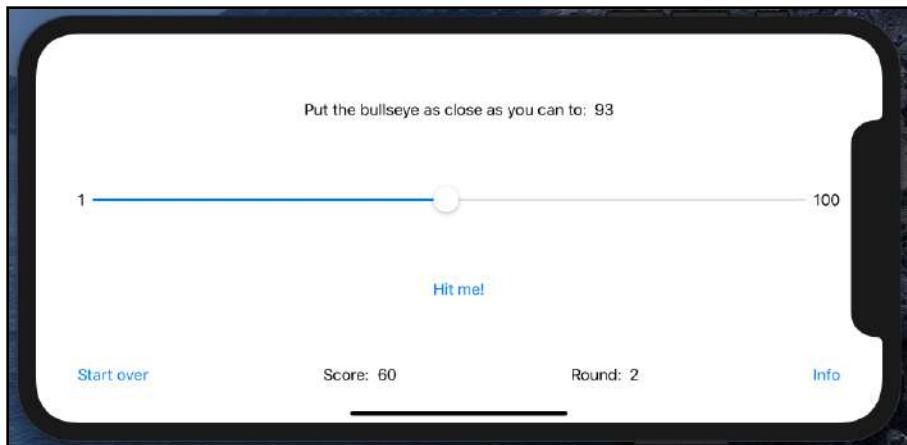
```
Text("\(self.round)")
```

- Run the app. When it starts, you'll see something like this:



Note that the starting round is no longer 999, but 1, which is the initial value assigned to round. The Text view now displays the correct round, which is always up to date because round is a @State variable.

- Click **Hit me!**, and then dismiss the alert pop-up when it appears. You'll see something like this:



The screen shows that you’re now on round 2.

All the display elements work now!

Key points

You’ve got a mostly-working game; feel free to take a victory lap.

In this chapter, you did the following:

- You improved `pointsForCurrentRound()`’s algorithm by using the `abs()` function from the Swift Standard Library, reducing the number of lines in the method by more than two-thirds.
- You also made `pointsForCurrentRound()` more readable by replacing a “magic number” with a constant.
- Added the ability to store and display the cumulative score and current round.

In the next chapter, you’ll do some more refactoring, tweak the game to improve it, and give the player the ability to start a new game.

You can find the project files for the app up to this point under **05 — Rounds and Score** in the Source Code folder. If you get stuck, compare your version of the app with these source files to see if you missed anything.

6 Chapter 6: Refactoring

Joey deVilla

At this point, your game is fully playable. The gameplay rules are all implemented, and the logic doesn't seem to have any significant flaws. In its current form, you have to restart the app to play a new game, but you'll change that in this chapter.

As far as we can tell, there aren't any bugs. That said, there's still some room for improvement!

This chapter will cover the following:

- **Improvements:** Small UI tweaks to make the game look and function better.
- **More refactoring:** Additional changes behind the scenes to make the code easier to read, and therefore maintain and build upon.
- **Starting over:** Resetting the game to start fresh.
- **Making the code less self-ish:** The keyword `self` is used all over the code. Are they all necessary?
- **Key points:** A quick review of what you learned in this chapter.



Improvements

While the game isn't very pretty yet — and don't worry, you'll fix that in the next chapter — there are still a couple of tweaks that you can make to improve its user experience.

The alert title

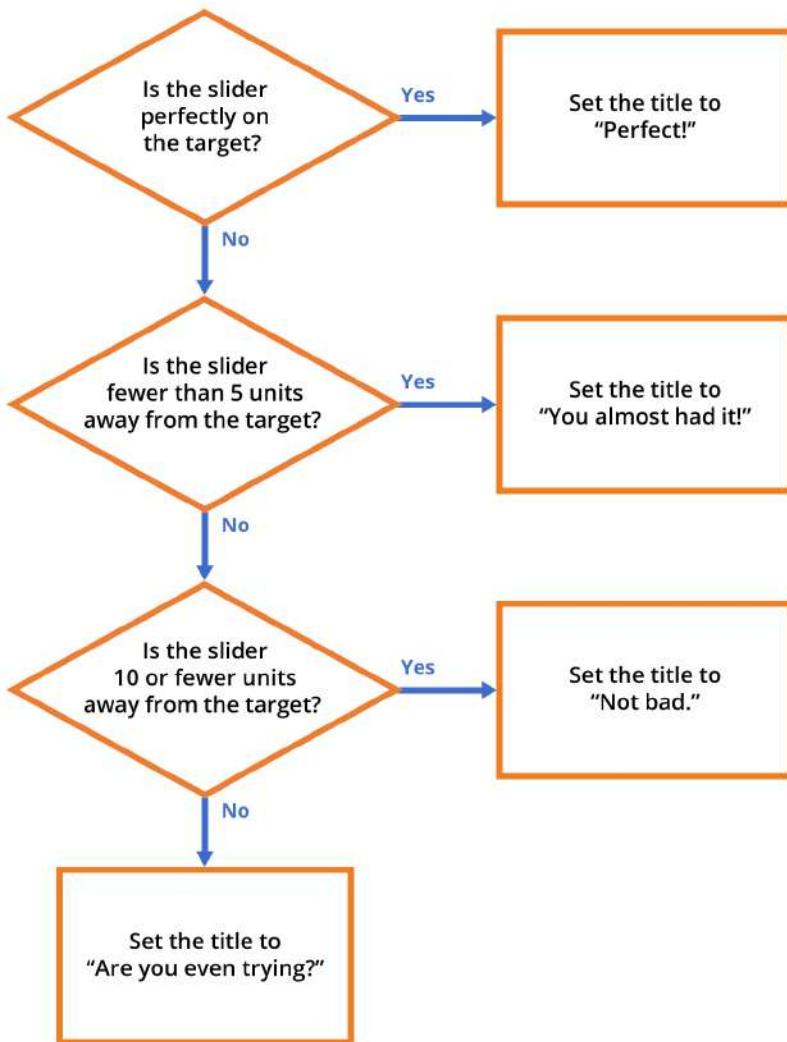
Unless you've already changed it, the title of the alert pop-up still says "Hello there!". That's something leftover from back when it was a single-button app. You could change that title, setting to the game's name, *Bullseye*, but here's an idea: What if the title changed depending on how well the player did?

Here are the details:

- If the player is either really lucky or has a very good eye and puts the slider right on the target, the alert's title could say "Perfect!"
- If the player didn't put the slider right on the target but got really close to the target but not quite there, say under five units away, the alert's title could say "You almost had it!"
- A close-ish attempt, say 10 or fewer units away, could be rewarded with the title "Not bad."
- In all other cases, the alert gives the player a little tough love with "Are you even trying?"



Here's a flowchart that illustrates the process:



Flowchart illustrating how the alert title is determined

Exercise: Think of a way to accomplish this. How would you program it? Hint: There are an awful lot of "if's" in the preceding sentences.

Before we write the code to set the alert pop-up's title based on how close the slider and target are, let's figure out where this code should go.

Let's look at the code that defines the alert pop-up:

```
Alert(title: Text("Hello there!"),
      message: Text(self.scoringMessage()),
      dismissButton: .default(Text("Awesome!")) {
        self.score = self.score + self.pointsForCurrentRound()
        self.target = Int.random(in: 1...100)
        self.round = self.round + 1
    }
)
```

The title of the alert pop-up is defined by the `Alert`'s `message:` parameter. Right now, the `Text` view that it's filled with contains the string "Hello there!". We need a way to change this string based on how well the player did in the current round.

In the previous chapter, we wrote the `scoringMessage()` method to generate the string that forms the main message of the alert pop-up:

```
func scoringMessage() -> String {
    return "The slider's value is \(self.sliderValueRounded).\n" +
        "The target value is \(self.target).\n" +
        "You scored \(self.pointsForCurrentRound()) points this
round."
```

We'll write a similar method to generate the string that forms the title of the alert pop-up. Like the `scoringMessage()` method, this new method returns a value and therefore should have a name that describes the value it returns. We'll call it `alertTitle()`.

► Add the following to the end of `ContentView`'s *Methods* section, just after `scoringMessage()`:

```
func alertTitle() -> String {
    let difference: Int = abs(self.sliderValueRounded -
self.target)
    let title: String
    if difference == 0 {
        title = "Perfect!"
    } else if difference < 5 {
        title = "You almost had it!"
    } else if difference <= 10 {
        title = "Not bad."
    } else {
        title = "Are you even trying?"
    }
    return title
}
```

This code first calculates `difference`, the difference between the slider position and the target. It then creates a string constant named `title`, which will contain the title that this method returns. As you can see, most of this method is taken up by an `if` statement that runs through different possibilities to select a title.

The `if` statement has four different *clauses*. Here's the first one:

```
if difference == 0 {  
    title = "Perfect!"
```

It says if the difference between the slider and the target is 0, set the value of `title` to **Perfect!**.

You're probably wondering what the `==` means. There's a critical difference between it — two “equals” signs in a row — and a single “equals” sign:

- The single equals sign, `=`, means “Set this variable or constant to a given value.” To take an example from the code above, `title = "Perfect!"` means “Set `title` to **Perfect!**.”
- The double equals sign, `==`, means “is equal to.” To take an example from the code above, `difference == 0` means “`difference` is equal to **0**”, a statement that is either `true` or `false`.

When you start programming, you may find yourself using `=` in when you should really be using `==`. Watch for this, especially when Xcode gives you an error message related to an `if` statement.

Here's the `if` statement's second clause:

```
} else if difference < 5 {  
    title = "You almost had it!"
```

If the first clause doesn't apply, and `difference` is less than 5, `title` is set to **You almost had it!**.

The `if` statement reads as follows:

```
} else if difference <= 10 {  
    title = "Not bad."
```

If the first and second clause don't apply, and `difference` is 10 or less, `title` is set to **Not bad**.

At the end of the if statement comes the else clause:

```
} else {  
    title = "Are you even trying?"
```

This handles the case when none of the other clauses apply. In this case, title is set to **Are you even trying?**.

Now that we have the alertTitle() method, it's time to make use of it.

- Change the code in body that defines the alert pop-up to the following:

```
Alert(title: Text(alertTitle()),  
      message: Text(scoringMessage()),  
      dismissButton: .default(Text("Awesome!")) {  
        self.score = self.score + self.pointsForCurrentRound()  
        self.target = Int.random(in: 1...100)  
        self.round = self.round + 1  
    }
```

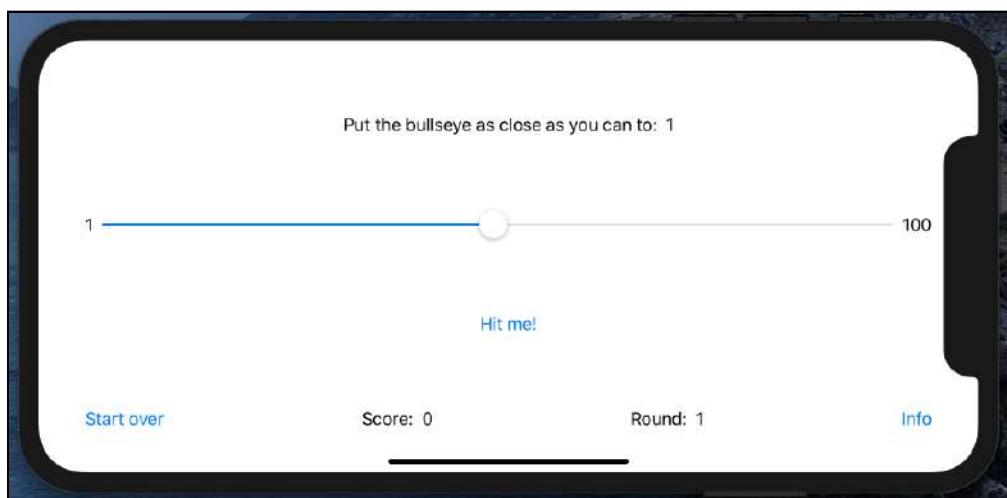
Note the change. You've replaced this hard-coded message:

```
Text("Hi there!")
```

With:

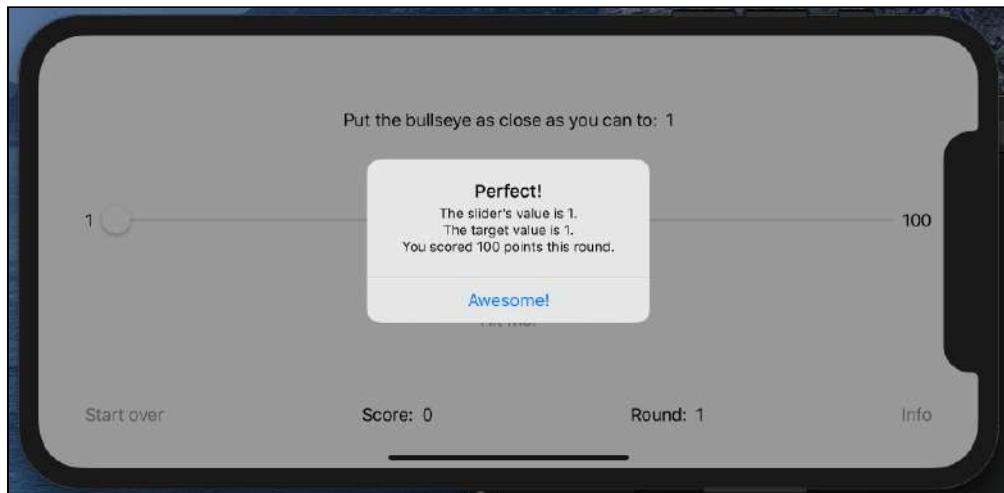
```
Text(scoringMessage())
```

- Run the app. When it starts, you'll see something like this:



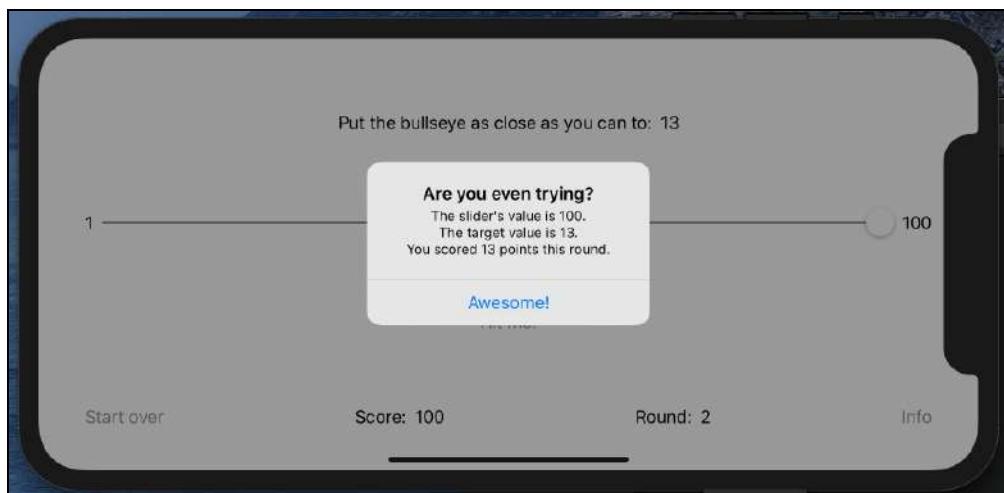
The game, with a target value of 1

The example is a really lucky one because a target of 1 is very easy to hit — you move the slider all the way to the left. Here's what the alert pop-up looked like after doing so:



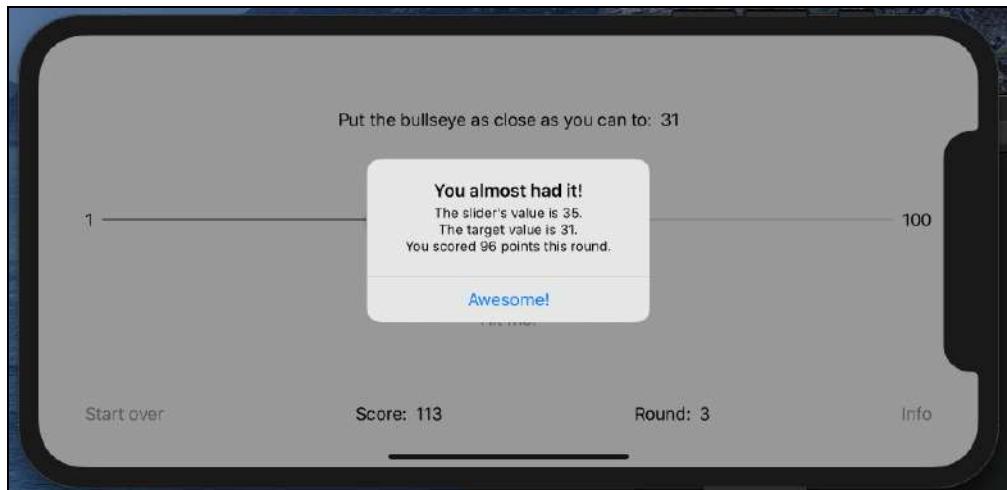
The alert pop-up, where the slider is right on the target, with the title 'Perfect!'

Here's what the alert looks like if **Hit me!** is pressed when the slider is positioned far from the target:



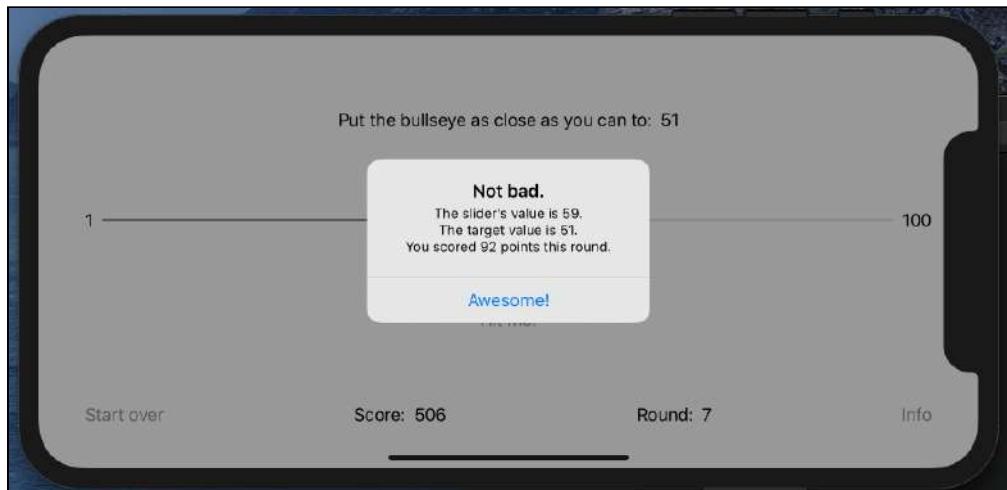
The alert pop-up, where the slider is way off the target, with the title 'Are you even trying?'

Here's the alert when the slider is fewer than 5 units away from the target:



The alert pop-up, where the slider is 4 units away from the target, with the title 'You almost had it!'

And finally, if the player presses **Hit me!** after putting the slider a “not bad” distance (between 5 and 10 units) from the target, they see this:



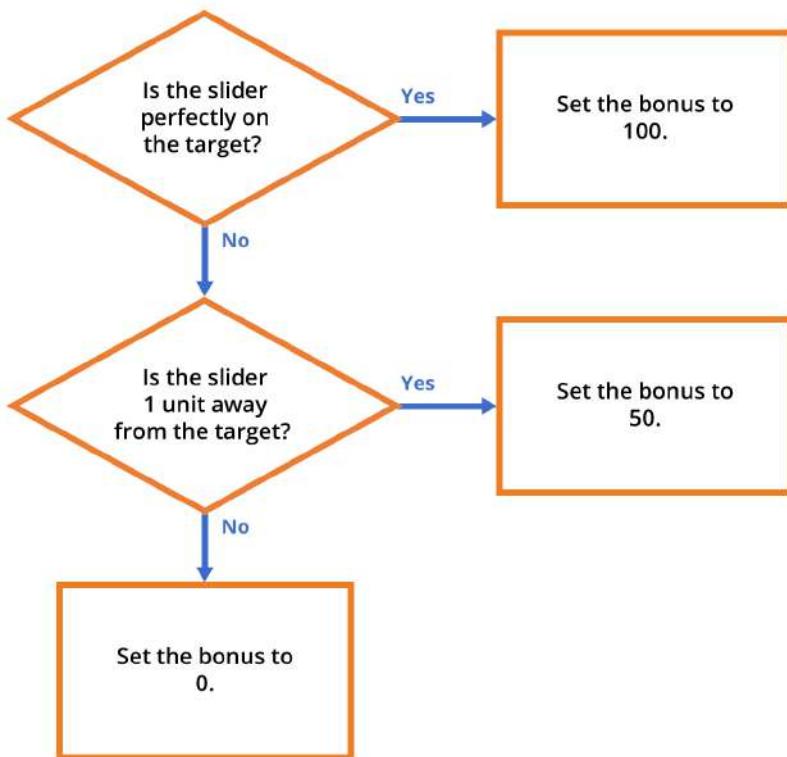
The alert pop-up, where the slider is 8 units away from the target, with the title 'Not bad.'

Bonus points

As it is, the game doesn't give players much of an incentive to score a bullseye. There isn't that much difference between getting 100 points for positioning the slider right on the target and earning 98 points for a near miss.

What if the game awarded bonuses for accuracy — say, 100 points for a bullseye and 50 points for being off by one?

Here's a flowchart showing the bonus process:



Flowchart illustrating how the bonus is determined

Exercise: Before you continue reading and look at the code that follows, think about how you'd implement this bonus rule.



The `pointsForCurrentRound()` method calculates how many points to award to the player, so it's probably where you should make the change.

► Update `pointsForCurrentRound()` to this:

```
func pointsForCurrentRound() -> Int {  
    let maximumScore: Int = 100  
    let difference = abs(self.sliderValueRounded - self.target)  
  
    let bonus: Int  
    if difference == 0 {  
        bonus = 100  
    } else if difference == 1 {  
        bonus = 50  
    } else {  
        bonus = 0  
    }  
  
    return maximumScore - difference + bonus  
}
```

Here's the code that determines if the player has earned a bonus:

```
let bonus: Int  
if difference == 0 {  
    bonus = 100  
} else if difference == 1 {  
    bonus = 50  
} else {  
    bonus = 0  
}
```

The `let bonus: Int` line declares a constant named `bonus`, where the calculated bonus will be stored.

The `if` statement that follows has three different *clauses*. Here's the first one:

```
if difference == 0 {  
    bonus = 100
```

It says that if the difference between the slider and the target is 0, set the value of `bonus` to 100.

Here's the `if` statement's second clause:

```
} else if difference == 1 {  
    bonus = 50
```

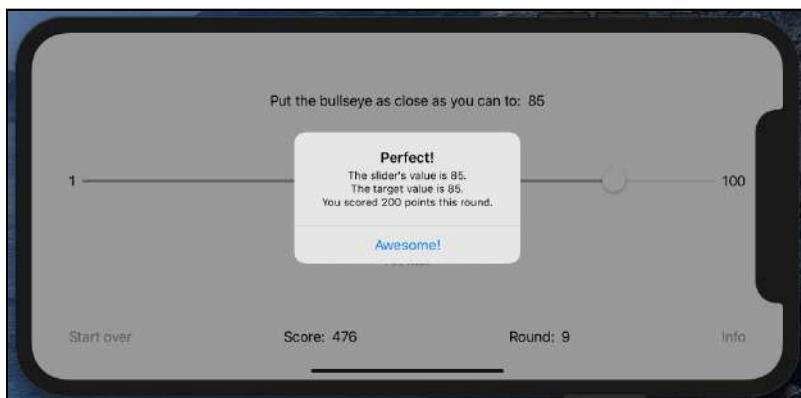
If the first clause doesn't apply, but this one does, bonus is set to 50. The clause applies only if difference is equal to 1.

The `else` clause at the end handles all the other cases not handled by the first or second clause:

```
} else {  
    bonus = 0  
}
```

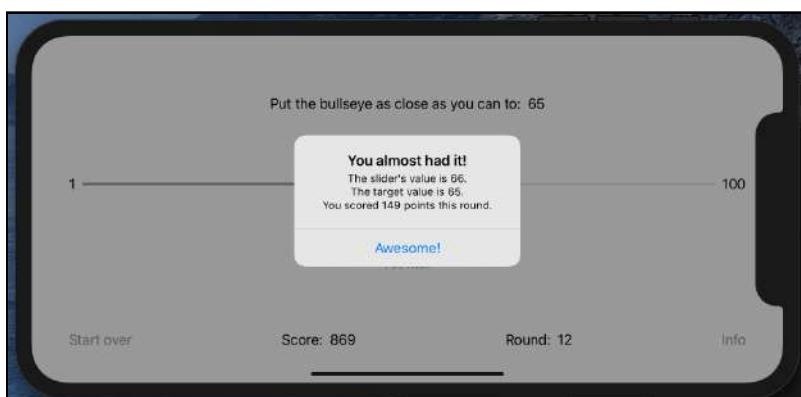
► Run the app to see if you can score some bonus points!

With the newly-added code for calculating a bonus score, here's what players will see when they position the slider perfectly at the target value:



The pop-up showing that the player got 200 points for positioning the slider perfectly

If they're off by one, they won't get the 100-point bonus, but they'll get the 50-point consolation bonus:



The pop-up showing that the player got 149 points for missing the target by one unit

More refactoring

Back in Chapter 4, I introduced you to the concept of refactoring. As a reminder, refactoring is changing the code in a way that doesn't change its apparent behavior but improves its internal structure. Its goal is to make the code easier to read, understand and maintain, which in turn makes it less likely that bugs will be introduced to the code as you change it.

Let's do some more refactoring now.

Refactoring the bonus algorithm

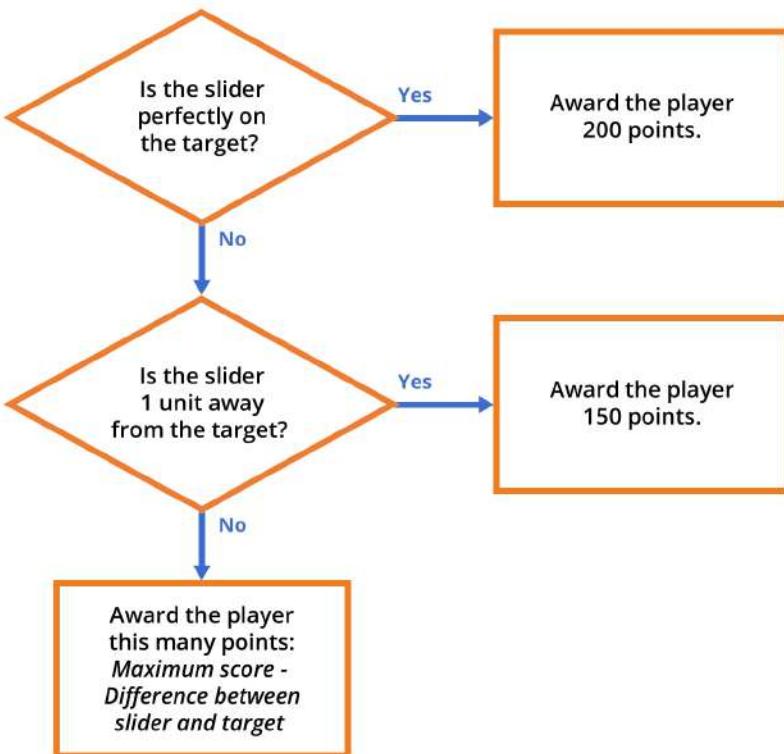
You may have noticed a couple of things about the bonus algorithm, either by looking carefully at its code or from playing the game and getting a perfect or off-by-one score:

- The player is always awarded 200 points for perfectly positioning the slider on the target value. This is because you can only get the 100-point bonus when you earn 100 points.
- If the player misses the target by one, they're always awarded 149 points. This comes from the fact that being off by one earns 99 points, and adding the 50-point bonus yields a not-quite-round 149 points.

Let's change the algorithm so that:

- If the slider value is equal to the target, give the player 200 points.
- If the slider value is not equal to the target but off by one, give the layer a nice round 150 points.
- If neither of the above applies, give the player the standard number of points, which is an arbitrary maximum value minus the difference between the slider value and the target value.

For those of you who like to think in pictures, here's the algorithm in flowchart form:



The pop-up showing that the player got 149 points for missing the target by one unit

- Let's implement this revised algorithm. Change `pointsForCurrentRound()` so that it looks like this:

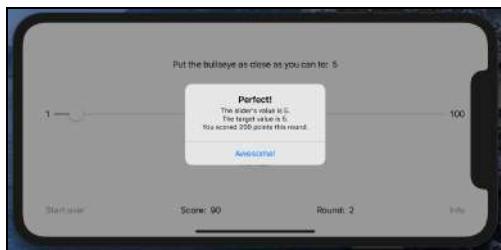
```

func pointsForCurrentRound() -> Int {
    let maximumScore: Int = 100
    let difference = abs(self.sliderValueRounded - self.target)

    let points: Int
    if difference == 0 {
        points = 200
    } else if difference == 1 {
        points = 150
    } else {
        points = maximumScore - difference
    }
    return points
}
  
```

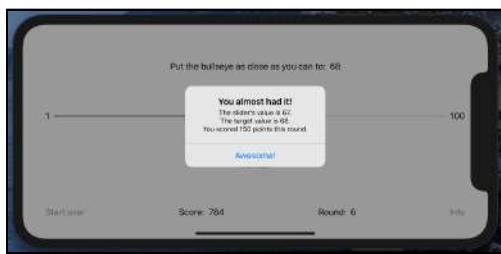


- Run the app and try to put the slider right on the target value. You'll see this message when you press **Hit me!**:



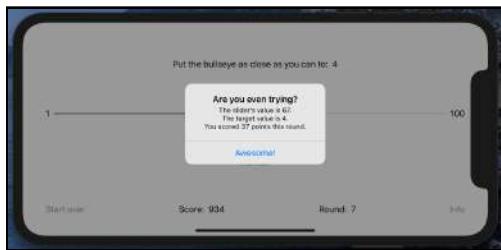
The pop-up showing 200 points for perfectly placing the slider

- Try to put the slider just one unit off the target. You'll see this:



The pop-up showing 150 points for being off by one unit

- Set the slider way off the target, and this is what you'll get:



The pop-up showing the score being calculated the usual way

DRYing up the code

The `pointsForCurrentRound()` method calculates the number of points to award to the user by looking at the difference between the slider's value and the target. It does so with this line of code:

```
let difference: Int = abs(self.sliderValueRounded - self.target)
```

The `alertTitle()` method determines the title that appears in the alert pop-up based on the difference between the slider's value and the target. Here's the line of code that used to do this, and it should give you a sense of *déjà vu*:

```
let difference: Int = abs(self.sliderValueRounded - self.target)
```

This code isn't DRY — Don't Repeat Yourself — because we're repeating ourselves! You might even say that it's WET — Write Everything Twice!

If you decide to change the way that the difference between the slider value and the target is calculated, you'll have to change it in *two* places. This increases the likelihood that you might forget to update one of those places or introduce an error by unintentionally varying how the difference is calculated.

Let's remove this redundancy by calculating the difference between the slider and target in just once place. This can be done with a method or a computed property. Since the difference is a simple calculation based on two properties — `sliderValueRounded` and `target` — it makes sense to use a computed property.

- Add the following computed property to the end of the *User interface views* part of `ContentView`'s *Properties* section:

```
var sliderTargetDifference: Int {  
    abs(self.sliderValueRounded - self.target)  
}
```

- Now that you have this method, use it in `pointsForCurrentRound()`...

```
func pointsForCurrentRound() -> Int {  
    let maximumScore = 100  
    let points: Int  
    if self.sliderTargetDifference == 0 {  
        points = 200  
    } else if self.sliderTargetDifference == 1 {  
        points = 150  
    } else {  
        points = maximumScore - self.sliderTargetDifference  
    }  
    return points  
}
```

- ...then use it in `alertTitle()`:

```
func alertTitle() -> String {  
    let title: String  
    if self.sliderTargetDifference == 0 {  
        title = "Perfect!"  
    } else if self.sliderTargetDifference == 1 {  
        title = "Close!"  
    } else {  
        title = "Try Again!"  
    }  
    return title  
}
```

```
    } else if self.sliderTargetDifference < 5 {
        title = "You almost had it!"
    } else if self.sliderTargetDifference <= 10 {
        title = "Not bad."
    } else {
        title = "Are you even trying?"
    }
    return title
}
```

- Run the app. It works, without any changes that the player will notice. Once again, you've improved the underlying code without affecting the user experience.

Starting over

The **Start over** button at the lower-left corner of the screen does nothing at the moment. Let's make it active! When the player presses it, the following should happen:

- The score should reset to 0.
- The round should reset to 1.
- The slider should return to its original midway position of 50.
- A new random target value between 1 and 100 inclusive should generate.

The code that does this should go into the `action:` parameter for the Button view for the **Start over** button. We could put some code directly in there, but the `body` variable is already pretty cluttered. Let's put the steps listed above into a method and then call that method from the `action:` parameter.

This method won't return a value; it will merely perform some actions. For this reason, we'll give it a name that describes what it does: `startNewGame`.

- Add the following method to the end of `ContentView`'s *Methods* section:

```
func startNewGame() {
    self.score = 0
    self.round = 1
    self.sliderValue = 50.0
    self.target = Int.random(in: 1...100)
}
```

- Now that we have the `startNewGame()` method, let's use it. Scroll to the *Score row* section of the `body` variable and change it so that it looks like the following:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
        Text("Start over")
    }
    Spacer()
    Text("Score:")
    Text("\(self.score)")
    Spacer()
    Text("Round:")
    Text("\(self.round)")
    Spacer()
    Button(action: {}) {
        Text("Info")
    }
}
.padding(.bottom, 20)
```

- Run the app, play a round or two, and then press the **Start over** button. You'll start a new game, with the slider restored to its original position.

More DRYing

Just as we put the code for starting a new game into its own method to declutter the `body` variable, let's do the same for the code that starts a new round. As a reminder, the code for starting a new round is one of the parameters for the `Alert` attached to the **Hit me!** button:

```
Alert(title: Text(alertTitle()),
      message: Text(scoringMessage()),
      dismissButton: .default(Text("Awesome!")) {
        self.score = self.score + self.pointsForCurrentRound()
        self.target = Int.random(in: 1...100)
        self.round = self.round + 1
    }
```

Let's put the code from the last parameter listed above into its own method, which we'll name `startNewRound()`.

- Add the following method to the end of `ContentView`:

```
func startNewRound() {
    self.score = self.score + self.pointsForCurrentRound()
```

```
    self.round = self.round + 1
    self.sliderValue = 50.0
    self.target = Int.random(in: 1...100)
}
```

Note we're also resetting the slider to the midpoint at the start of a new round. Just as we do when starting a new game.

- Let's make use of the `startNewRound()` method. Scroll to the *Button row* section of the body variable and change it so that it looks like the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alert isVisible = true
}) {
    Text("Hit me!")
}
.presentation(self.$alert isVisible) {
    Alert(title: Text(alertTitle()),
        message: Text(scoringMessage()),
        dismissButton: .default(Text("Awesome!")) {
            self.startNewRound()
        }
    )
}
```

- Run the app and play a couple of rounds to confirm that the changes you made are working properly.

The methods you most recently added, `startNewGame()` and `startNewRound()` are at the end of `ContentView`:

```
func startNewGame() {
    self.score = 0
    self.round = 1
    self.sliderValue = 50.0
    self.target = Int.random(in: 1...100)
}

func startNewRound() {
    self.score = self.score + self.pointsForCurrentRound()
    self.round = self.round + 1
    self.sliderValue = 50.0
    self.target = Int.random(in: 1...100)
}
```

Both `startNewGame()` and `startNewRound()` end with the same two lines! We can DRY up this code by taking those two lines and putting them in their own method.

- Change the methods to look like this:

```
func startNewGame() {
    self.score = 0
    self.round = 1
    self.resetSliderAndTarget()
}

func startNewRound() {
    self.score = self.score + self.pointsForCurrentRound()
    self.round = self.round + 1
    self.resetSliderAndTarget()
}

func resetSliderAndTarget() {
    self.sliderValue = 50.0
    self.target = Int.random(in: 1...100)
}
```

- Once again, run the app to confirm that the changes you made work properly.

Making the code less self-ish

If you look at the code you've written so far, you'll see the keyword `self` all over the place. What does `self` mean, anyway?

`self` is Swift's way of saying "the current object." In this case, that object is `ContentView`. Any time you've had to refer to one of `ContentView`'s properties, you've prefaced it with `self`. For example, to reset the slider's position back to 50, you did it with this code:

```
self.sliderValue = 50.0
```

This is how you say "Set the `sliderValue` property of the current object to 50.0" in Swift.

It's the same for calls to `ContentView`'s methods — you also prefaced them with `self`. For example, to call the method that resets the slider and target values, you did it with this code:

```
self.resetSliderAndTarget()
```

Most of the time, when referring to the properties or methods of an object from within that object, `self` isn't necessary. This is another one of those cases where Swift is smart enough to infer what you mean.



Let's declutter the code and make it easier to read (and therefore easier to maintain and expand upon) by removing the unnecessary instances of `self`. We'll start with the methods:

- Remove all the instances of `self` from the *Methods* section so that it looks like this:

```
// Methods
// =====

func pointsForCurrentRound() -> Int {
    let maximumScore = 100
    let points: Int
    if sliderTargetDifference == 0 {
        points = 200
    } else if sliderTargetDifference == 1 {
        points = 150
    } else {
        points = maximumScore - sliderTargetDifference
    }
    return points
}

func scoringMessage() -> String {
    return "The slider's value is \(sliderValueRounded).\n" +
        "The target value is \(target).\n" +
        "You scored \(pointsForCurrentRound()) points this
round."
}

func alertTitle() -> String {
    let title: String
    if sliderTargetDifference == 0 {
        title = "Perfect!"
    } else if sliderTargetDifference < 5 {
        title = "You almost had it!"
    } else if sliderTargetDifference <= 10 {
        title = "Not bad."
    } else {
        title = "Are you even trying?"
    }
    return title
}

func startNewGame() {
    score = 0
    round = 1
    resetSliderAndTarget()
}

func startNewRound() {
    score = score + pointsForCurrentRound()
```



```
    round = round + 1
    resetSliderAndTarget()
}

func resetSliderAndTarget() {
    sliderValue = 50.0
    target = Int.random(in: 1...100)
}
```

- Run the app to confirm that removing all those instances of `self` didn't break it.

Now it's time to work on the properties, starting with those that *aren't* body. That's a really big property, and we'll look at it on its own in a moment.

- Remove all the instances of `self` from the `User interface views` part of the `Properties` section so that it looks like this:

```
// User interface views
@State var alertIsVisible = false
@State var sliderValue = 50.0
@State var target = Int.random(in: 1...100)
var sliderValueRounded: Int {
    Int(sliderValue.rounded())
}
@State var score = 0
@State var round = 1
var sliderTargetDifference: Int {
    abs(sliderValueRounded - target)
}
```

- Once again, run the app to confirm that removing those additional instances of `self` didn't break it.

For the `body` property, let's work on it in sections.

- Remove any instances of `self` from the `Target row` and `Slider row` sections so that they look like this:

```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
    Text("\(target)")
}

Spacer()

// Slider row
HStack {
    Text("1")
    Slider(value: $sliderValue, in: 1...100)
```

```
    Text("100")
}
```

- Run the app to confirm that it still works with these changes.

It's time to work on the *Button row* section. Here's what it looks like at the moment:

```
// Button row
Button(action: {
    self.alertIsVisible = true
}) {
    Text("Hit me!")
}.alert(isPresented: self.$alertIsVisible) {
    Alert(title: Text(alertTitle()),
        message: Text(self.scoringMessage()),
        dismissButton: .default(Text("Awesome!")) {
            self.startNewRound()
        }
    )
}
```

- Change the button's `action:` parameter so that it no longer includes the `self` keyword:

```
// Button row
Button(action: {
    alertIsVisible = true
}) {
```

Within a second or two of making this change, Xcode shows this error message:

```
// Button row
Button(action: {
    alertIsVisible = true
}) {
```

✖ Reference to property 'alertIsVisible' in closure requires explicit 'self.' to make capture semantics explicit

Xcode shows an error message after you remove the first 'self' in the button row

The error message is cryptic: “Reference to property ‘`alertIsVisible`’ in closure requires explicit ‘`self.`’ to make capture semantics explicit.” What does that mean?

The first unfamiliar word in the error message is *closure*. You could look up its meaning in Wikipedia, but it’s so dense with esoteric computer science terminology that you might know *less* about closures after reading it!

Instead of worrying about what the technical definition of a closure is, think of closures as code that you can put into variables or pass to methods and functions to be executed at a later time. That’s what the button’s `action:` parameter is a closure: it’s code for the button to execute whenever it’s pressed.

As a closure, the code in the button's `action:` parameter isn't part of `ContentView`, but separate from it. That means that they have no sense of the object they may be in or any of its properties or methods. However, they *can* access — or, as Xcode puts it, *capture* — the local variables around them.

That's what the "capture" in the error message refers to. Inside an object, the `self` variable is available anywhere, and the closure captures it.

Simply put, closures need to use `self` when referring to the properties or methods of the object they're in.

- With what you now know about closures and `self` in mind, remove only those instances of `self` from the *Button row* section that aren't inside closures. The resulting code should look like this:

```
// Button row
Button(action: {
    self.alertVisible = true
}) {
    Text("Hit me!")
}
.alert(isPresented: $alertVisible) {
    Alert(title: Text(alertTitle()),
        message: Text(scoringMessage()),
        dismissButton: .default(Text("Awesome!")) {
            self.startNewRound()
        }
    )
}
```

- Run the app to confirm that it still works with these changes.

Note: In the beginning, it may not always be clear when you can drop the `self` keyword and when it's absolutely necessary to use it. Until you get the hang of it, you can always err on the side of *not* using `self` and rely on Xcode's error messages to tell you when you need to include it.

- And finally, remove any instances of `self` from the *Score row* section that aren't in closures. This should be the result:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    })
}
```

```
        Text("Start over")
    }
    Spacer()
    Text("Score:")
    Text("\(score)")
    Spacer()
    Text("Round:")
    Text("\(round)")
    Spacer()
    Button(action: {}) {
        Text("Info")
    }
}
.padding(.bottom, 20)
```

- Run the app to confirm that it still works with these changes.

You've now removed all the unnecessary instances of `self` from the code. It's much easier to read now!

A couple more enhancements

After so many “behind the scenes” changes, it’s time for enhancements that the player *can* see! These will be easy to add, but they’ll also enhance the player experience.

Randomizing the slider position at the start of each round

Rather than reset the slider to the midpoint at the start of each round, let’s move it to a random position instead. Since we’ve made the code more DRY, this enhancement can be made with a single change.

- Update `resetSliderAndTarget()` to the following:

```
func resetSliderAndTarget() {
    sliderValue = Double.random(in: 1...100)
    target = Int.random(in: 1...100)
}
```

Remember that the slider is so precise that its values are `Double`, not `Int`. That’s why its value is randomized using `Double.random()` instead of `Int.random()`.

Randomizing the slider position when the game launches

When the game launches, the target and slider values are determined by their initial values, which are set when their variables are declared:

- The slider is set to 50.0.
- The target is set to a random whole number between 1 and 100 inclusive.

After the very first round, the game uses the `resetSliderAndTarget()` method to set the slider and target values. `resetSliderAndTarget()` is called by two different methods:

- `startNewRound()`: Called when the player dismisses the alert pop-up.
- `startNewGame()`: Called when the player presses the **Start over** button.

To make the game more consistent, it should call `startNewGame()` when the game launches. Luckily, there's a way to do that.

Every View object has a built-in set of methods that get called when certain view-related events happen. One of these events is when the view first appears. The method that gets called when this happens is called `onAppear()`. You provide `onAppear()` with the code that should be executed when the view appears.

All the onscreen elements in the game appear when the game launches, so you can use the `onAppear()` method for any of the views in `ContentView`'s `body` property. Since the `VStack` in `body` acts as the container for all the onscreen elements, we'll use its `onAppear()` method.

The call to `onAppear()` will look like this:

```
.onAppear() {  
    self.startNewGame()  
}
```

It will be called only once: the very first time that the `VStack` is drawn on the screen, which will happen only when the game is launched.

► Change the end of `body` so that it looks like this. I've included a lot of the surrounding code so because it can be hard to tell where the code should go:

```
// Score row  
HStack {
```

```
        Button(action: {
            self.startNewGame()
        }) {
            Text("Start over")
        }
        Spacer()
        Text("Score:")
        Text("\(score)")
        Spacer()
        Text("Round:")
        Text("\(round)")
        Spacer()
        Button(action: {}) {
            Text("Info")
        }
    }
    .padding(.bottom, 20)
}
.onAppear() {
    self.startNewGame()
}
}

// Methods
// ======
```

► Run the app. The slider position will now be randomized at the very start, instead of always starting at 50.0.

Key points

In this chapter, you did the following:

- You enhanced the message that appears when the player does particularly well.
- You also made improvements to the way points and bonuses are awarded.
- You enabled the **Start over** button.
- You did a fair bit of refactoring, which included removing redundant and unnecessary code.
- You learned about closures.
- You learned about views' `onAppear()` method and used it to automatically perform a task when the app is first launched.



At this point, *Bullseye* is pretty polished, and your task list is getting shorter. In the next chapter, you'll transform the game from its plain look and feel into something a little more polished.

You can find the project files for the current version of the app under **06 - Refactoring** in the Source Code folder.



Chapter 7: The New Look

Joey deVilla

Bullseye is looking good! The gameplay elements are complete and there's one item left in your to-do list: "Make it look pretty."

As satisfying as it was to get the game working, it's far from pretty. If you were to put it on the App Store in its current form, very few people would get excited to download it. Fortunately, iOS and SwiftUI make it easy for you to create good-looking apps. So, let's give *Bullseye* a makeover and add some visual flair.

This chapter covers the following:

- **Landscape orientation revisited:** Making changes to the project to improve its support for landscape orientation.
- **Spicing up the graphics:** Adding custom graphics to the app's user interface to give it a more polished look.
- **The “About” screen:** It's time to make the “Info” button work, which means that pressing it should take the player to *Bullseye*'s “About” screen.



Landscape orientation revisited

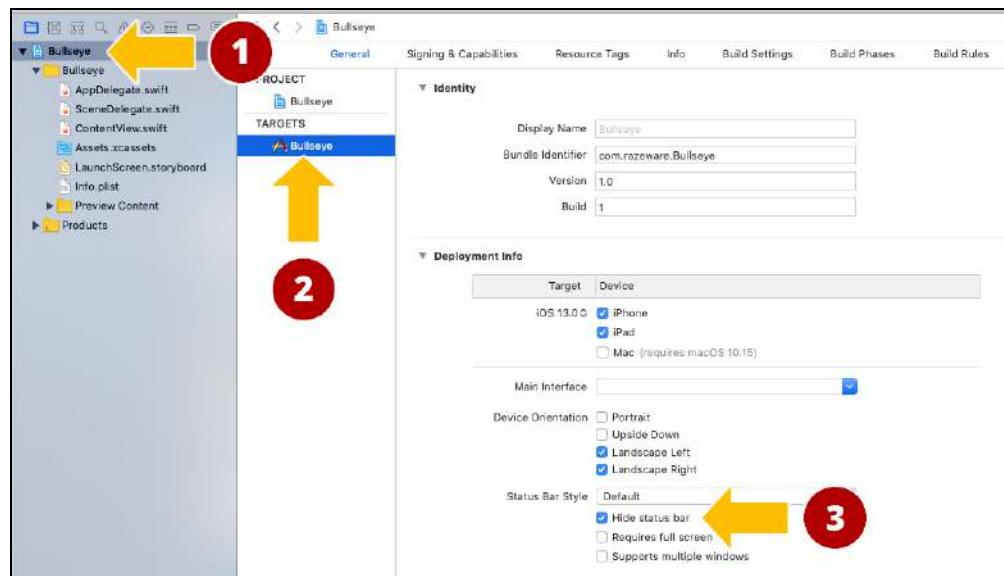
Let's revisit another item in the to-do list — “Put the app in landscape orientation.” Didn't you already do this?

You did! By changing the project settings so that the app supported only the **Landscape Left** and **Landscape Right** orientations. There's one last bit of cleaning up that you need to do to make landscape orientation support complete.

Apps in landscape mode don't display the iPhone status bar — the display at the top of the screen — unless you tell them to. That's great for *Bullseye*. Games require a more immersive experience and the status bar detracts from that.

The system automatically handles hiding the status bar for your game. But, you can improve the way *Bullseye* handles the status bar by making sure that it's always hidden, even when the app is launching.

► Go to the **Project Settings** screen and scroll down to **Deployment Info**. In the section marked **Status Bar Style**, check **Hide status bar**.



Hiding the status bar when the app launches

It's a good idea to hide the status bar while the app is launching. It takes a few seconds for the operating system to load the app into memory and start it up. During that time the status bar remains visible unless you hide it using this option.

It's only a small detail, but the difference between a mediocre app and a great one is the small details.

- That's it! Run the app and you'll see that the status bar is history.

Info.plist

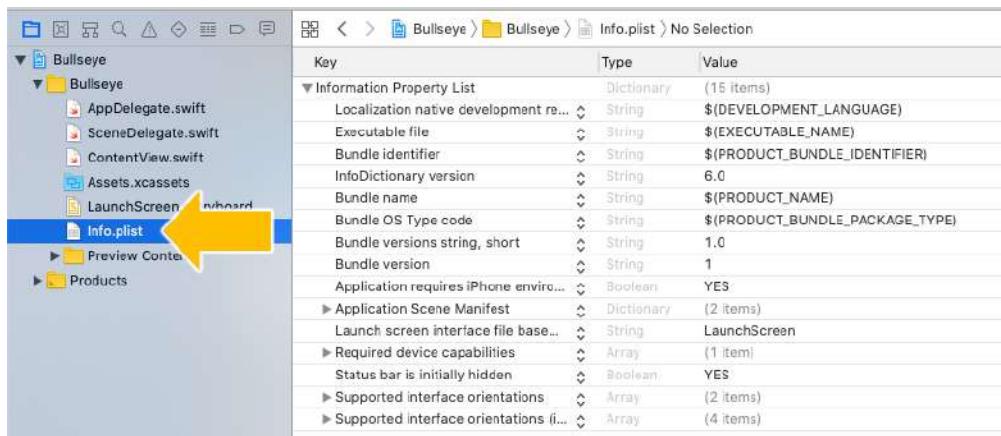
Most of the options from the **Project Settings** screen, such as the supported device orientations and whether the status bar is visible during launch, are stored in a configuration file called **Info.plist**.

The information in **Info.plist** tells iOS how the app will behave. It also describes certain characteristics of the app that don't fit anywhere else. Such as the app's version number.

In earlier versions of Xcode, you often had to edit **Info.plist** by hand. This was a tedious and sometimes error-prone process. With the latest versions of Xcode, this is hardly necessary anymore. You can make most of the changes directly from the **Project Settings** screen.

Even with the changes to Xcode that minimize the amount of time you have to work directly with **Info.plist**, it's still good to know of its existence and what it looks like.

- Go to the **Project navigator** and select the file named **Info.plist** to take a peek at its contents.

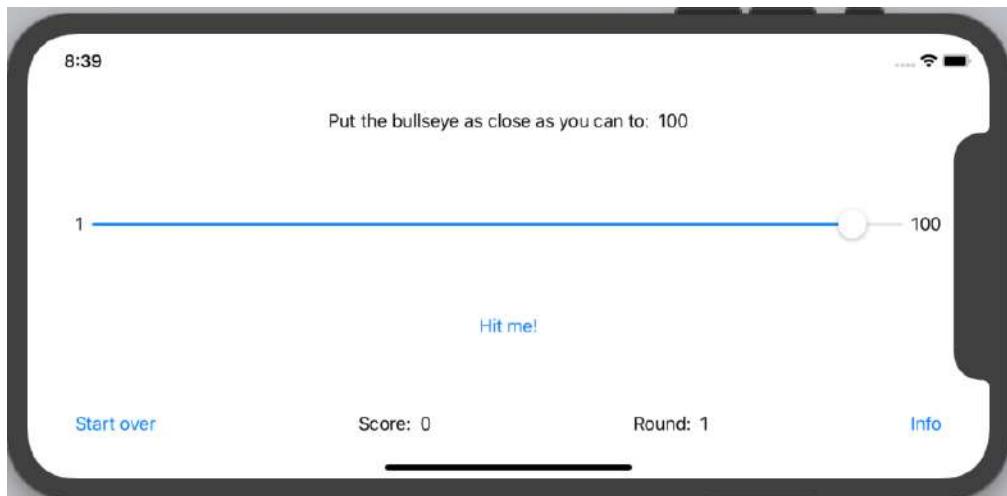


The **Info.plist** file is a list of configuration options and their values. Most of these may not make sense to you, but that's OK. They don't always make sense to many experienced developers either.

Notice the option **Status bar is initially hidden**. It has the value YES. This is the option that you just changed.

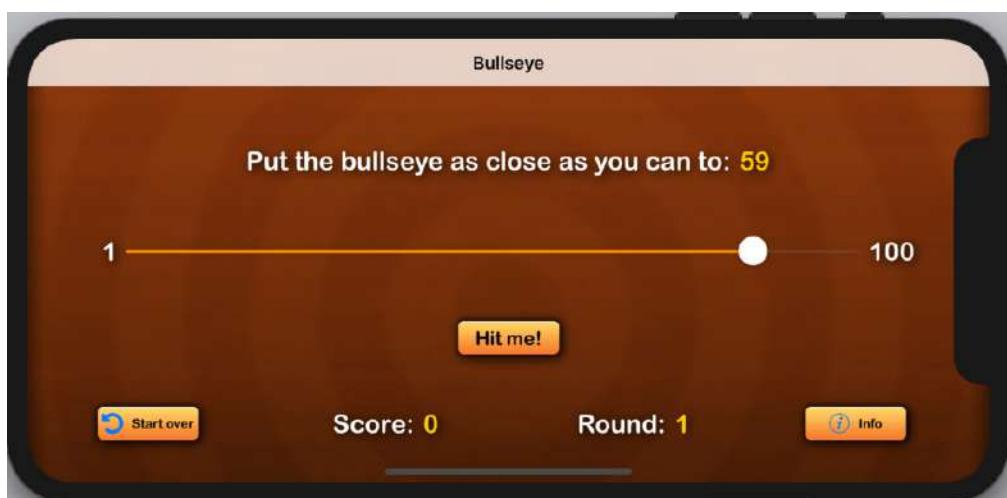
Spicing up the graphics

Getting rid of the status bar is only the first step. We want to go from this...



Yawn...

...to something that's more like this:



How the app will look in the end



The actual controls won't change. You'll simply be using images to spruce up their look. You'll also adjust the user interface's colors and typefaces.

You can put an image in the background, on the buttons, and even on the slider, to customize the appearance of each. The images you use should generally be in PNG format, though JPG files would work too.

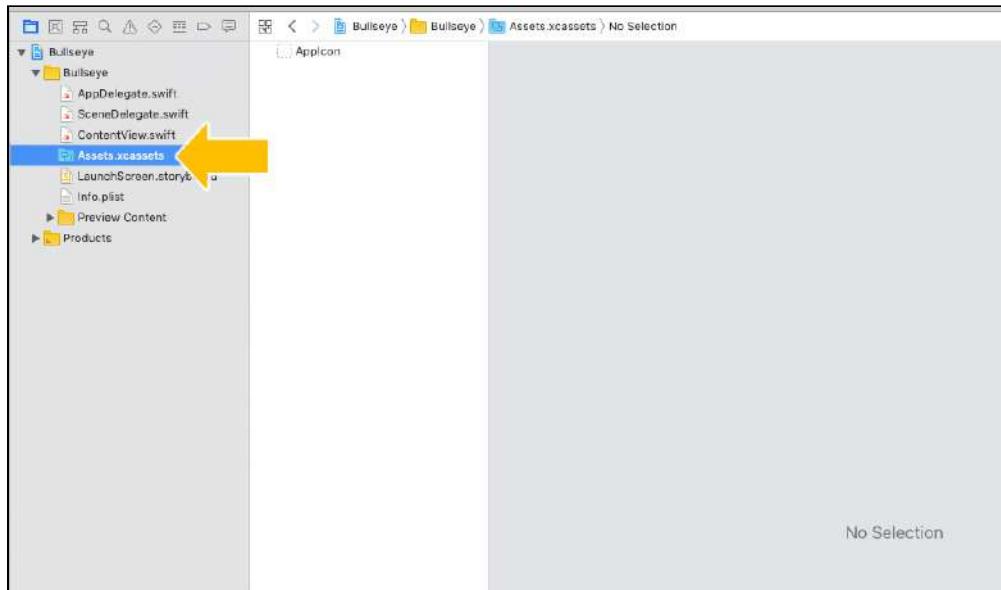
Adding the image assets

If you're artistically challenged, then don't worry: we've provided a set of images for you. But if you do have *mad Photoshop skillz*, then by all means feel free to design and use your own images.

The **Resources** folder that comes with this book contains a subfolder named **Images**. You'll import these images into the Xcode project.

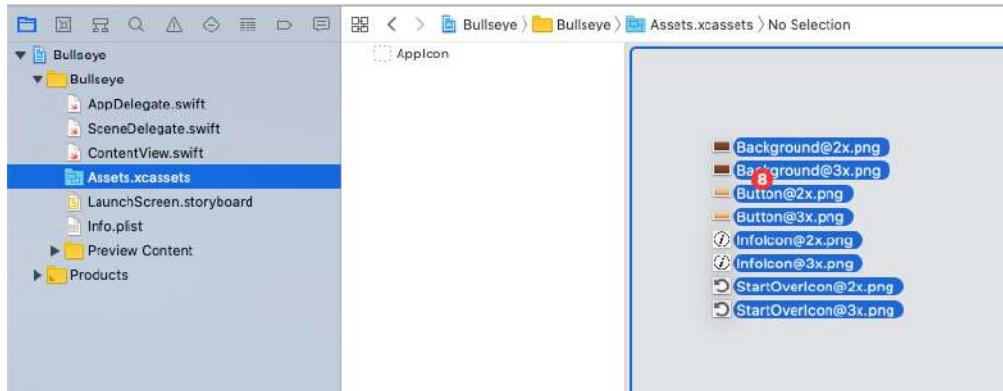
- In the **Project navigator**, find **Assets.xcassets** and click on it.

This item is the app's *asset catalog*, which stores all the images that go into it. Right now, it's empty and contains a placeholder for the app icon. You'll add an icon and images soon:



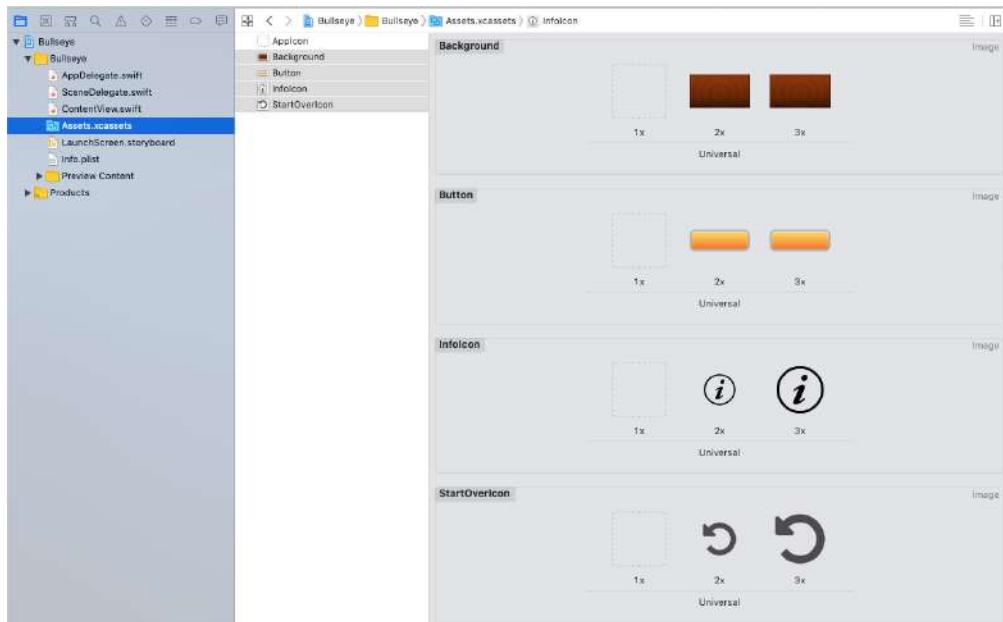
The asset catalog is initially empty

- Open the **Resources** folder that comes with this book, then open the subfolder named **Images**. Drag the files within into the Xcode project:



Dragging files into the asset catalog

Xcode will copy all the image files from the **images** folder into the project's asset catalog:



The images are now inside the asset catalog

Note: If Xcode added a folder named **Images** instead of the individual image files, then try again. This time, make sure that you drag the files inside the **Images** folder into Xcode rather than the folder itself.

1x, 2x, and 3x displays

For each image you dragged into the asset catalog, you created an **image set**. Image sets allow an app to support different devices with different screen resolutions. Each image set has a slot for the **1x**, **2x** and **3x** version of the image:

- **1x** images are for low-resolution screens, with pixels that seem big and chunky by today's standards. Only the first iPhones — the original, 2G, 3G, and 3GS — have these screens. None of these devices can run a version of iOS released after 2012. You're pretty unlikely to write apps that use **1x** graphics.
- **2x** images are for high-resolution Retina screens. As the name implies, they're drawn with twice the number of pixels as a **1x** image. A wide range of iPhones — from the iPhone 4 through 8 and the iPhone XR — and iPads and late-model iPods have these screens.
- **3x** images are for high-resolution Retina HD screens, which the iPhone X, XS, XR and any iPhone with a “+” in its name have. These images are drawn with three times the number of pixels as a **1x** image.

When an app displays an image, iOS tries to use the version of the image that best matches the device's screen resolution. If that's not available, it uses the next best version.

When you dragged the images into the asset catalog, Xcode used their filenames to determine which image set and slot to drop them into. For example, the images in the **Background@2x.png** and **Background@3.png** files go into the **Background** image set. The **Background@2x.png** image goes into the **2x** slot and the **Background@3x.png** image goes into the **3x** slot. Had there been a **Background.png** file, it would go into the **Background** image set's **1x** slot. Any file whose name doesn't end with **2x** or **3x** is assumed to be a **1x** image.

If you'd rather determine which images are **1x**, **2x** and **3x**, you can also drag and drop individual images into their respective slots.

Putting up the wallpaper

Let's begin by replacing *Bullseye*'s drab white background with the more appealing **Background** image that you added to the app's asset catalog:



SwiftUI makes it easy to change the background of any view. This is done using view's `background()` method, which lets you specify another view to use as the background. In this case, we'll create an `Image` view containing the **Background** image asset and use it as the background for the main screen.

Remember that `ContentView`'s `body` property contains all the user interface elements on the app's screen. If you look at its contents, you can see that it contains a `VStack`. This, in turn, contains all other user interface elements:

```
var body: some View {
    VStack {
        Spacer()

        // Target row
        ...
    }
}
```

We'll use the `background()` method of this `VStack` to set its background image.

► Scroll to the end of the `VStack` at the end of the `body` property and change it so that it looks like this. Add the call to the `VStack`'s `background()` method on the line after the call to the `onAppear()` method:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    })
}
```

```
        })
    Text("Start over")
}
Spacer()
Text("Score:")
Text("\(self.score)")
Spacer()
Text("Round:")
Text("\(self.round)")
Spacer()
Button(action: {}) {
    Text("Info")
}
.padding(.bottom, 20)
}
.onAppear() {
    self.startNewGame()
}
.background(Image("Background"))
```

The line of code that you just added, `.background(Image("Background"))`, creates an `Image` view and fills it with the appropriate **Background** image from the asset catalog. This is either the `2x` or `3x` version, depending on the device it's running on. The code then makes it the background view for the `VStack`.

► Let's see what this code does. Run the app.

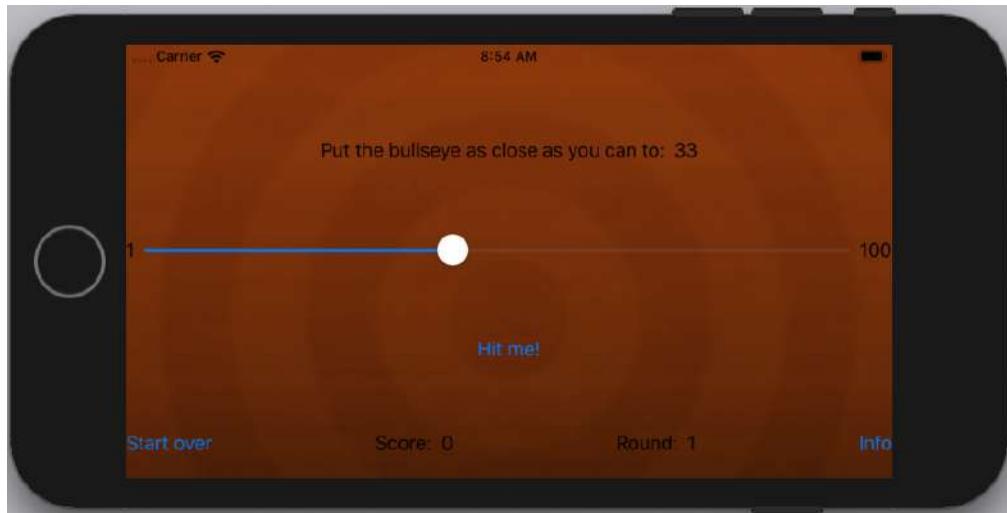
Here's what the app looks like on the Simulator when it's simulating the iPhone XR, which uses the `3x` background:



The 3x background on the iPhone XR

Let's try the app on a smaller device without the "notch". We'll use the iPhone 8, which uses the **2x** background.

- Select **iPhone 8** and run the app:



The 2x background on the iPhone 8

That takes care of the background. Let's work on the text.

Changing the text

Now that *Bullseye* has its new background image, the black text is now nearly illegible. We'll need to change it so that it stands out better. Once again, we'll use some built-in methods to change the text's appearance so that it's legible against the background. Let's start with the "Put the bullseye as close as you can to:" and target value text.

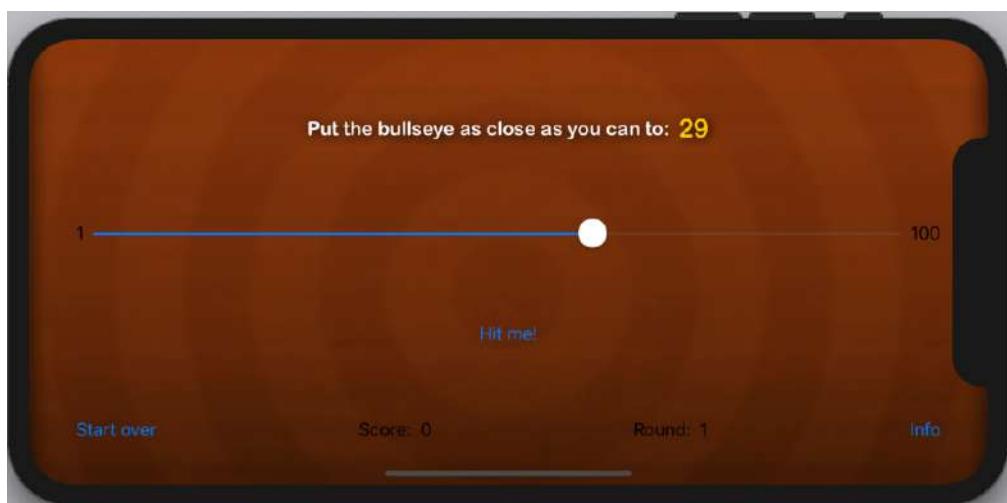
- Scroll to the part of the body property marked *Target row* and change it so that it becomes the following:

```
// Target row
HStack {
    Text("Put the bullseye as close as you can to:")
        .font(Font.custom("Arial Rounded MT Bold", size: 18))
        .foregroundColor(Color.white)
        .shadow(color: Color.black, radius: 5, x: 2, y: 2)
    Text("\(self.target)")
        .font(Font.custom("Arial Rounded MT Bold", size: 24))
        .foregroundColor(Color.yellow)
```

```
        .shadow(color: Color.black, radius: 5, x: 2, y: 2)  
    }
```

On both Text objects, three methods are being called in a chain:

- `font()`, which specifies the typeface that the Text object should use. It expects a `Font` object, and we're using its `custom` method to create one with a specified typeface — Arial Rounded MT Bold — and size in points. We'll make the target value a little bigger for emphasis. `font()`'s output is a new Text object in the specified typeface, which is then immediately fed to...
 - `foregroundColor()`, which specifies the color that the Text object should be. It expects a `Color` object. We're using two built-in values: `Color.white` for the instruction text and `Color.yellow` for the target value text. Its output is a Text object in the new color, which is passed to...
 - `shadow()`, which draws a shadow behind the Text object. It expects the size of the shadow's radius (how far it spreads) and its x and y-offsets in points. Its output is a Text object with a shadow, and this is the object that's drawn onscreen.
- Switch the Simulator back to **iPhone XR** and run the app. You should be able to read the instructions and the target value now:



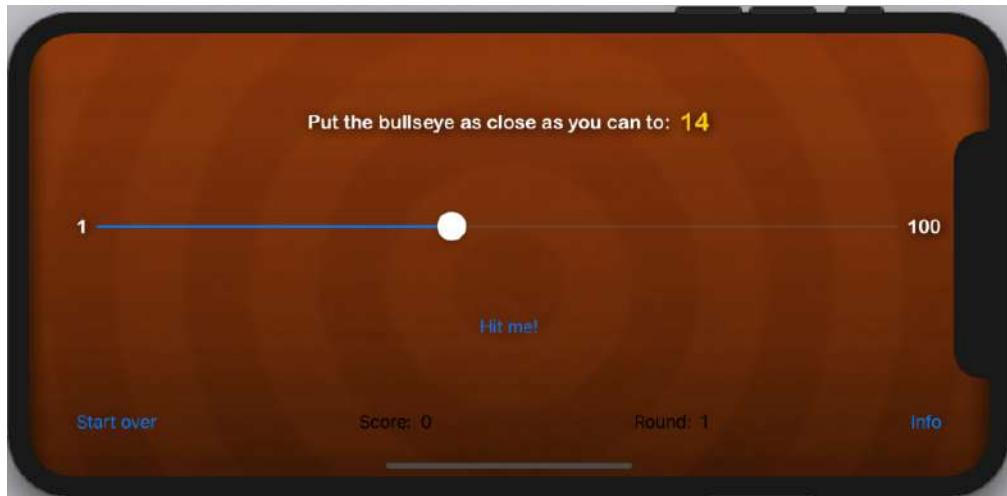
The target row, with styled text

Let's apply similar changes to the "1" and "100" on either side of the slider.

- Scroll to the part of the body property marked *Slider row* and change it so that it becomes the following:

```
// Slider row
HStack {
    Text("1")
        .font(Font.custom("Arial Rounded MT Bold", size: 18))
        .foregroundColor(Color.white)
        .shadow(color: Color.black, radius: 5, x: 2, y: 2)
    Slider(value: self.$sliderValue, from: 1.0, through: 100.0)
    Text("100")
        .font(Font.custom("Arial Rounded MT Bold", size: 18))
        .foregroundColor(Color.white)
        .shadow(color: Color.black, radius: 5, x: 2, y: 2)
}
```

- Run the app. The numbers on either side of the slider should be legible:



The target and slider rows, with styled text

At the bottom of the screen are the text elements that display the score and round. We'll give this row the same treatment as the target row: white text for titles, and larger yellow text for values.

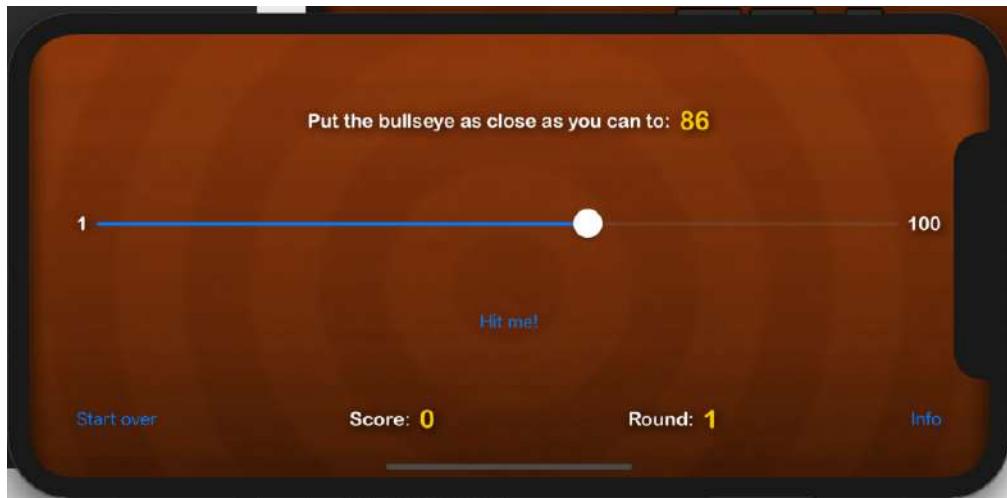
- Scroll to the part of the body property marked *Score row* and change it so that it becomes the following:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
```

```
        Text("Start over")
    }
Spacer()
Text("Score:")
    .font(Font.custom("Arial Rounded MT Bold", size: 18))
    .foregroundColor(Color.white)
    .shadow(color: Color.black, radius: 5, x: 2, y: 2)
Text("\(self.score)")
    .font(Font.custom("Arial Rounded MT Bold", size: 24))
    .foregroundColor(Color.yellow)
    .shadow(color: Color.black, radius: 5, x: 2, y: 2)
Spacer()
Text("Round:")
    .font(Font.custom("Arial Rounded MT Bold", size: 18))
    .foregroundColor(Color.white)
    .shadow(color: Color.black, radius: 5, x: 2, y: 2)
Text("\(self.round)")
    .font(Font.custom("Arial Rounded MT Bold", size: 24))
    .foregroundColor(Color.yellow)
    .shadow(color: Color.black, radius: 5, x: 2, y: 2)
Spacer()
Button(action: {}) {
    Text("Info")
}
.padding(.bottom, 20)
```

- Run the app to see all the text changes. You might notice that Xcode is taking more than the usual amount of time to compile your code.

Once the app starts on the Simulator, it will look like this:



The app, with all its text styled

The app's looking a lot better now. Let's work on those buttons next!

Making the buttons look like buttons

Let's make the buttons look more like buttons.

Just as we used the `background()` method to change the background of the `VStack` that contains the app's user interface, we'll do the same for the buttons. We'll do so by using each `Button` view's `background()` method and the image in the asset catalog named **Button**:



The button image

We'll also use the `font()` and `color()` methods of the `Text` objects contained within those buttons to customize their typeface and color, and the `shadow()` method on the button's `Image` object.

Let's update the appearance of the **Hit me!** button with the button background image and the Arial Rounded MT Bold typeface at size 18.

- Scroll to the part of the `body` property marked *Button row* and change it so that it becomes the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertIsVisible = true
}) {
    Text("Hit me!")
        .font(Font.custom("Arial Rounded MT Bold", size: 18))
        .foregroundColor(Color.black)
    }
    .background(Image("Button")
        .shadow(color: Color.black, radius: 5, x: 2, y: 2)
    )
    .alert(isPresented: self.$alertIsVisible) {
        Alert(title: Text(alertTitle()),
            message: Text(scoringMessage()),
            dismissButton: .default(Text("Awesome!")) {
                self.startNewRound()
            }
        )
    }
}
```

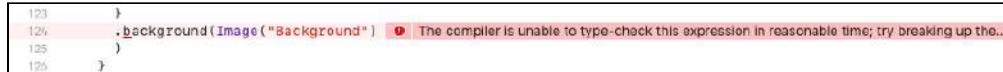
- Run the app to see the **Hit me!** button's new look. You may find that Xcode seems to come to a halt while compiling your app as it displays something like this at the top of its window:



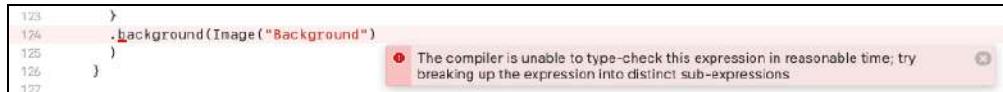
Something's gone wrong, and it's time to find out what that is. Let Xcode try to compile your code for a little longer. It will eventually give up, and you'll get a notification that the build failed. You'll see something like this at the top of the Xcode window:



- Tap that red error icon. This error message will appear:



- Tap the error message's red error icon to see it in full:



The message may sound cryptic: *The compiler is unable to type-check this expression in reasonable time; try breaking up the expression into distinct sub-expressions*. Simply put, Xcode is saying: “That thing that you’ve put into the body property is a lot to deal with. Is there any way you can simplify it?”

It turns out that all those extra methods to change fonts, colors and backgrounds and add shadows to the text and button views in body are too much for the compiler to handle.

With this news, you might be tempted to throw your hands in the air and walk away from your computer in frustration. Don’t worry, there *is* a solution.

Introducing ViewModifier

In programming, you’ll sometimes find that the solution to an error is embedded in its error message. It’s true for this particular case. The fix to our compiler problem is in the last part of the message: *try breaking up the expression into distinct sub-expressions*. In other words, Xcode is asking us to break that big body property into smaller parts.

If you look at `body` in its current state, you'll see a lot of repetition. For starters, there are *five* instances where the following methods are called on a `Text` view:

```
.font(Font.custom("Arial Rounded MT Bold", size: 18))  
.foregroundColor(Color.white)
```

There are also *three* instances where the following methods are called on a `Text` view:

```
.font(Font.custom("Arial Rounded MT Bold", size: 24))  
.foregroundColor(Color.yellow)
```

And there are *nine* instances where this method is called on `Text` and `Image` views to give it a shadow:

```
.shadow(color: Color.black, radius: 5, x: 2, y: 2)
```

If there was some way to DRY up `body`'s code and put these repeated calls to these methods into a package that can be called again and again, it might help the compiler get past its stumbling block. Fortunately for us, there *is* a way: The `ViewModifier` protocol. Remember the definition for "protocol" from a couple of chapters back: It's a set of properties and methods that an object agrees to include as part of its code.

The `ViewModifier` protocol states that any object that adopts it agrees to furnish a method named `body()`, which accepts some content. That content can then have a number of calls to any number of `View` methods made on it. We're going to create some objects that adopt the `ViewModifier` protocol and use them to create packages of methods that we'll use to style different parts of the user interface.

It might be easier to show you how to use `ViewModifier` instead of explaining it.

► Add the following in the space between the end of `ContentView` and the start of the preview code:

```
// View modifiers  
// ======  
  
struct LabelStyle: ViewModifier {  
    func body(content: Content) -> some View {  
        content  
            .font(Font.custom("Arial Rounded MT Bold", size: 18))  
            .foregroundColor(Color.white)  
            .shadow(color: Color.black, radius: 5, x: 2, y: 2)  
    }  
}
```

The most important part of this object is its `body()` method. The `body()` method accepts a single piece of information, `content`, which contains the content of the view that called it. It then calls a set of specified methods on that content.

In the case of `LabelStyle`, it calls the `font()`, `foregroundColor()` and `shadow()` methods on the content to change its font to Arial Rounded MT Bold with a size of 18 points, its color to white and with a shadow. We'll use this to style the `Text` objects that display the instructions, as well as the labels for the slider, score, and number of rounds.

- Add the following in the space between the end of `LabelStyle` and the start of the preview code:

```
struct ValueStyle: ViewModifier {  
    func body(content: Content) -> some View {  
        content  
            .font(Font.custom("Arial Rounded MT Bold", size: 24))  
            .foregroundColor(Color.yellow)  
            .shadow(color: Color.black, radius: 5, x: 2, y: 2)  
    }  
}
```

`ValueStyle` is almost identical to `LabelStyle`. The key difference is its calls to `font()` and `foregroundColor()` change the content's font to Arial Rounded MT Bold with a size of 24 points, and its color to yellow. We'll use this to style the `Text` objects that display the game values: the target, score, and number of rounds.

Now that we have these two objects that adopt the `ViewModifier` protocol — `LabelStyle` and `ValueStyle` — we can use them to style the views in `ContentView`'s `body` property.

- Change the *Target row* section of `ContentView`'s `body` property to the following:

```
// Target row  
HStack {  
    Text("Put the bullseye as close as you can  
to:").modifier(LabelStyle())  
    Text("\(self.target)").modifier(ValueStyle())  
}
```

The code's a lot simpler! Instead of calling a chain of `font()`, `foregroundColor()` and `shadow` methods on the `Text` objects in this row, we're using `View`'s `modifier()` method to style them. The `modifier()` method takes a single argument — an object that has adopted the `ViewModifier` protocol — and uses that object to style its `View`.

In the code above, `modifier()` uses `LabelStyle` to style the “Put the bullseye as close as you can to:” text, and `ValueStyle` to style the displayed value of `target`.

- Change the *Slider row* section of `ContentView`’s body property to the following:

```
// Slider row
HStack {
    Text("1").modifier(LabelStyle())
    Slider(value: self.$sliderValue, from: 1.0, through: 100.0)
    Text("100").modifier(LabelStyle())
}
```

Again, we’re using `modifier()` and `LabelStyle`, this time to style the text on either side of the slider. This leaves us with one more row to update.

- Change the *Score row* section of `ContentView`’s body property to the following:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
        Text("Start over")
    }
    Spacer()
    Text("Score:").modifier(LabelStyle())
    Text("\(self.score)").modifier(ValueStyle())
    Spacer()
    Text("Round:").modifier(LabelStyle())
    Text("\(self.round)").modifier(ValueStyle())
    Spacer()
    Button(action: {}) {
        Text("Info")
    }
}
.padding(.bottom, 20)
```

- It’s time to see if all these changes worked. Run the app. This time, you’ll see that it compiles quickly, and that all the text and button styling has taken effect:



Some refactoring and more styling

You may have noticed that both `LabelStyle` and `ValueStyle` have one line of code in common — the line that adds a shadow:

```
.shadow(color: Color.black, radius: 5, x: 2, y: 2)
```

You also may have noticed that this method is also called on the `Image` views inside the `Button` views. This repetition suggests that we do some DRYing and put this code in a single place where we can call it. It's time to create a `ViewModifier` for shadows!

- Add the following after `ValueStyle` and before the start of the preview code:

```
struct Shadow: ViewModifier {  
    func body(content: Content) -> some View {  
        content  
            .shadow(color: Color.black, radius: 5, x: 2, y: 2)  
    }  
}
```

Before we continue, take a look at the signature of the `body` method of every `ViewModifier` object:

```
func body(content: Content) -> some View
```

This says that the `body` method returns either a `View` or something that behaves like a `View`. If you couple this with the fact that `View` objects use `ViewModifier` objects by way of the `modifier()` method, it means that `ViewModifiers` can use other `ViewModifiers`!

- Change `LabelStyle` and `ValueStyle` so that they incorporate Shadow:

```
struct LabelStyle: ViewModifier {  
    func body(content: Content) -> some View {  
        content  
            .font(Font.custom("Arial Rounded MT Bold", size: 18))  
            .foregroundColor(Color.white)  
            .modifier(Shadow())  
    }  
}  
  
struct ValueStyle: ViewModifier {  
    func body(content: Content) -> some View {  
        content  
            .font(Font.custom("Arial Rounded MT Bold", size: 24))  
            .foregroundColor(Color.yellow)  
            .modifier(Shadow())  
    }  
}
```

```
    }
```

- Run the app to confirm that these changes work.

Now that there's a `ViewModifier` for shadows, we can use it on the button image in the button row.

- Scroll to the part of the body property marked *Button row* and change it so that it becomes the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertIsVisible = true
}) {
    Text("Hit me!")
        .font(Font.custom("Arial Rounded MT Bold", size: 18))
        .foregroundColor(Color.black)
    }
    .background(Image("Button"))
        .modifier(Shadow())
    )
    .alert(isPresented: self.$alertIsVisible) {
        Alert(title: Text(alertTitle()),
            message: Text(scoringMessage()),
            dismissButton: .default(Text("Awesome!")) {
                self.startNewRound()
            }
        )
    }
}
```

It's time to make the buttons in the score row look like buttons, complete with shadows.

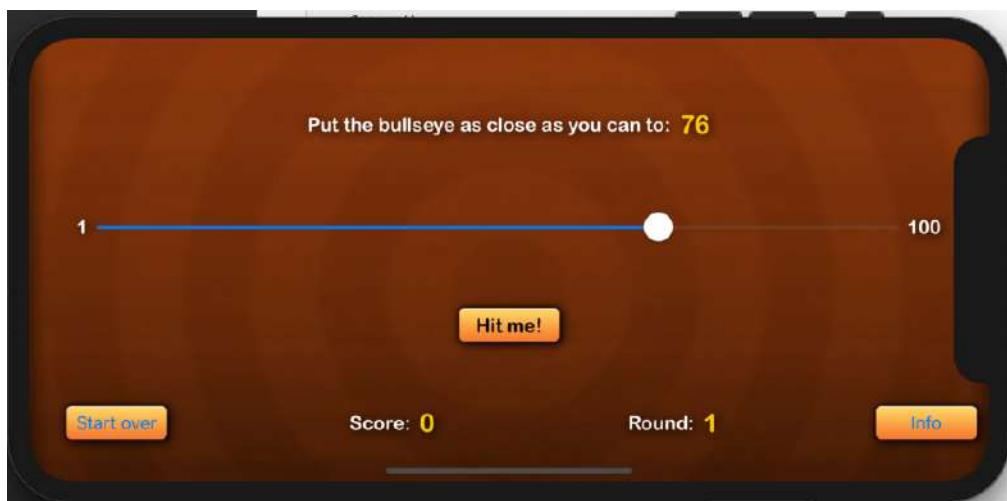
- Scroll to the part of the body property marked *Score row* and change it so that it becomes the following:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
        Text("Start over")
    }
    .background(Image("Button"))
        .modifier(Shadow())
    )
    Spacer()
    Text("Score:").modifier(LabelStyle())
```



```
Text("\(self.score)").modifier(ValueStyle())
Spacer()
Text("Round:").modifier(LabelStyle())
Text("\(self.round)").modifier(ValueStyle())
Spacer()
Button(action: {}) {
    Text("Info")
}
.background(Image("Button")
    .modifier(Shadow())
)
.padding(.bottom, 20)
```

- Run the app and marvel at its complete set of buttons:



All the buttons now look like buttons

Let's create some `ViewModifiers` for the button text. We'll create one with larger text called `ButtonLargeTextStyle` for the **Hit me!** button, and one with smaller text called `ButtonSmallTextStyle` for the **Start over** and **Info** buttons.

- Add the following after `Shadow` and before the start of the preview code:

```
struct ButtonLargeTextStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(Font.custom("Arial Rounded MT Bold", size: 18))
            .foregroundColor(Color.black)
    }
}
```

```
struct ButtonSmallTextStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(Font.custom("Arial Rounded MT Bold", size: 12))
            .foregroundColor(Color.black)
    }
}
```

With these new `ViewModifiers`, we can style the button text.

- Scroll to the part of the `body` property marked *Button row* and update it to the following:

```
// Button row
Button(action: {
    print("Button pressed!")
    self.alertVisible = true
}) {
    Text("Hit me!").modifier(ButtonLargeTextStyle())
}
.background(Image("Button")
    .modifier(Shadow()))
.alert(isPresented: self.$alertVisible) {
    Alert(title: Text(alertTitle()),
        message: Text(scoringMessage()),
        dismissButton: .default(Text("Awesome!")) {
            self.startNewRound()
        }
    )
}
```

- Scroll to the part of the `body` property marked *Score row* and update it to the following:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
        Text("Start over").modifier(ButtonSmallTextStyle())
    }
    .background(Image("Button")
        .modifier(Shadow()))
    Spacer()
    Text("Score:").modifier(LabelStyle())
    Text("\(self.score)").modifier(ValueStyle())
    Spacer()
    Text("Round:").modifier(LabelStyle())
    Text("\(self.round)").modifier(ValueStyle())
}
```



```
Spacer()  
Button(action: {}) {  
    Text("Info").modifier(ButtonSmallTextStyle())  
}  
.background(Image("Button")  
.modifier(Shadow())  
)  
}  
.padding(.bottom, 20)
```

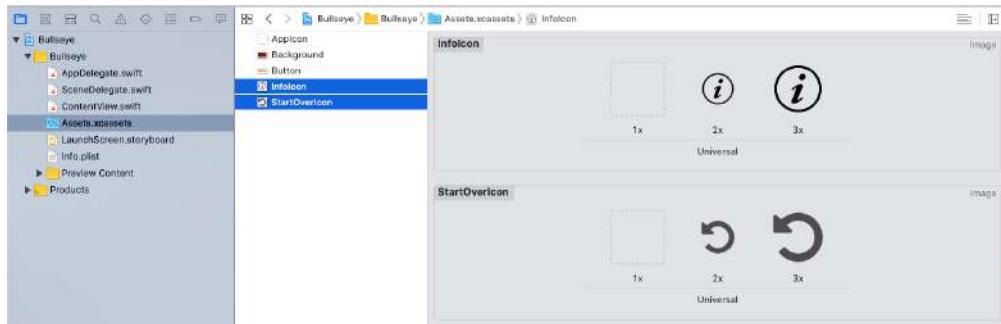
- Run the app. It's looking pretty nice now!



All the buttons now have styled text

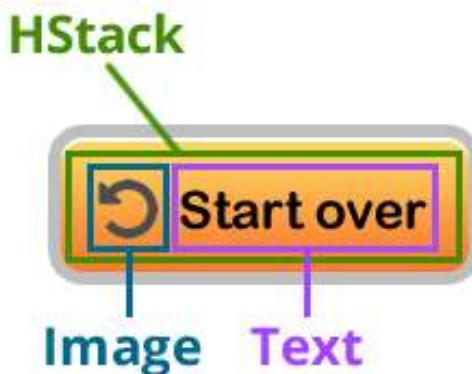
Putting images inside buttons

Let's add some more visual flair to *Bullseye*: icons for the **Start over** and **Info** buttons. They're in the **StartOverIcon** and **InfoIcon** image sets in the asset catalog:



InfoIcon and StartOverIcon in the asset catalog

Button objects are a kind of View, and like all views, they can contain other views. This makes it possible to create buttons that contain more than a single line of text. We'll customize the **Start over** button by combining an Image view and a Text view inside an HStack, as shown below:



- Change the *Score* row section of ContentView's body property to the following:

```
HStack {  
    Button(action: {  
        self.startNewGame()  
    }) {  
        HStack {  
            Image("StartOverIcon")  
            Text("Start over").modifier(ButtonSmallTextStyle())  
        }  
    }.background(Image("Button"))  
    .modifier(Shadow())  
}  
Spacer()  
Text("Score:").modifier(LabelStyle())  
Text("\(self.score)").modifier(ValueStyle())  
Spacer()  
Text("Round:").modifier(LabelStyle())  
Text("\(self.round)").modifier(ValueStyle())  
Spacer()  
Button(action: {}) {  
    HStack {  
        Image("InfoIcon")  
        Text("Info").modifier(ButtonSmallTextStyle())  
    }  
}.background(Image("Button"))  
    .modifier(Shadow())  
}
```

```
.padding(.bottom, 20)
```

- Run the app to see the changes:



Adding accent colors

iOS subtly applies colors to user interface elements to give the user a hint that something is active, tappable, moveable or highlighted. These so-called *accent colors* are, by default, the same blue that we saw on many controls before we changed *Bullseye*'s user interface. Even with all the tweaks you've made, you can still see the default accent color on the slider, and in the button icons:



You can change a view's accent color, along with the accent color of any views it contains, using the `accentColor()` method. Let's change the slider's accent color to green, which should stand out against its background.

- Change the `Score` row section of `ContentView`'s body property to the following:

```
// Slider row
HStack {
    Text("1").modifier(LabelStyle())
    Slider(value: self.$sliderValue, from: 1.0, through: 100.0)
        .accentColor(Color.green)
    Text("100").modifier(LabelStyle())
}
```

- Run the app. You'll now see the slider's accent color, which highlights the left side of its track, is now green:



The slider, with its new custom accent color

You're not limited to using pre-defined colors. Let's create a custom color, midnight blue, and use it as the accent color for the **Start over** and **Info** buttons.

First, we need to define what midnight blue is. If you're familiar with web development, you probably know the RGB (red, green and blue) color model. If not, you specify colors as a combination of three numbers representing red, green and blue on a scale of 0 through 255.

We'll define midnight blue as this color:



Midnight blue

In web development, you specify RGB colors as a set of three hexadecimal (base 16) numbers. The color we're calling midnight blue is defined by these values:

- **red**: 0 in hexadecimal, which is also 0 in decimal.
- **green**: 33 in hexadecimal, which is 51 in decimal.
- **blue**: 66 in hexadecimal, which is 102 in decimal.

When you instantiate a `Color` object in SwiftUI, it expects to get the values for red, green and blue on a scale of 0 to 1. Converting the decimal values for midnight blue to this scale is simple: Divide each one by 255, which gives us:

- **red**: 0
- **green**: 0.2
- **blue**: 0.4

First, you need to define midnight blue.

► Change the start of `ContentView` so that it looks like this:

```
// Properties
// =====

// Colors
let midnightBlue = Color(red: 0,
                           green: 0.2,
                           blue: 0.4)

// Game stats
```

```
@State var target: Int = Int.random(in: 1...100)
@State var score: Int = 0
@State var round: Int = 1
```

Now that we have defined `midnightBlue`, let's apply it to the `HStack` containing the score row. This will set the accent color for all the views contained within.

- Change the *Score row* section of `ContentView`'s `body` property to the following:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
        HStack {
            Image("StartOverIcon")
            Text("Start over").modifier(ButtonSmallTextStyle())
        }
    }
    .background(Image("Button"))
    .modifier(Shadow())
}
Spacer()
Text("Score:").modifier(LabelStyle())
Text("\(self.score)").modifier(ValueStyle())
Spacer()
Text("Round:").modifier(LabelStyle())
Text("\(self.round)").modifier(ValueStyle())
Spacer()
Button(action: {}) {
    HStack {
        Image("InfoIcon")
        Text("Info").modifier(ButtonSmallTextStyle())
    }
}
.background(Image("Button"))
.modifier(Shadow())
}
.padding(.bottom, 20)
.accentColor(midnightBlue)
```

- Run the app. The accent color for the **Start over** and **Info** buttons is now midnight blue.

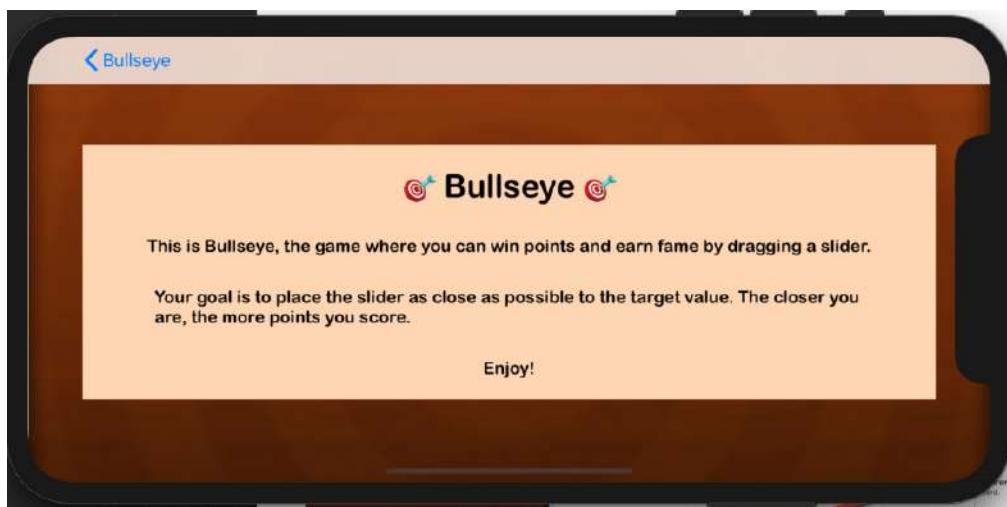
The “About” screen

Your game looks awesome and your to-do list is done. Does this mean that you are done with *Bullseye*?

Not so fast! Remember the **Info** button at the lower right corner of the screen? Try tapping it. Does it do anything? No?

Oops! Looks as if we forgot to add any functionality to that button! It's time to rectify that — let's add an “About” screen to the game. It'll appear whenever the player presses **Info**.

Here's what it will look like at the end:



The finished About screen

Most apps, even very simple games, have more than one screen. This is as good a time as any to learn how to add additional screens to your apps.

It's worth repeating: The term “view” can refer to any element on a screen in an app, but also the screen itself. Views can contain other views, and the screen is a view that contains all the views on that screen.

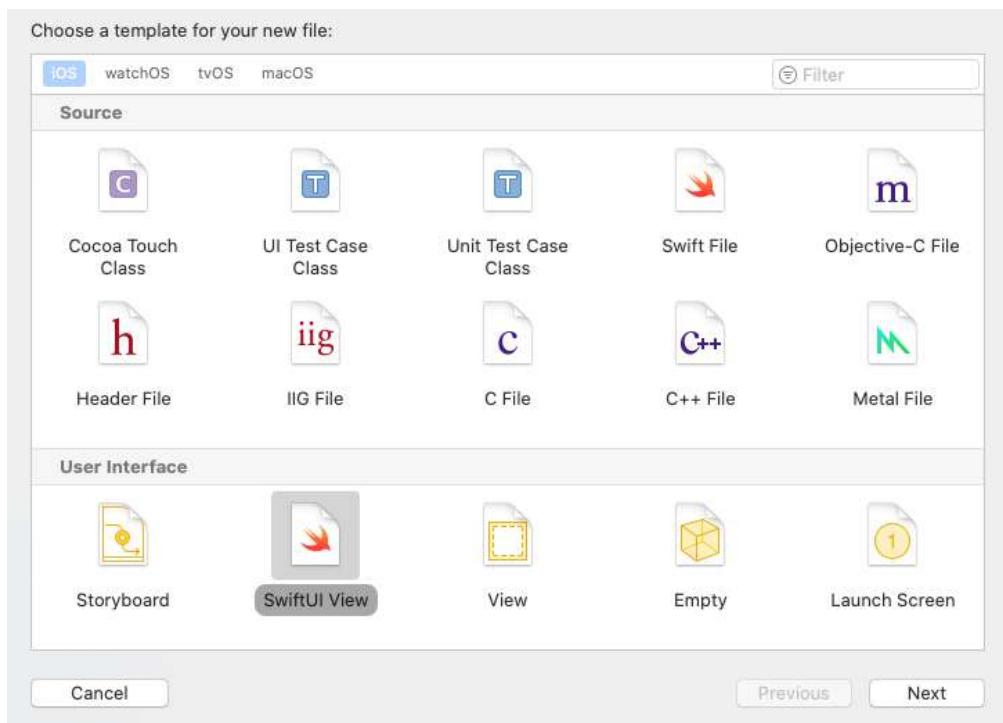
`ContentView` is the name that Xcode assigns to the single view (or screen) when it creates a single view app. When Xcode created it, it also created the file that contains it: `ContentView.swift`.

Xcode makes it easy to create additional views and their containing files. Let's

Xcode automatically created the main `ViewController` object for you. But you'll have to create the view controller for the About screen yourself. Fortunately, it's pretty easy to do this.

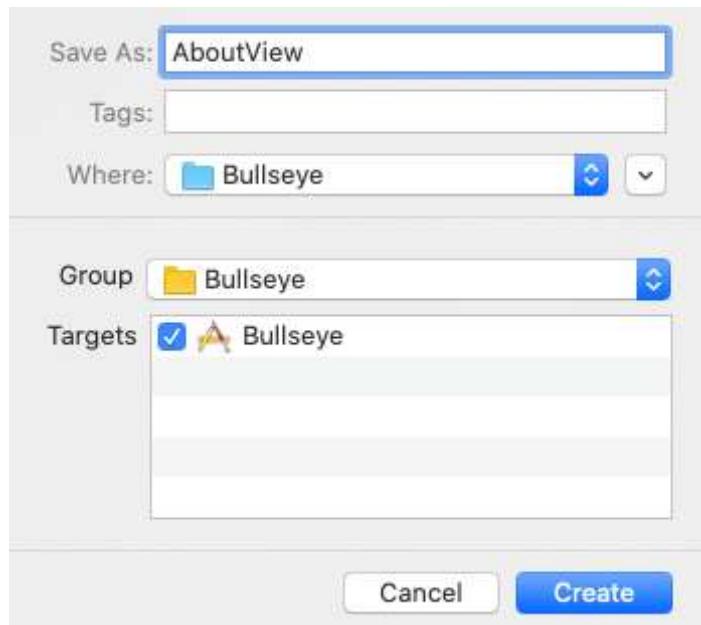
Adding a new view

- Go to Xcode's **File** menu and choose **New ➤ File....** In the window that pops up, choose the **SwiftUI Views** template (if you don't see it then make sure **iOS** is selected at the top).



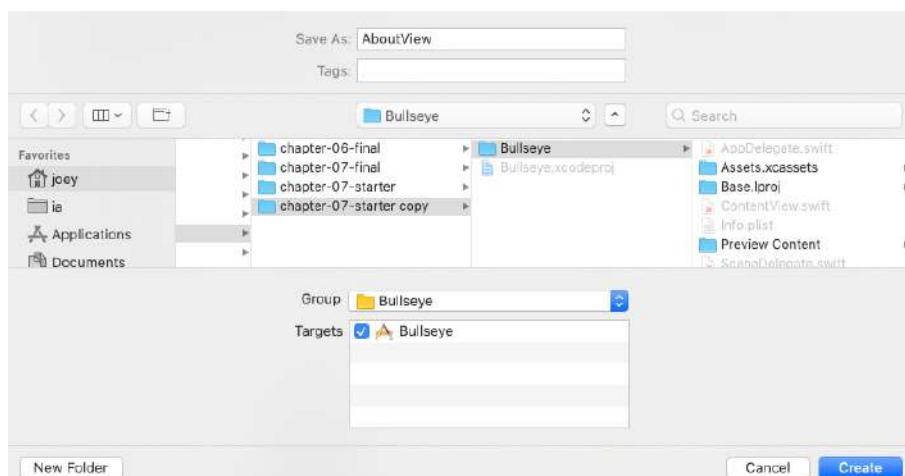
Choosing the file template for SwiftUI View

- Click **Next**. Xcode will ask you what to name this new view file and where to save it. You'll either see this...



The options for the new file

...or this:



The options for the new file

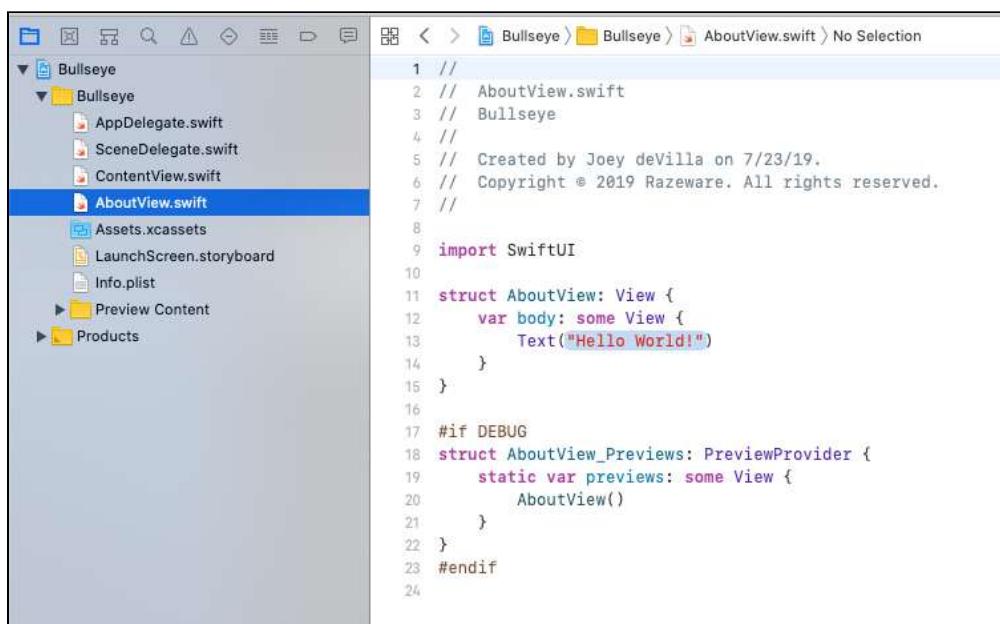
► In either case, change the contents of the **Save As:** field to **AboutView**, then click the **Create** button.

► Choose the **Bullseye** folder (this folder should already be selected).

Also make sure **Group** says **Bullseye** and that there is a checkmark in front of **Bullseye** in the list of **Targets**.

► Click **Create**.

Xcode will create a new file and add it to your project. As you might have guessed, the new file is **AboutView.swift**. Xcode will show you the contents of that new file. You should have a sense of *deja vu*: this is what **ContentView.swift** looked like at the start of Chapter 2. You've come a long way:



The screenshot shows the Xcode interface with the project navigation bar at the top. Below it, the project structure is shown in the left sidebar, with the 'Bullseye' group expanded to show files like AppDelegate.swift, SceneDelegate.swift, ContentView.swift, and the newly created AboutView.swift. The right pane displays the code for AboutView.swift. The code is as follows:

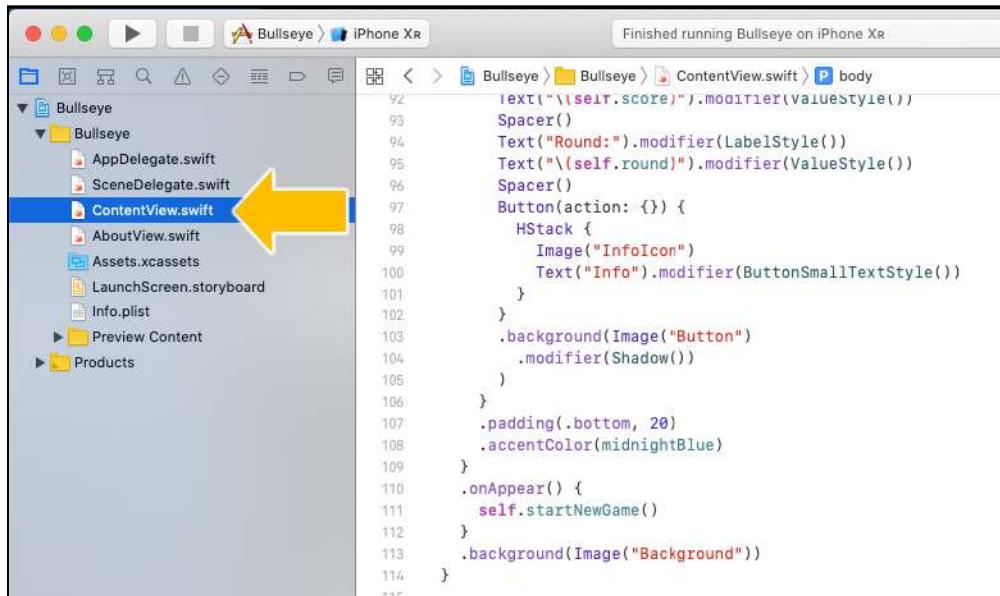
```
1 //  
2 //  AboutView.swift  
3 //  Bullseye  
4 //  
5 //  Created by Joey DeVilla on 7/23/19.  
6 //  Copyright © 2019 Razeware. All rights reserved.  
7 //  
8  
9 import SwiftUI  
10  
11 struct AboutView: View {  
12     var body: some View {  
13         Text("Hello World!")  
14     }  
15 }  
16  
17 #if DEBUG  
18 struct AboutView_Previews: PreviewProvider {  
19     static var previews: some View {  
20         AboutView()  
21     }  
22 }  
23 #endif  
24
```

The newly-created *AboutView*

Connecting the “Info” button to AboutView

It's time to make the **Info** button on **ContentView** do its thing!

- Switch back to editing **ContentView.swift** by clicking on it in the Project Navigator:



```
Finished running Bullseye on iPhone XR
Bullseye > Bullseye > ContentView.swift > body
92     text("\(self.score)").modifier(valueStyle())
93     Spacer()
94     Text("Round:").modifier(LabelStyle())
95     Text("\(self.round)").modifier(ValueStyle())
96     Spacer()
97     Button(action: {}) {
98         HStack {
99             Image("InfoIcon")
100            Text("Info").modifier(ButtonSmallTextStyle())
101        }
102        .background(Image("Button"))
103        .modifier(Shadow())
104    }
105    }
106    }
107    .padding(.bottom, 20)
108    .accentColor(midnightBlue)
109    }
110    .onAppear() {
111        self.startNewGame()
112    }
113    .background(Image("Background"))
114 }
```

Back to ContentView

The simplest way to navigate between views is to make use of a **NavigationView**. It's a special kind of view with just one purpose: To make it simple to navigate back and forth between other views.

We're going to take **ContentView** and put it inside a **NavigationView**. Doing this causes a couple of things to happen automatically:

- It sets up a *Navigation Bar* at the top of the view. This can house buttons that allow the user to easily navigate between views.
- It sets up **ContentView** so that it's easy to navigate to other views. It also returns back to **ContentView** with a **Back** button that appears in the navigation bar.

Let's add a **NavigationView** to **ContentView**.

- Scroll to the start of ContentView's body property and select everything starting with VStack and ending with the .background(Image("Background")). The start of your selection should look like this:

```

32 // User interface content and layout
33 var body: some View {
34     VStack {
35         Spacer()
36
37         // Target row
38         HStack {
39             Text("Put the bullseye as close as you can to!").modifier(LabelStyle())
40             Text("\(self.target)").modifier(ValueStyle())
41         }
42
43         Spacer()
44
45         // Slider row
46         HStack {
47             Text("1").modifier(LabelStyle())
48             Slider(value: self.sliderValue, from: 1.0, through: 100.0)
49             .accentColor(.green)
50             Text("100").modifier(LabelStyle())
51         }
52
53         Spacer()
54     }
55 }
```

The start of your selection

And the end of your selection should look like this:

```

77 // Score row
78 HStack {
79     Button(action: {
80         self.startNewGame()
81     }) {
82         HStack {
83             Image("StartOverIcon")
84             Text("Start over").modifier(ButtonSmallTextStyle())
85         }
86         .background(Image("Button"))
87         .modifier(Shadow())
88     }
89     Spacer()
90     Text("Score:").modifier(LabelStyle())
91     Text("\(self.score)").modifier(ValueStyle())
92     Spacer()
93     Text("Round:").modifier(LabelStyle())
94     Text("\(self.round)").modifier(ValueStyle())
95     Spacer()
96     Button(action: {}) {
97         HStack {
98             Image("InfoIcon")
99             Text("Info").modifier(ButtonSmallTextStyle())
100        }
101        .background(Image("Button"))
102        .modifier(Shadow())
103    }
104    .padding(.bottom, 20)
105    .accentColor(midnightBlue)
106 }
107 .onAppear() {
108     self.startNewGame()
109 }
110 .background(Image("Background"))
111 }
112 }
113 }
```

The end of your selection

- With that code still selected, press ⌘+] to indent your selection one level.

- Scroll to the start of body and add a NavigationView so that it looks like this:

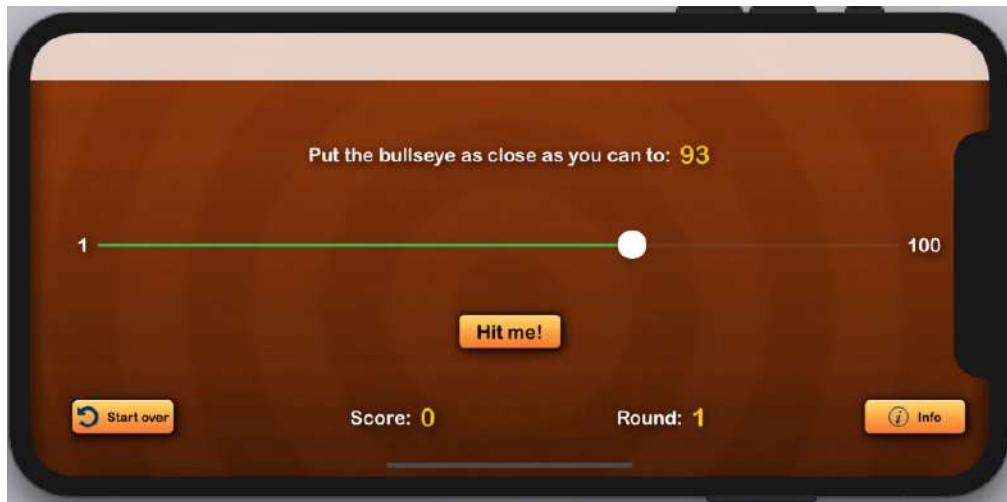
```
// User interface content and layout
var body: some View {
    NavigationView {
        VStack {
            Spacer()

            // Target row
            ...
        }
    }
}
```

- Scroll to the end of body and close theNavigationView with a closing brace and a couple of methods. The end result should look like this:

```
.onAppear() {
    self.startNewGame()
}
.background(Image("Background"))
.navigationViewStyle(.stack)
}
```

- Run the app. It now displays a navigation bar at the top of the screen:



The navigation bar appears

By putting ContentView inside a NavigationView, it's now possible to make use of controls to take the user to a different view. We're going to replace the Button that was used for **Info** and replace it with a **NavigationLink**.

The `NavLink` link won't be all that different from a `Button`. It will still contain an `HStack` the button icon and text, and it will still use the button background image. However, instead of giving it code to perform when it's pressed, you specify a destination view.

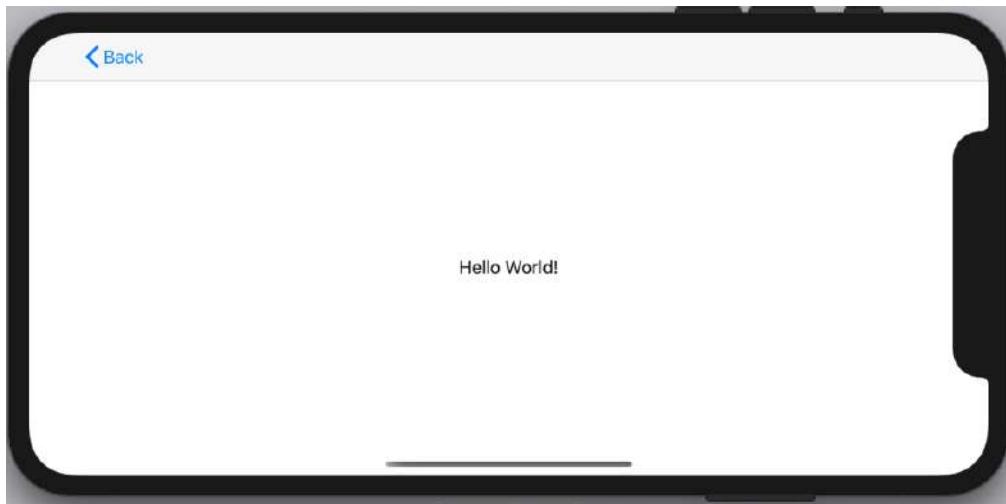
► Go to the *Score row* section of `ContentView`'s `body` property and change this line...

```
Button(action: {}) {
```

...to this:

```
// Score row
HStack {
    Button(action: {
        self.startNewGame()
    }) {
        HStack {
            Image("StartOverIcon")
            Text("Start over").modifier(ButtonSmallTextStyle())
        }
    }
    .background(Image("Button"))
    .modifier(Shadow())
}
Spacer()
Text("Score:").modifier(LabelStyle())
Text("\(self.score)").modifier(ValueStyle())
Spacer()
Text("Round:").modifier(LabelStyle())
Text("\(self.round)").modifier(ValueStyle())
Spacer()
NavigationLink(destination: AboutView()) {
    HStack {
        Image("InfoIcon")
        Text("Info").modifier(ButtonSmallTextStyle())
    }
}
.background(Image("Button"))
.modifier(Shadow())
}
.padding(.bottom, 20)
.accentColor(midnightBlue)
```

- Run the app and press **Info**. You'll be taken to **AboutView**, which will look like this:



AboutView

- Press the **Back** button in the navigation bar. You'll be returned back to **ContentView**.

Now that the player can navigate between views, it's time to fill **AboutView**.

- Switch to **AboutView.swift** in Xcode and change **AboutView**'s **body** property to the following:

```
var body: some View {
    VStack {
        Text("🎯 Bullseye 🎯")
        Text("This is Bullseye, the game where you can win points and earn fame by dragging a slider.")
        Text("Your goal is to place the slider as close as possible to the target value. The closer you are, the more points you score.")
        Text("Enjoy!")
    }
}
```

In case you've forgotten, the keyboard command to enter emojis is **control+⌘+space**. You can then find the **🎯** character by typing **bullseye** into the emoji pop-up's search text field.

- Run the app and press **Info**. AboutView now contains the proper text, but the formatting needs work:



Let's improve the formatting with a couple of **ViewModifiers**. We'll make one for the heading, and one for the body text beneath it.

- Add the following between `AboutView` and the preview section:

```
// View modifiers
// =====

struct AboutHeadingStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(Font.custom("Arial Rounded MT Bold", size: 30))
            .foregroundColor(Color.black)
            .padding(.top, 20)
            .padding(.bottom, 20)
    }
}

struct AboutBodyStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(Font.custom("Arial Rounded MT Bold", size: 16))
            .foregroundColor(Color.black)
            .padding(.leading, 60)
            .padding(.trailing, 60)
            .padding(.bottom, 20)
    }
}
```

These are similar to the `ViewModifiers` in `ContentView`. The only significant difference is that these include some padding for spacing between the heading and paragraphs.

Now that there are some `ViewModifiers`, it's time to apply them to the text.

- Change `AboutView`'s `body` property to the following:

```
var body: some View {
    VStack {
        Text("🎯 Bullseye 🎯")
            .modifier(AboutHeadingStyle())
        Text("This is Bullseye, the game where you can win points
and earn fame by dragging a slider.")
            .modifier(AboutBodyStyle())
        Text("Your goal is to place the slider as close as possible
to the target value. The closer you are, the more points you
score.")
            .modifier(AboutBodyStyle())
        Text("Enjoy!")
            .modifier(AboutBodyStyle())
    }
}
```

- Run the app and press **Info**. We're getting closer...:

That second paragraph — the one that begins with “Your goal is to place the slider as close as possible to the target value” keeps getting cut off. We want it to display the full text.

Text views, it turns out, display a single line by default. Any text that goes beyond a single line is cut off and replaced with an ellipsis (the “...”). This default setting can be overridden with the `lineLimit()` method, which lets you specify the maximum number of lines the `Text` view will display. You can also give `lineLimit()` a value of `nil`, which means “no limit”. That’s what we’ll use for the first and second paragraphs.

- Change `AboutView`'s `body` property to the following:

```
var body: some View {
    VStack {
        Text("🎯 Bullseye 🎯")
            .modifier(AboutHeadingStyle())
        Text("This is Bullseye, the game where you can win points
and earn fame by dragging a slider.")
            .modifier(AboutBodyStyle())
            .lineLimit(nil)
    }
}
```

```
        Text("Your goal is to place the slider as close as possible  
        to the target value. The closer you are, the more points you  
        score.")  
        .modifier(AboutBodyStyle())  
        .lineLimit(nil)  
    Text("Enjoy!")  
    .modifier(AboutBodyStyle())  
}  
}
```

- Run the app and press **Info**. Now *all* the text is displayed:

The clever reader may ask “Why did we attach make two separate calls to `lineLimit()` with two different `Text` views? Wouldn’t it be more DRY to put *one* call to `lineLimit()` from within `AboutBodyStyle`? ”

If you asked this question, you should congratulate yourself. Under normal circumstances, you’d be right. Unfortunately, as of this writing (we’re using Xcode 11 beta 4), `lineLimit()` seems to work only if you call it directly from the object you want to apply it to, and not from within a `ViewModifier`. This may change as newer versions of Xcode come out.

There are only a couple of tasks left. We need to create a plain beige background for the text, and behind that, we’ll use the same background image as `ContentView`.

The first step is to create a custom beige color. It’s like creating the midnight blue color, just with different values for red, green, and blue.

- Add the following to `AboutView` *above* the `body` property:

```
// Constants  
let beige = Color(red: 1.0,  
                  green: 0.84,  
                  blue: 0.70)
```

Now that we have the beige color defined, let’s make it the background of the `VStack` that holds all the `Text` views.

- Change `AboutView`’s `body` property to the following:

```
var body: some View {  
    VStack {  
        Text("🎯 Bullseye 🎯")  
        .modifier(AboutHeadingStyle())
```

```
Text("This is Bullseye, the game where you can win points  
and earn fame by dragging a slider.")  
    .modifier(AboutBodyStyle())  
    .lineLimit(nil)  
Text("Your goal is to place the slider as close as possible  
to the target value. The closer you are, the more points you  
score.")  
    .modifier(AboutBodyStyle())  
    .lineLimit(nil)  
Text("Enjoy!")  
    .modifier(AboutBodyStyle())  
}  
.background(beige)  
}
```

- Run the app and press **Info**. The `VStack` is now visible as a beige rectangle. It's large enough to accommodate the views it contains, complete with padding:

We now need some kind of view whose only purpose is to act as a container for the background image. There's a type of `View` called `Group`, and it's used to group views together. It also expands to fill the view which contains it, which would be the entire screen. Let's put the `VStack` inside a `Group`, and then set the `Group`'s background to the background image.

- In the body property, select everything starting with `VStack` and ending with the `.background(beige)`. Your selection should look like this...

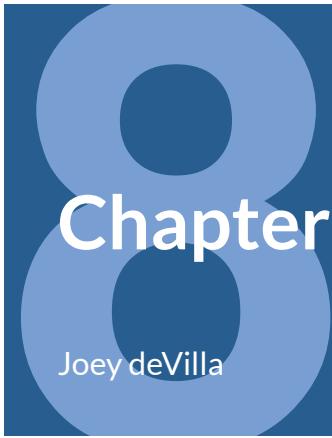
```
var body: some View {  
    Group {  
        VStack {  
            Text("🎯 Bullseye 🎯")  
                .modifier(AboutHeadingStyle())  
            Text("This is Bullseye, the game where you can win points  
and earn fame by dragging a slider.")  
                .modifier(AboutBodyStyle())  
                .lineLimit(nil)  
            Text("Your goal is to place the slider as close as  
possible to the target value. The closer you are, the more  
points you score.")  
                .modifier(AboutBodyStyle())  
                .lineLimit(nil)  
            Text("Enjoy!")  
                .modifier(AboutBodyStyle())  
        }  
        .background(beige)  
    }.background(Image("Background"))  
}
```

► Run the app and press **Info**. It looks like we've made it:

Congrats! This completes the game. All the functionality is there and – as far as I can tell – there are no bugs to spoil the fun.

You can find the project files for the finished app under **07 - The New Look** in the Source Code folder.





Chapter 8: The Final App

Joey deVilla

You might be thinking, “Okay, *Bullseye* is now done and I can move on to the next app!” If you were, I’m afraid that you’re in for some disappointment — there’s just a teensy bit more to do in the game.

“But the task list is complete!” you might say, and you’d be right. It’s just that software has a way of finding more things for you to do. In this chapter, we’ll add a few more touches to *Bullseye* to make it truly polished. One of these touches is absolutely necessary for apps to be published in the App Store. And finally, there’s the matter of trying out *Bullseye* on a real device instead of the Simulator.

Don’t worry; you’re almost done!

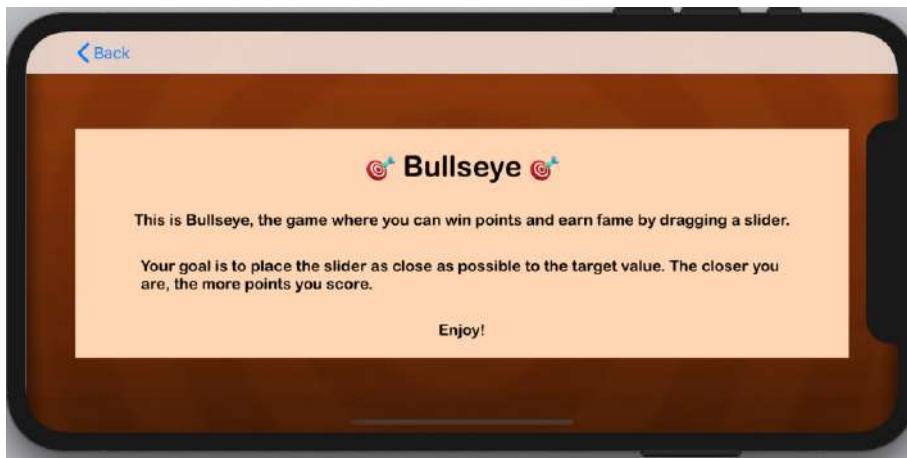
Here are the specific items covered in this chapter:

- **Including animation:** Add some animation to make the start of a new round or game a bit more dynamic.
- **Adding an icon:** Giving the app its own distinctive icon, replacing the default blank one.
- **Display name:** Set the display name — the one users see on the home screen — for the app.
- **Running the app on a device:** How to configure everything to run your app on an actual device.



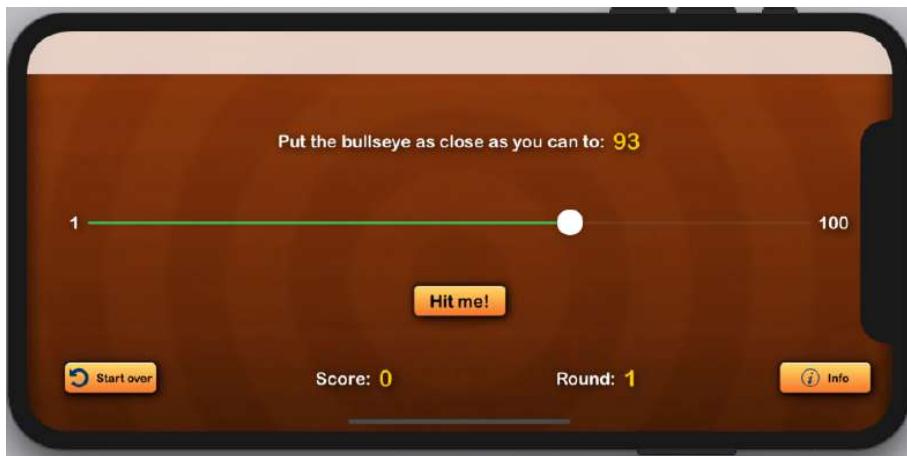
Adding a title to the navigation bar

The `NavigationView` object that *Bullseye* uses to take the user from the main screen to the “About” screen and back adds a translucent navigation bar that runs across the top of the screen. This bar gives the user a visual hint that the app has more than one screen and also provides a place for navigation controls, such as the **Back** button that automatically appears on the “About” screen:



The ‘About’ screen, with the navigation bar displaying a ‘Back’ button

The navigation bar seems a little less useful on the main screen. There, it’s an empty translucent strip that gives the user the impression that the developer — that’s *you!* — didn’t quite finish working on the user interface:



The main screen with a blank navigation bar

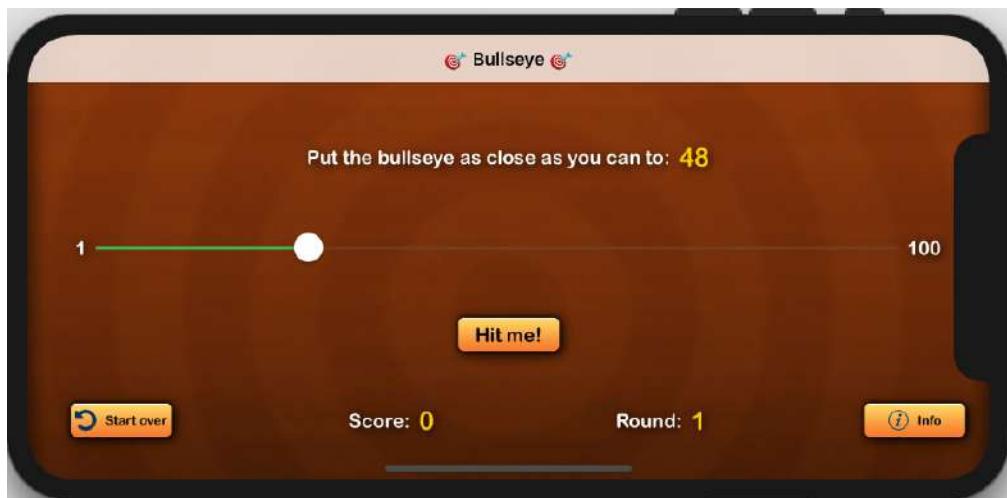
We can solve this problem and do a little app marketing at the same time by putting a title into the navigation bar when it's on the main screen. We can do this with a method available to any view object called `navigationBarTitle()`, which accepts a `Text` object to set text that appears in the center of the navigation bar.

The `navigationBarTitle()` method can be called from any view inside the `NavigationView`. To make it easy to see that the `NavigationView` is getting a title, we'll call `navigationBarTitle()` from the first `Spacer` in the main screen's `NavigationView`.

- Change the start of `ContentView`'s body variable to the following:

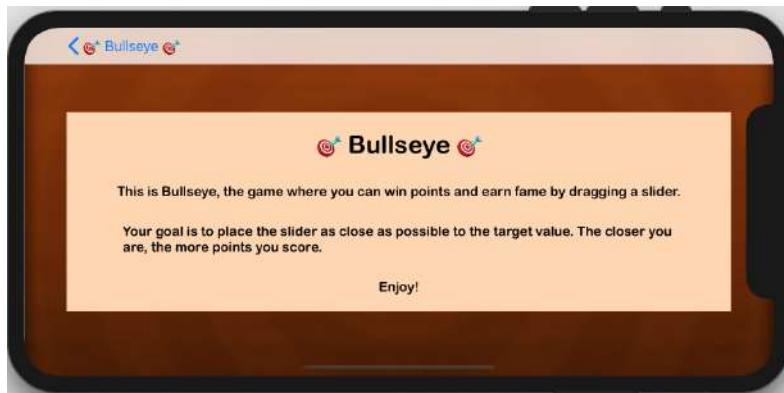
```
var body: some View {  
    NavigationView {  
        VStack {  
            Spacer().navigationBarTitle("🎯 Bullseye 🎯")  
            ...  
        }  
    }  
}
```

- Run the app. You should see a title in the formerly blank navigation bar:



The main screen with a navigation bar with a title

When you navigate away from a page with navigation bar title, the “Back” button on the destination page displays the name of the page you just left:



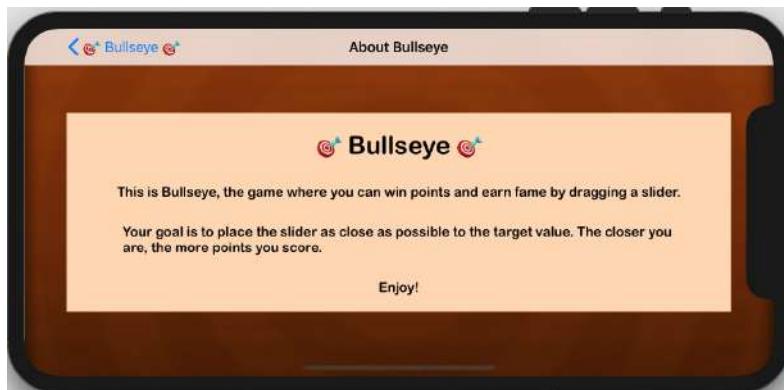
The 'About' screen with a 'Bullseye' back button in the navigation bar

We can also add a navigation bar title to the “About” page. Once again, it’s a matter of calling the `navigationBarTitle()` method from any of its views. We’ll cakk it from the first `Text` element on that screen.

- Change the start of `AboutView`’s `body` variable to the following:

```
var body: some View {  
    Group {  
        VStack {  
            Text("🎯 Bullseye 🎯")  
                .modifier(AboutHeadingStyle())  
                .navigationBarTitle("About Bullseye")  
        }  
    }  
}
```

- Run the app and press the **Info** button. The “About” page will now have a title:



The 'About' screen with a navigation bar title

Including animation

There's one last improvement that you can add to the app: Animation. iOS contains a technology called Core Animation that makes it very easy to add animation effects to your app's views. With Core Animation, you can get several different kinds of animation with very little code. SwiftUI gives you access to Core Animation's power and, with it, you can add subtle animations (with an emphasis on *subtle!*) that can make your app a delight to use.

Right now, when a new round or game begins, the slider simply snaps to its randomly-determined position. You'll add an animation to the slider so that it *slides* to that randomly-determined position instead.

- In the *Slider row* section of `ContentView`'s body property, change the line that defines the slider to look like this:

```
Slider(value: $sliderValue, in: 1...100)
    .accentColor(Color.green)
    .animation(.easeOut)
```

- Run the app. You can either play a few rounds or, if you're feeling impatient, you can simply press the **Start over** button over and over again. Either way, you'll see the slider slide to its new random position.

The call to the slider's `animation()` method added the animation effect to the slider. It takes a parameter that specifies the kind of animation that should be used. In this case, you provided a parameter called `.easeOut`, which is an “ease out” animation. “Ease out” means that as the slider approaches its destination it slows down a little, making its motion look more natural.

The `animation()` method is called whenever the app changes the position of the slider, which in turn happens when the app changes the value of the `sliderValue` property. This means that the animation takes place whenever the game starts, at the start of a new round, and at the start of a new game.

Adding an icon

You're almost done with the app, but there are still a few loose ends to tie up. You may have noticed that the app has a really boring white icon. That won't do!



- Open the asset catalog (**Assets.xcassets**) and select **AppIcon**:

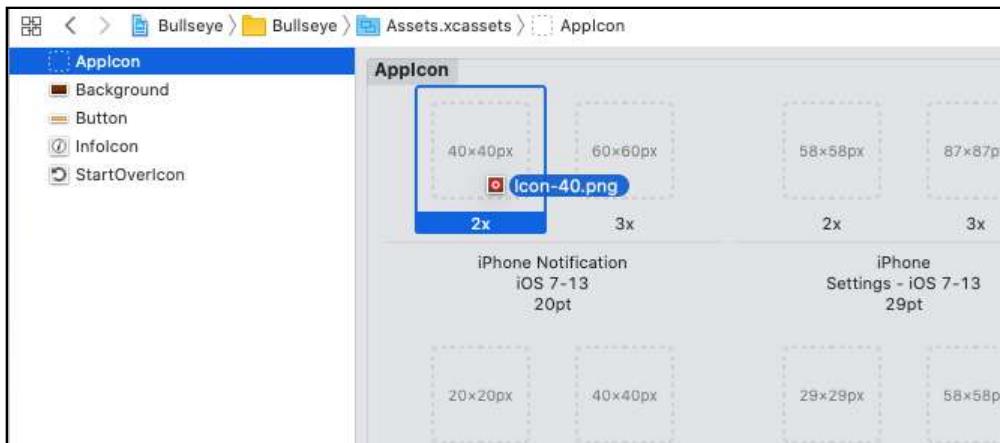


The AppIcon group in the asset catalog

It turns out that there isn't just one icon for an app, but 18 of them, to account for various device screen resolutions and different uses. They range in size from a minuscule 20 by 20 pixels for use in notifications to an enormous 1024 by 1024 pixels for the App Store.

Importing icons is like importing images — it's a drag-and-drop process.

- In Finder, open the **Resources** folder for this book and then open the **Icon** folder. Drag the **Icon-40.png** file into the **2x** slot in the area marked **iPhone Notification / iOS 7 - 13 / 20pt**:



Dragging the icon into the asset catalog

You've probably figured out that the numbers in the filenames in the **Icon** folder indicate their dimensions in pixels. The **Icon-40.png** file is a 40-pixel by 40-pixel icon. So why are you dragging the **Icon-40.png** file into the slot marked **20pt** and not **Icon-20.png**?

The reason is that the filenames specify the icon size in *pixels* and the slots specify the icon size in *points*. On pre-Retina iPhones, one point was equal to one pixel. Retina iPhones and iPads have twice the pixel density. So, on those devices, one point is equal to *two* pixels. The Retina HD iPhones — the X, XS, XS Max, and any iPhone with a “+” in its name — have three times the pixel density. For them, one point is equal to *three* pixels.

Also, if you look at each of the slots, they've conveniently put the pixel dimensions in each one.

- Drag the **Icon-60.png** file into the **3x** slot next to it. This is for the iPhone Plus devices with their **3x** resolution.
- For the **iPhone / Settings - iOS 7 - 13 / 29pt** slots, drag the **Icon-58.png** file into the **2x** slot and **Icon-87.png** into the **3x** slot. What, you don't know your times table for 29?
- For the **iPhone Spotlight / iOS 7 - 13 / 40pt** slots, drag the **Icon-80.png** file into the **2x** slot and **Icon-120.png** into the **3x** slot.
- For the **iPhone App / iOS 7 - 13 / 60pt** slots, drag the **Icon-120.png** file into the **2x** slot and **Icon-180.png** into the **3x** slot.

That's four icons in two different sizes. Phew!

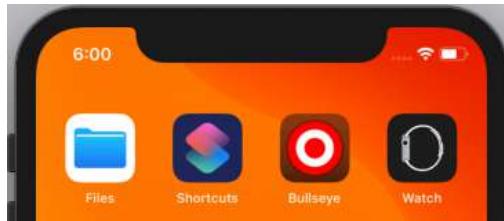
The other AppIcon groups are mostly for the iPad.

- Drag the specific icons — based on size — into the proper slots for iPad. Notice that the iPad icons need to be supplied in **1x** and **2x** sizes but not **3x**. You can either do the math to match pixel and point sizes or use the points specified in each slot to figure out which icons need to go in which iPad slots.



The full set of icons for the app

- Run the app and close it. You'll see that the icon has changed on the Simulator's springboard. If not, remove the app from the Simulator and try again (sometimes the Simulator keeps using the old icon and re-installing the app will fix this).



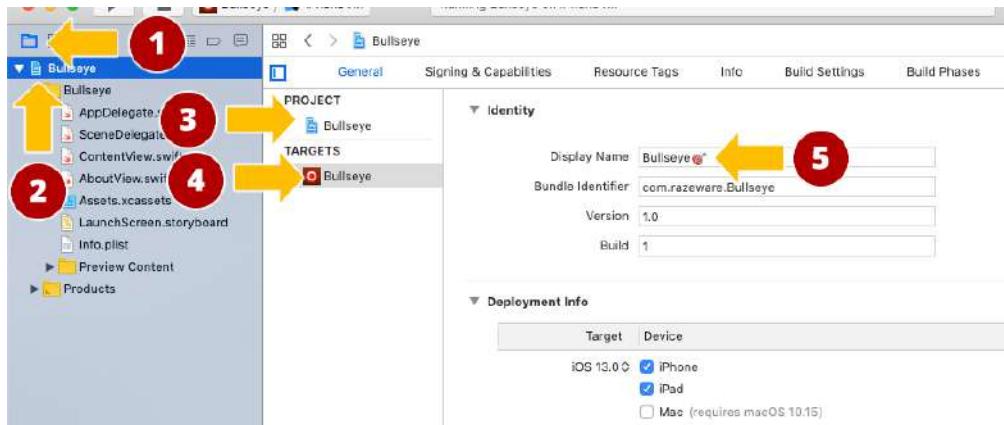
The icon on the Simulator's springboard

Display name

The text below the app's icon on the Home screen is its *display name*. It can be different from the project name, and it often *has* to be — there's a limited amount of space under any app's icon, and display names can be only one line long.

You started with the project with the name **Bullseye**. Xcode automatically used it as the app's display name. Sometimes, as you work on a project, you'll come up with a better one. Let's pretend that this happened, and in a fit of inspiration, you decided to change the name of the app to **Bullseye🎯** (note the “bullseye” emoji added at the end).

- Go to the **Project Settings** screen. The very first option is **Display Name**. Change this to **Bullseye🎯**:

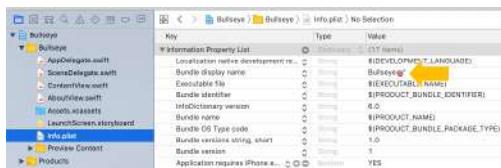


Changing the display name of the app

In case you've forgotten, you access the macOS emoji keyboard by pressing **ctrl + ⌘ + space**.

As with many of your project's settings, you can also find the display name in the app's **Info.plist** file. Let's take a look.

- From the **Project navigator**, select **Info.plist**.

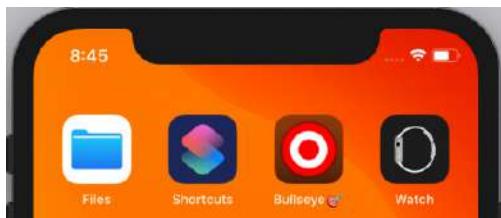


The display name of the app in Info.plist

The row **Bundle display name** contains the new name you've just entered.

Note: If **Bundle display name** is not present, the app will use the value from the field **Bundle name**. That has the special value “\$(PRODUCT_NAME)”, meaning Xcode will automatically put the project name, BullsEye, in this field when it adds the Info.plist to the application bundle. By providing a **Bundle display name** you can override this default name and give the app any name you want.

- Run the app and quit it to see the new name under the icon.



The bundle display name setting changes the name under the icon

Guess what — your very first app is complete!

You can find the project files for the finished app under **08 - The Final App** in the **Source Code** folder.

Running the app on a device

So far, you've run the app on the Simulator. That's nice, but you probably didn't take up learning iOS development simply to make apps for pretend devices. You want to make apps that run on real iPhones and iPads and even distribute them in the App Store! There's hardly anything more satisfying than seeing an app that you made running on your own device — except showing off the fruits of your labor to other people!

Don't get me wrong: Developing your apps on the Simulator works very well. When developing, it's very convenient to spend most of your time with the Simulator and only test the app on a device every so often.

However, you *do* need to run your creations on a real device in order to test them properly. There are some things the Simulator simply can't do. For example, if your app needs the iPhone's accelerometer, you have no choice but to test that functionality on an actual device. You can't just sit there and shake your Mac! (Well, you *can*, but it'll have no effect.)

In the past, you needed a paid Developer Program account to run apps on your iPhone. These days, you can do it for free. All you need is an Apple ID, and the latest Xcode makes it easier than ever before.

Configuring your device for development

- ▶ Connect your iPhone, iPod touch or iPad to your Mac using a USB cable.
- ▶ From the Xcode menu bar select **Window ▶ Devices and Simulators** to open the **Devices and Simulators** window. Yours will look similar to the one shown below:



The Devices and Simulators window

The left column contains a list of devices that are currently connected to your Mac and which can be used for development.

- Click on a device name in the left column to select it.

If this is the first time you're using the selected device with Xcode, the window will show a message that says something like: "iPhone is not paired with your computer." To pair the device with Xcode, you'll need to unlock the device first. Once you've unlocked it, an alert will pop up on the device asking you to trust the computer you're trying to pair with. Tap on **Trust** to continue.

Xcode will now refresh the page and let you use the device for development. Give it a few minutes and see the progress bar in the main Xcode window. If it takes too long, you may need to unplug the device and plug it back in.

At this point it's possible you may get the error message: "An error was encountered while enabling development on this device." You'll need to unplug the device and reboot it. Make sure to restart Xcode before you reconnect the device.

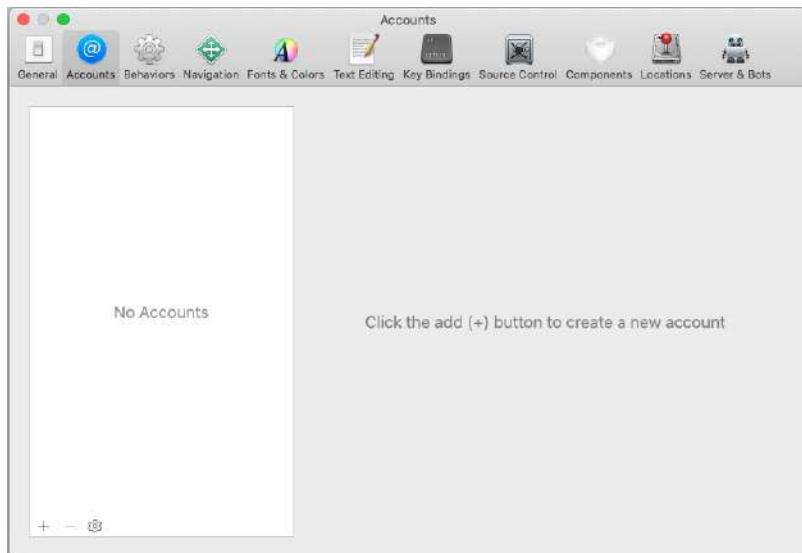
Also, note the checkbox that says **Connect via network?** That checkbox (gasp!) allows you to deploy and debug code on your iPhone over WiFi!

That takes care of deploying to your device — cool!

Adding your developer account to Xcode

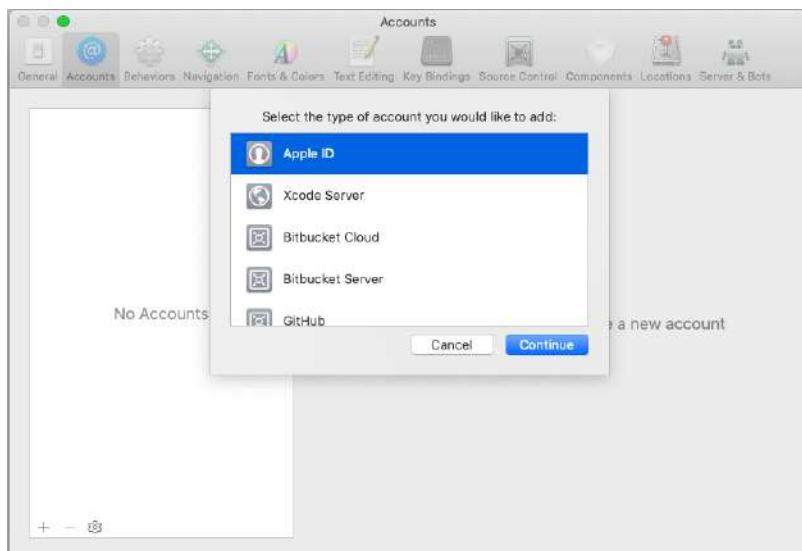
The next step is setting up your Apple ID with Xcode. It's okay to use the same Apple ID that you're already using with iTunes and your iPhone for hobby projects. However, if you run a business you might want to create a new Apple ID strictly for that purpose. Of course, if you've already registered for a paid Developer Program account, you should use that Apple ID.

- Open the **Accounts** pane in the Xcode Preferences window:



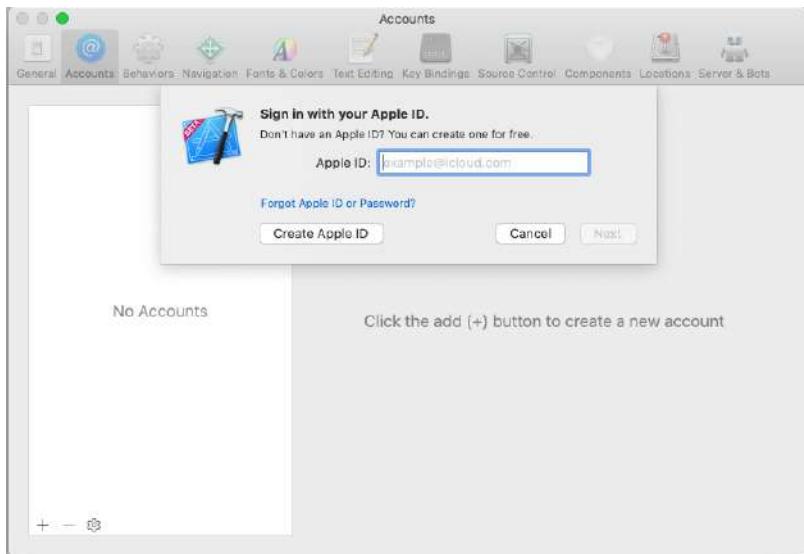
The Accounts preferences

- Click the + button at the bottom, select **Add Apple ID** from the list of options and click **Continue**:



Xcode Account Type selection

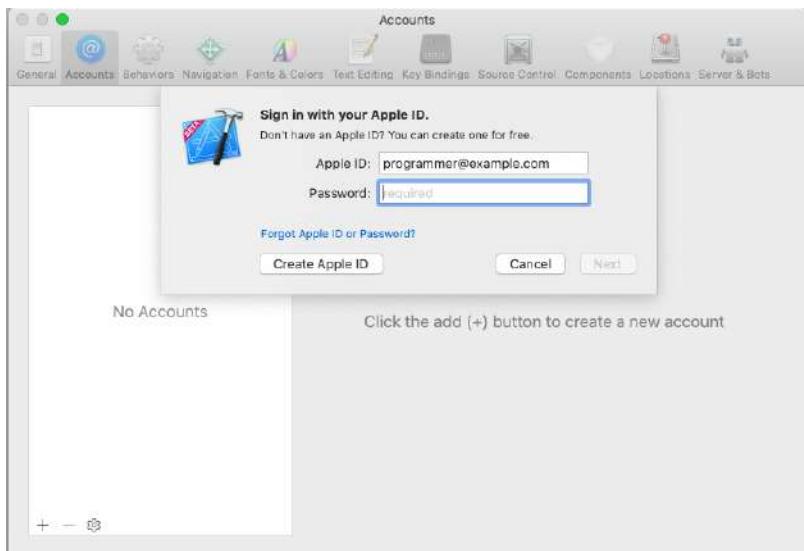
Xcode will ask for your Apple ID:



Adding your Apple ID to Xcode

- Type your Apple ID username and click **Next**.

Xcode will ask for your Apple ID password:



Entering your Apple ID password

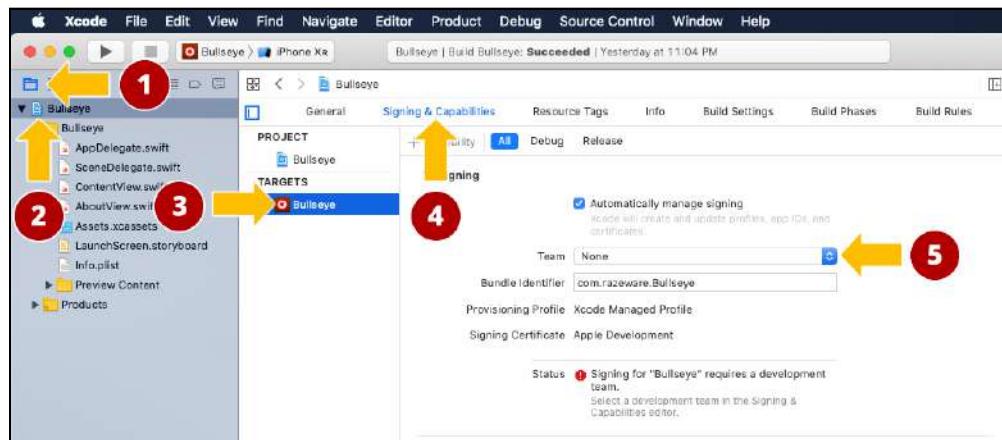
Xcode verifies your account details and adds it to the stored list of accounts.

Note: It's possible that Xcode is unable to use the Apple ID you provided. For example, if it has been used with a Developer Program account in the past that is now expired. The simplest solution is to make a new Apple ID. It's free and only takes a few minutes. appleid.apple.com

You still need to tell Xcode to use this account when building your app.

Code signing

- Go to the **Project Settings** screen for your app target. In the **General** tab go to the **Signing & Capabilities** section.



The Signing options in the Project Settings screen

In order to allow Xcode to put an app on your device, the app must be *digitally signed* with your **Development Certificate**. A *certificate* is an electronic document that identifies you as an iOS application developer and is valid only for a specific amount of time. Creating and using a development certificate is free.

Apps that you want to submit to the App Store must be signed with another certificate, the **Distribution Certificate**. To create and use a distribution certificate, you must be a member of the paid Developer Program.

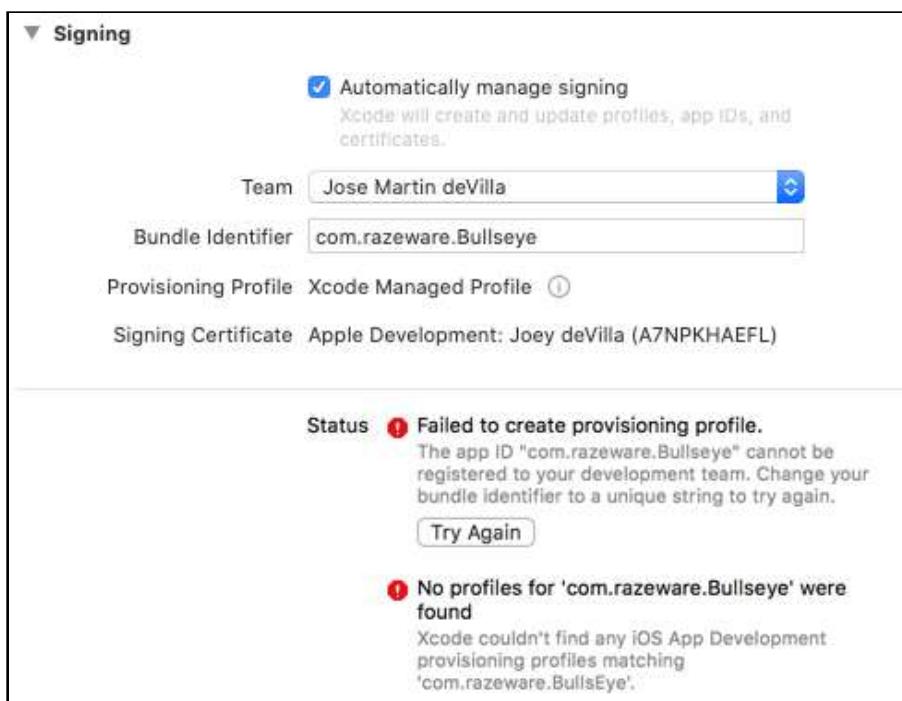
In addition to having a valid certificate, you also need a **Provisioning Profile** for each app you make. Xcode uses this profile to sign the app for use on your particular device or devices. The specifics don't really matter. Just know that you need a provisioning profile or the app won't be installed on your device.

Making the certificates and provisioning profiles used to be a really frustrating and error-prone process. Fortunately, those days are over: Xcode now makes it really easy. When the **Automatically manage signing** option is enabled, Xcode will take care of all this business with certificates and provisioning profiles and you don't have to worry about a thing.

- Click on **Team** to select your Apple ID.

Xcode will now automatically register your device with your account, create a new Development Certificate and download and install the Provisioning Profile on your device. These are all steps you had to do by hand in the past, but now Xcode takes care of all that.

You could get some signing errors like these:



Signing/team set up errors

The app's **Bundle Identifier** must be unique. If another app is already using that identifier, your app can't use it. That's why it's suggested that you start the Bundle ID with your own domain name. The fix is easy: change the Bundle Identifier field to something else and try again.

It's also possible you get this error (or something similar):



No devices registered

Xcode must know about the device that you're going to run the app on. That's why you were told to connect your device first. Double-check that your device is still connected to your Mac and that it is listed in the Devices window.

Running the app on your device

If everything goes smoothly, go back to Xcode's main window and click on the dropdown in the toolbar to change where you will run the app. The name of your device should be in that list somewhere. It should look something like this:



Setting the active device

You're all set and ready to go!

- Tap **Run** to launch the app.

At this point, you may get a pop-up with the question: “Codesign wants to sign using key... in your keychain.” If so, answer with **Always Allow**. This is Xcode trying to use the new Development Certificate you just created — you just need to give it permission first.

Does the app work? Awesome! If not, read on...

When things go wrong...

There are a few things that can go wrong when you try to put the app on your device, especially if you've never done this before, so don't panic if you run into problems.

The device is not connected

Make sure your iPhone, iPod touch or iPad is connected to your Mac. The device must be listed in Xcode's Devices window and there should not be a yellow warning icon.

The device does not trust you

You might get this warning:



Quick, call security!

On the device itself, there will be a pop-up with the text: “Untrusted Developer. Your device management settings do not allow using apps from developer...”

If this happens, open the Settings app on the device and go to **General ▶ Profile**. Your Apple ID should be listed in that screen. Tap it, followed by the **Trust button**. Then, try running the app again.

The device is locked

If your phone locks itself with a passcode after a few minutes, you might get this warning:



The app won't run if the device is locked

Simply unlock your device by holding the home button, typing in the 4-digit passcode or using FaceID and tap **Run** again.

Signing certificates

If you're curious about these certificates, then open the **Preferences** window and go to the **Accounts** tab. Select your account and click the **Manage Certificates...** button in the bottom-right corner.

This brings up another panel, listing your signing certificates:



The Manage Certificates panel

When you're done, close the panel and go to the **Devices and Simulators** window. You can see the provisioning profiles that are installed on your device by right-clicking the device name and choosing **Show Provisioning Profiles**.

Provisioning profiles installed on Fahim's iPhone 6s:	
Name	Expiration Date
iOS Dev	23 Dec 2017
iOS Team Provisioning Profile: *	22 Mar 2018
iOS Team Provisioning Profile: [REDACTED]	22 May 2018
iOS Team Provisioning Profile: [REDACTED]	22 May 2018
iOS Team Provisioning Profile: [REDACTED]	8 May 2018
iOS Team Provisioning Profile: [REDACTED]	8 May 2018
iOS Team Provisioning Profile: [REDACTED]	4 May 2018
iOS Team Provisioning Profile: [REDACTED]	4 May 2018
iOS Team Provisioning Profile: orf_farock *	17 Mar 2018
+ -	

Done

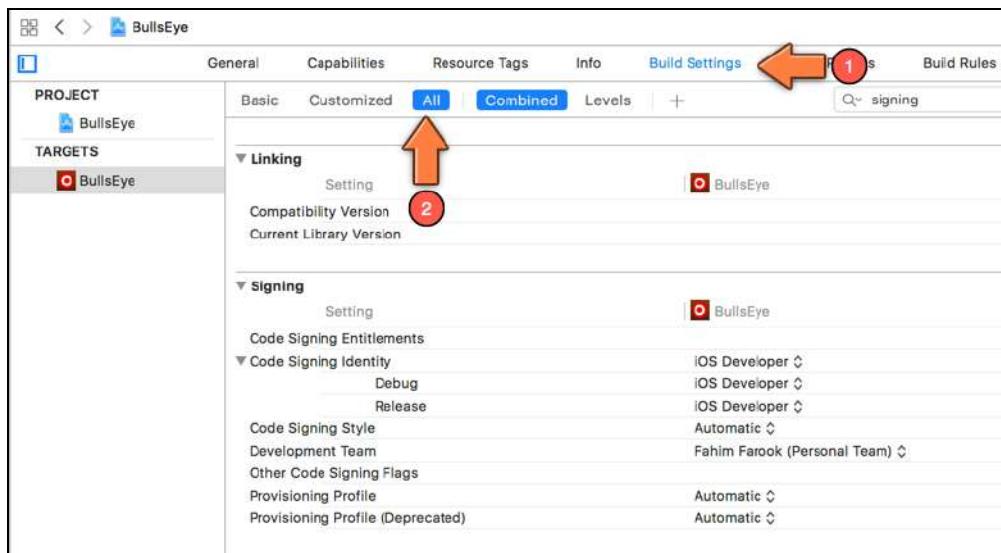
The provisioning profiles on your device

The “iOS Team Provisioning Profile” is the one that allows you to run the app on your device. By the way, they call it the “team” profile because often there is more than one developer working on an app and they can all share the same profile.

You can have more than one certificate and provisioning profile installed. This is useful if you're on multiple development teams or if you prefer to manage the provisioning profiles for different apps by hand.

To see how Xcode chooses which profile and certificate to sign your app with, go to the **Project Settings** screen and switch to the **Build Settings** tab. There are a lot of settings in this list, so filter them by typing **signing** in the search box. Also make sure **All** is selected, not Basic.

The screen will look something like this:



The Code Signing settings

Under **Code Signing Identity** it says **iOS Developer**. This is the certificate that Xcode uses to sign the app. If you click on that line, you can choose another certificate. Under **Provisioning Profile** you can change the active profile. Most of the time, you won't need to change these settings but at least you know where to find them now.

And that concludes everything you need to know about running your app on an actual device.

The end... or the beginning?

It's been a bit of a journey to get to this point — if you're new to programming, you've had to get a lot of new concepts into your head. I hope your brain didn't explode! At the very least, you should have gotten some insight into what it takes to develop an app.

I don't expect you to totally understand everything that you did, especially not the parts that involved writing Swift code and the finer points of building a user interface with SwiftUI. It is perfectly fine if you didn't, as long as you're enjoying yourself and you sort of get the basic concepts of objects, methods and variables. If you were able to follow along and do the exercises, you're in good shape!

I encourage you to play around with the code a bit more. The best way to learn programming is to do it, and that includes making mistakes and messing things up. I hereby grant you full permission to do so! Maybe you can add some cool new features to the game (and if you do, please let me know).

In the **Source Code** folder for this book, you can find the complete source code for the *Bullseye* app. If you're still unclear about anything you did between the start of the project and this finished product, it might be a good idea to look at this cleaned up source code.

If you're interested in how the graphics for *Bullseye* were made, take a peek at the Photoshop files in the Resources folder. The wood background texture was made by Atle Mo from subtlepatterns.com.

If you're feeling exhausted after all that coding, pour yourself a drink and put your feet up for a bit. You've earned it! On the other hand, if you just can't wait to get coding again, let's move on to our next app!

Section 2: Checklists

In this section, you'll build *Checklists*. As you may have gathered from the name, it's a TODO app that lets the user create, manage, and track items in one or more lists. Lists are a key part of many apps, and the lessons you'll learn while building Checklists will serve you well when you start coding your creations.

You'll make a multi-screen app, which will teach you the concepts of navigating from screen to screen, and sharing information between screens. You'll see how SwiftUI makes it easy to display lists of data. You'll also learn about data models (how data is represented in a program) and data persistence (saving data). And finally, you'll use local notifications to present the user with timely reminders and important messages that appear at the top of the screen. By the end of this section, you'll be able to write some basic (but useful) productivity apps.

This section contains the following chapters:

9. List Views: It's time to start your next iOS project. Are you ready for the challenge? In this chapter, we will commence our next app using SwiftUI, Checklists. Prepare for NavigationView, Arrays, Loops and removing items from the list.

10. A "Checkable" List: A Checklist app without being able to tick off the items? In this chapter, you will add the toggle for a Checklist item.

11. The App Structure: You have eagerly made great progress on creating a TODO list app by adding the checked status. In this chapter, you will start adding more features and start thinking about iOS design patterns.

12. Adding Items to the List: Your goal in this chapter is to start adding new items to your TODO list app. It also included learning about CRUD (Create, report, update and delete).

13. Editing Checklist Items: Congratulations! You can now add new items to your TODO list app, in this chapter, it's time to start editing your list and changing the text.

14. Saving & Loading: In this final chapter for your TODO list app, you will learn

about data persistence. Right now all the items are hardcoded so it's time to persist and go!



9

Chapter 9: List Views

Joey deVilla

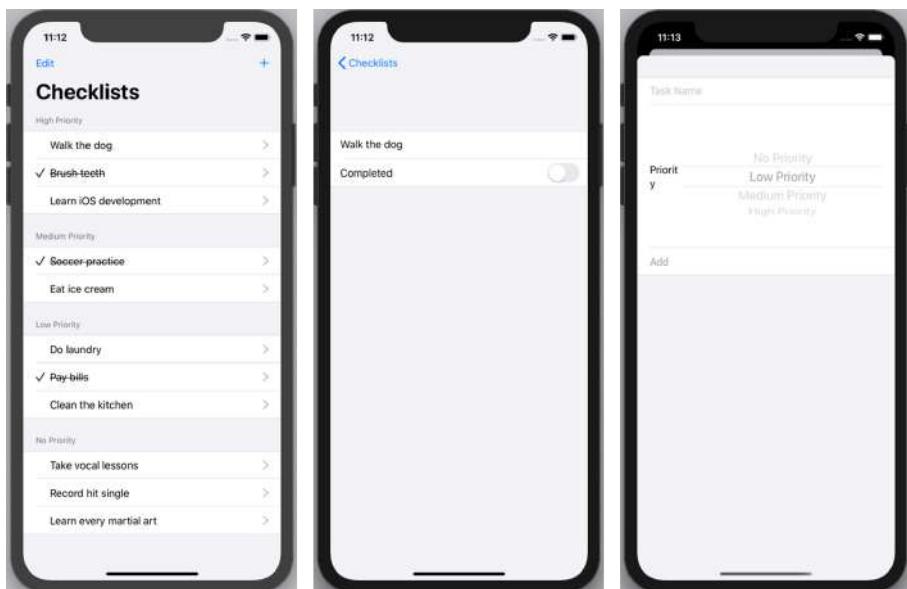
Ready to get started on your next app? Here you go!

To-do list apps are one of the most popular types of app on the App Store. Do a search on the web for “iOS to-do list apps” and you’ll see reviews of dozens of list apps, even though iOS comes bundled with the **Reminders** app. From pilots to busy parents to surgeons to Santa Claus, many people need to have a good checklist.

Building a to-do list app is a rite of passage for budding iOS developers. It’s the kind of app that’s so useful that it has its own category. Making one will teach you to master code features and functionality that you’ll end up using in so many other apps. It makes sense for you to create one as well.



Your own to-do list app, **Checklist**, will look like this when you're finished:



The finished Checklist app

This app lets you organize to-do items into a list with various priority levels. You can check off the items once you've completed them.

As far as to-do list apps go, *Checklist* is very basic, but don't let that fool you. Even a simple app such as this has a lot of complexity behind the scenes. In building it, you'll learn many different programming concepts. You're going to continue to use SwiftUI, the brand new way to build iOS apps, to build *Checklist*.

This chapter will cover:

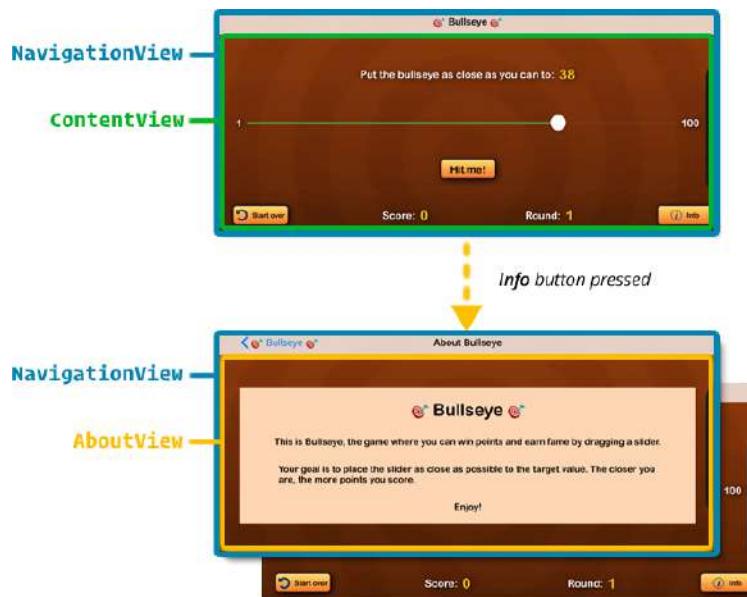
- **NavigationView and List views:** You'll quickly review NavigationView, then you'll learn to use List. You'll also see how both views are often used together in apps that you probably use every day. Finally, you'll build an app that displays a basic list.
- **Arrays:** Just as Lists organize arrange views in a row, arrays organize data in the same way. Arrays are so useful that they're the most common data structure that programmers use.
- **Loops:** One of the reasons that computers are so useful is that they're very good at doing the same thing over and over again, no matter how tedious it is. How do they do this? With loops. You'll learn how to make your first dynamic list using List, arrays and loops.

- **Deleting items from the list:** At this point, you've filled the onscreen list with items. You'll then learn how to give the user the power to delete any item with a swipe of their finger.
- **Moving list items:** Deleting items is pretty cool, but giving the user the ability to rearrange the items on the list is even more impressive. It's amazing how few lines of code this takes.
- **Key points:** A quick review of what you've learned in this chapter.

NavigationView and List views

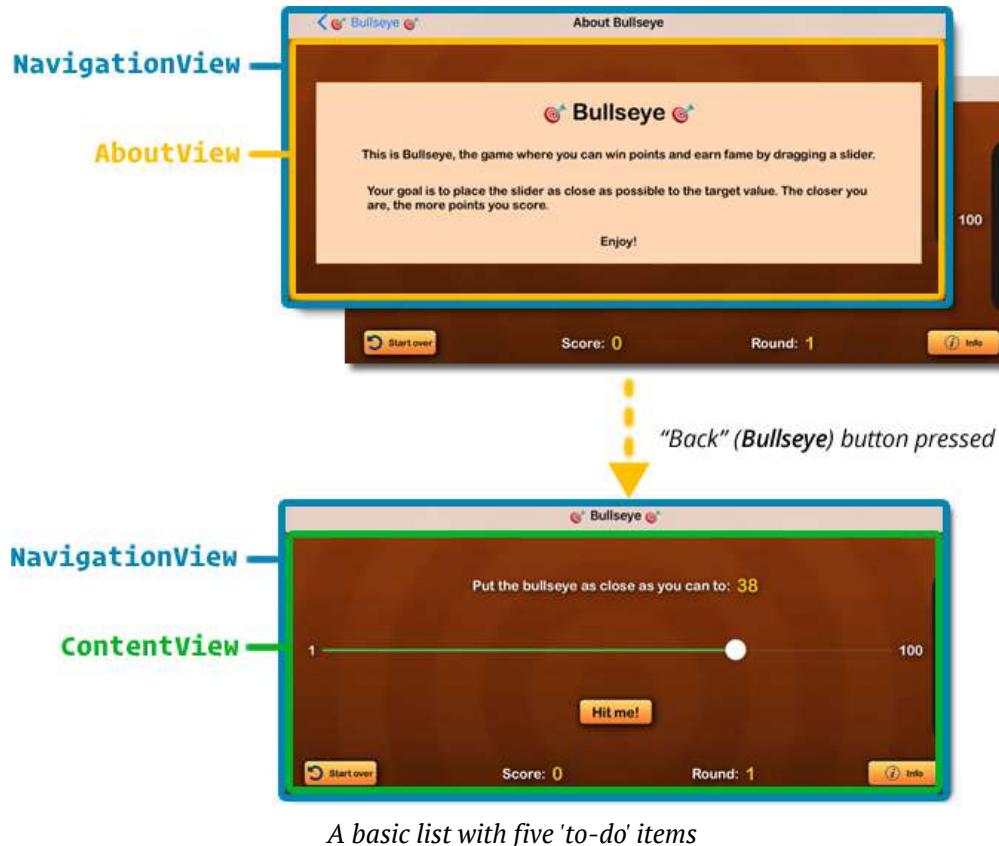
In the previous app, **Bullseye**, you learned to use `NavigationView`. `NavigationView` acts as a container for screens and makes it possible to navigate between them by creating a **navigation hierarchy** that's similar to how you navigate the web.

Embedding Bullseye's main screen, `ContentView`, inside a `NavigationView` lets users navigate to other screens by stacking those screens on top of `ContentView`. You did this by using a `NavLink` as the **Info** button. When a user presses the `NavLink`, that creates an instance of `AboutView`, which is then stacked on top of `ContentView`. Since `AboutView` is on top of the stack contained within `NavigationView`, the user sees that screen:



A basic list with five 'to-do' items

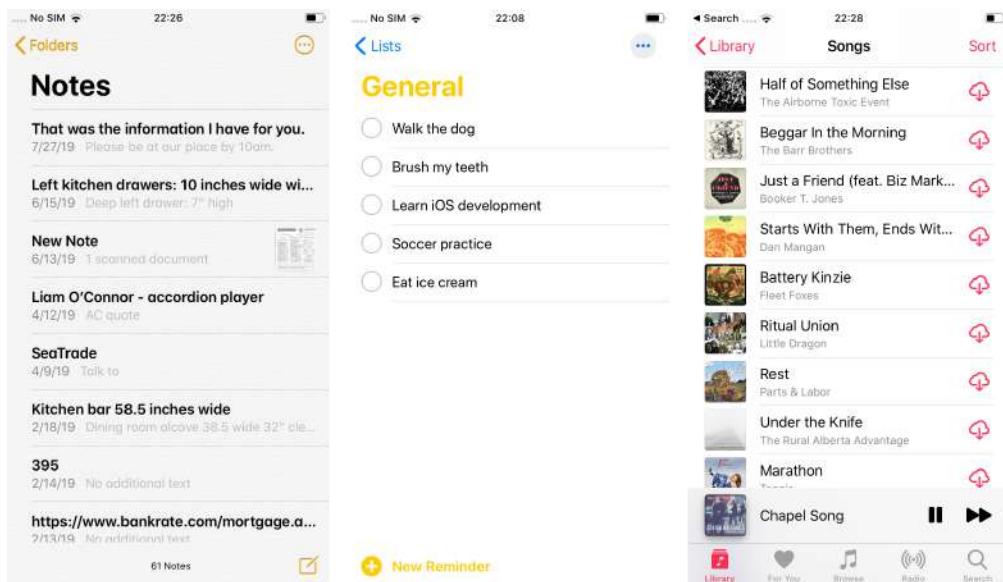
When the stack within a `NavigationView` is two or more screens deep, the navigation bar displays a “Back” button with the title of the screen just below. With Bullseye, the “Back” button that appears on `AboutView` looks like this: Bullseye. Pressing the “Back” button removes `AboutView` from the stack of screens within the `NavigationView`, which makes `ContentView` visible again:



List views

As its name implies, a `List` view displays lists, which are rows of data arranged in a single column. This user interface element is extremely versatile, the most important one to master in iOS development.

Take a look at the apps that come with your iPhone – **Notes**, **Reminders**, **Music**, **Mail** and **Settings**. You’ll notice that even though they look slightly different, all these apps work the same way, using a combination of Navigation and List views.



A basic list with five 'to-do' items

The **Music** app also has a tab bar at the bottom, which you'll learn about later on in this book.

If you want to learn how to program iOS apps in SwiftUI, you need to master these views as they make an appearance in almost every app. That's exactly what you'll focus on in this section of the book. You'll also learn how to pass data from one screen to another, a very important topic that often puzzles beginners.

A basic list

Start with a **static list**, which is one where the items and their order don't change. You might use this sort of list when you want to present the user with several choices or a table of contents.

This first app will simply display the following “to do” items in a list:

- Walk the dog.
- Brush my teeth.
- Learn iOS development.
- Soccer practice.
- Eat ice cream.

Go ahead and create a new Xcode Project using the **Single View App** template.

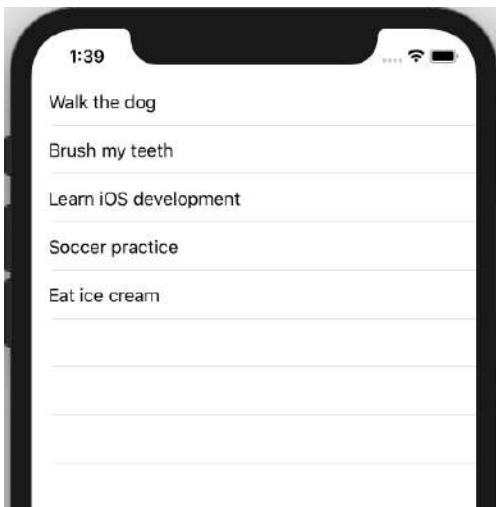
Since you just created a new project using the **Single View App** template, `ContentView`'s body property will look like this:

```
var body: some View {  
    Text("Hello World")  
}
```

► Replace `Text` with `List` so that the declaration for `body` looks like this:

```
var body: some View {  
    List {  
        Text("Walk the dog")  
        Text("Brush my teeth")  
        Text("Learn iOS development")  
        Text("Soccer practice")  
        Text("Eat ice cream")  
    }  
}
```

► Build and run the app in the Simulator. You'll see the “to do” items in a list form:



A basic list with five 'to-do' items

Lists are often used as the “master” part of a **master-detail interface**. In these interfaces, the user sees a **master list** of items, where each item in the list contains the item’s name and a couple of details about it. The user can select an item in the master list, which takes them to the **detail view** for the item, which displays all its information.

Many apps that come with your iOS device, like Calendar, Messages, Notes, Contacts, Mail and Settings, use a master-detail interface. For example, Mail has a master list showing each incoming email's sender and subject line along with a short excerpt. Selecting an email from the master list takes you to a screen that displays the full email.

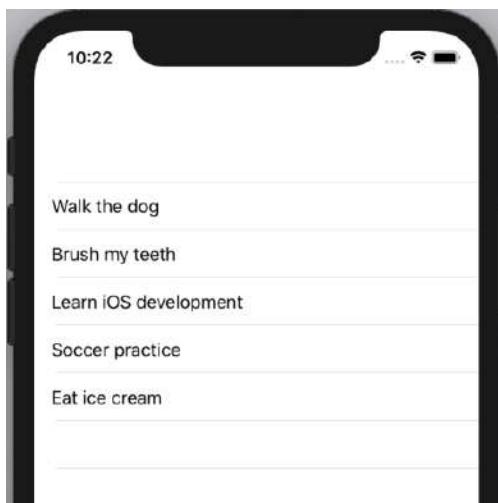
On the iPhone, you construct master-detail interfaces by putting a `List` inside a `Navigation`. `List` functions as the master list and `Navigation` makes it possible to navigate to the detail view and back.

Now, put the `List` you just created into a `Navigation` view.

- Change `ContentView`'s body so that `Navigation` now contains `List`. The result should be the following:

```
var body: some View {
    NavigationView {
        List {
            Text("Walk the dog")
            Text("Brush my teeth")
            Text("Learn iOS development")
            Text("Soccer practice")
            Text("Eat ice cream")
        }
    }
}
```

- Run the app. You'll see that `List` no longer takes up the whole screen:



The list, now contained within a navigation view

Navigation is a container that lets you build a hierarchy of screens that lead from one screen to another. It adds a navigation bar at the top, which can hold a title and some controls that allow the user to navigate between screens and perform edits on the screen that Navigation contains.

At the moment, the navigation bar is empty, which makes the app look like it's missing something. Fix that by adding a title to the navigation bar using `navigationBarTitle()`.

- Add a call to `navigationBarTitle()` to the `List` view so that the declaration for `body` becomes:

```
var body: some View {
    NavigationView {
        List {
            Text("Walk the dog")
            Text("Brush my teeth")
            Text("Learn iOS development")
            Text("Soccer practice")
            Text("Eat ice cream")
        }
        .navigationBarTitle("Checklist")
    }
}
```

- Run the app. It now has a title:



The app with a title in the navigation bar

Lists with sections

There are two styles of List: **plain** and **grouped**. Right now, the app's List uses the plain style.

Use the plain style for lists where all the items in the list are similar to one another, yet independent. One example is an email app, where each item in the List represents an email.

Use the grouped style when you can organize the items in the list by a particular attribute, like genre categories for a list of books. You could also use the grouped style table to show related information that doesn't necessarily have to go together, like a contact's address, contact information, and e-mail.

Lists default to the plain style. Using the grouped style requires the following:

1. Using List's `listStyle()` to specify that the list should use the grouped style.
2. Adding a **Section** inside the List for each group, which appears as a separate sublist within the list. Sections can contain their own headers.

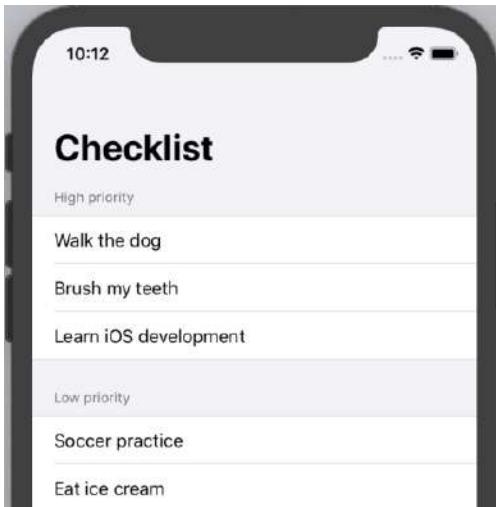
Now, change the list to the grouped style. You'll split the list into two groups using two **Section** views: One for high-priority and one for low-priority tasks. The first three items in the list will be high-priority, and the last two will be low-priority.

► Change the declaration for `body` to the following:

```
var body: some View {
    NavigationView {
        List {
            Section(header: Text("High priority")) {
                Text("Walk the dog")
                Text("Brush my teeth")
                Text("Learn iOS development")
            }
            Section(header: Text("Low priority")) {
                Text("Soccer practice")
                Text("Eat ice cream")
            }
        }
        .listStyle(GroupedListStyle())
        .navigationBarTitle("Checklist")
    }
}
```



- Run the app to see the grouped list in action:



The app with a grouped style list

The limits of views

Most people who use checklists have more than just five to-do items. At this point, make a more realistic list by adding more items so that both the high-priority and low-priority groups each have ten to-do items.

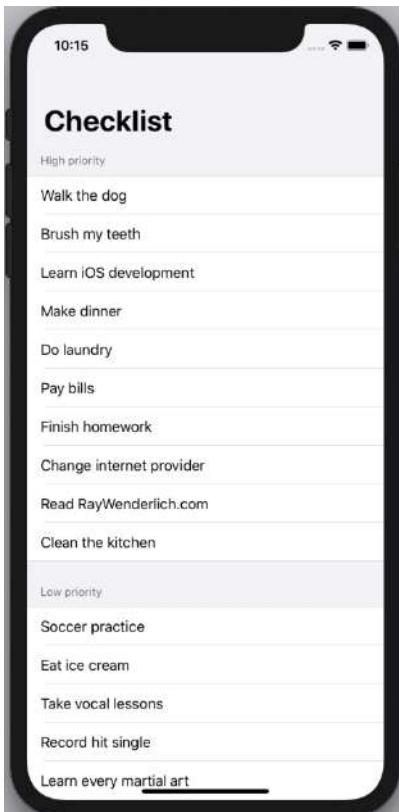
- Add more items to each Section so that the declaration for body looks like this:

```
var body: some View {
    NavigationView {
        List {
            Section(header: Text("High priority")) {
                Text("Walk the dog")
                Text("Brush my teeth")
                Text("Learn iOS development")
                Text("Make dinner")
                Text("Do laundry")
                Text("Pay bills")
                Text("Finish homework")
                Text("Change internet provider")
                Text("Read Raywenderlich.com")
                Text("Clean the kitchen")
            }
            Section(header: Text("Low priority")) {
                Text("Soccer practice")
                Text("Eat ice cream")
                Text("Take vocal lessons")
            }
        }
    }
}
```

```
        Text("Record hit single")
        Text("Learn every martial art")
        Text("Design costume")
        Text("Design crime-fighting vehicle")
        Text("Come up with superhero name")
        Text("Befriend space raccoon")
        Text("Save the world")
    }
}
.listStyle(GroupedListStyle())
.navigationBarTitle("Checklist")
}
}
```

Make a note of the number of Texts inside each Section: Ten. This number will become important shortly.

- Run the app. Even on the largest iPhones, you'll need to scroll to see the entire list:



The list with ten items in each section

Now, add one more item to the high-priority group in the list, making for eleven Texts inside the first Section.

- Add one more item to the first Section so that the declaration for body looks like this:

```

var body: some View {
    NavigationView {
        List {
            Section(header: Text("High priority")) {
                Text("Walk the dog")
                Text("Brush my teeth")
                Text("Learn iOS development")
                Text("Make dinner")
                Text("Do laundry")
                Text("Pay bills")
                Text("Finish homework")
                Text("Change internet provider")
                Text("Read RayWenderlich.com")
                Text("Clean the kitchen")
                Text("Wash the car")
            }
            Section(header: Text("Low priority")) {
                Text("Soccer practice")
                Text("Eat ice cream")
                Text("Take vocal lessons")
                Text("Record hit single")
                Text("Learn every martial art")
                Text("Design costume")
                Text("Design crime-fighting vehicle")
                Text("Come up with superhero name")
                Text("Befriend space raccoon")
                Text("Save the world")
            }
        }
        .listStyle(GroupedListStyle())
        .navigationBarTitle("Checklist")
    }
}

```

Shortly after you add that item, Xcode will respond with a cryptic complaint:
Ambiguous reference to member 'buildBlock()'.

```

19  var body: some View {
20      NavigationView {
21          List {
22              | Section(header: Text("High priority")) { | Ambiguous reference to member 'buildBlock()'
23                  Text("Walk the dog")
24                  Text("Brush my teeth")
25                  Text("Learn iOS development")
26                  Text("Make dinner")
27                  Text("Do laundry")

```

The 'Ambiguous reference' error message

This is one of those technically correct, but ultimately unhelpful error messages that Xcode will surprise you with from time to time. Rather than bore you with the internal details of how SwiftUI builds user interfaces based on a View's body, I'll keep it simple. You've just run into a big limit of working with Views: They're limited to holding a maximum of ten Views.

In a static list, the simplest way to get around this limitation is to use a `Group`. Its sole purpose is to provide a way for you to treat a group of two to ten views as a single view.

- In `body`'s first `List`, put the first six `Text` views into a `Group`, then do the same for the last five. You should end up with a `body` that looks like this:

```
var body: some View {
    NavigationView {
        List {
            Section(header: Text("High priority")) {
                Group {
                    Text("Walk the dog")
                    Text("Brush my teeth")
                    Text("Learn iOS development")
                    Text("Make dinner")
                    Text("Do laundry")
                    Text("Pay bills")
                }
                Group {
                    Text("Finish homework")
                    Text("Change internet provider")
                    Text("Read RayWenderlich.com")
                    Text("Clean the kitchen")
                    Text("Wash the car")
                }
            }
            Section(header: Text("Low priority")) {
                Text("Soccer practice")
                Text("Eat ice cream")
                Text("Take vocal lessons")
                Text("Record hit single")
                Text("Learn every martial art")
                Text("Design costume")
                Text("Design crime-fighting vehicle")
                Text("Come up with superhero name")
                Text("Befriend space raccoon")
                Text("Save the world")
            }
        }
        .listStyle(GroupedListStyle())
        .navigationBarTitle("Checklist")
    }
}
```

- Run the app. It should work now because the first List now contains only *two* views, which is well under the limit of ten.

The limits of static lists

Static lists have their uses, but they're missing a lot of features that a checklist app needs. The user needs to be able to add and delete items from the list, edit existing items and change their order in the list.

So far, you've seen only one way to change the contents of a list: By changing the contents of the List view during the programming process. You lock in these changes at **compile time** — that is, when you compile your code into an app that the device can run. What you need is a way to change the list at **run time**, which is while the app is running.

This is what you'll do next, but before you can, you'll need to become acquainted with arrays.

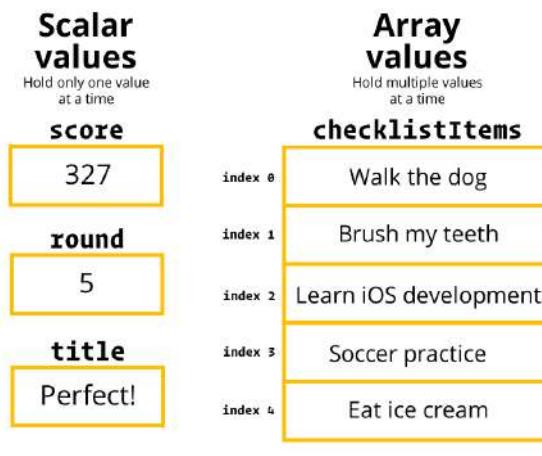
Arrays

Up to this point, you've been working with **scalar** variables and constants. These are variables and constants that hold one value at any given time. You used several scalar variables and constants in the Bullseye app. Each one acted as a container for one value, such as the score, the round, the value of the target and the message that the user received based on their score.

Not all data return a single value. A lot of data come as an ordered series of values, such as the mid-day temperature of a given location for the past month, the grades for every student in a given course... or the items in a checklist. Most programming languages, Swift included, have a data type called an **array**, which can hold many values at the same time.

If you think of scalar variables and constants as boxes that hold one value, think of an array as a *collection* of boxes. Each box holds one value, and the boxes are numbered in ascending order starting with 0. The numbering scheme lets you point to the contents of a specific box ("What's inside box 5?") or to put a value into a specific box ("Put the value **42** into box 3").

In an array, you call these boxes **elements**.



Scalar values and array values

In Swift, arrays can grow by having more elements added to them; they can also shrink by having elements removed from them. You'll use this ability, coupled with arrays' ability to hold many values, to store information about the checklist in this app.

The key to mastering programming is to learn by doing, especially when you're dealing with arrays and other concepts that you don't use outside of coding. So put this knowledge into practice by setting up the app to make use of an array to store list items.

Creating an array

Your next step is to create an array to hold the checklist items. Enter the following, immediately after the start of `ContentView` (the `struct ContentView: View` { line):

```
var checklistItems = ["Walk the dog", "Brush my teeth", "Learn
iOS development", "Soccer practice", "Eat ice cream"]
```

To walk through this line of code:

- The `var checklistItems` says “This is a variable named `checklistItems`. It’s not followed by a colon (:), which means that it’s up to Swift to infer what kind of variable `checklistItems` is.

- The = in Swift stands for “takes the following value”. You should read var checklistItems = as “This is a variable named checklistItems and it takes the following value.”
- The simplest way to define an array is to take a list of values separated by commas (,) and surround them with square brackets, which is what appears on the right side of the =. This value is an array of the five strings that made up our original set of checklist items.

Accessing array elements

The name checklistItems refers to the entire array. When you need to provide the entire array to an object or method, as you’ll need to do shortly, you’ll use that.

To access a specific element of an array, you use the array’s name followed by the number of the element you want to access in square brackets. For example, if you wanted to access element 2 of the checklistItems array, you’d use this syntax:

```
checklistItems[2]
```

The number inside the square bracket specifies its location in the array; it’s called the **index**. The combination of the square brackets and the number inside is called the **subscript**. In the case of checklistItems[2]:

- checklistItems[2] is an **element** of checklistItems.
- 2 is the **index** that specifies the third element. Remember, the first array index is **0**, not **1**.
- [2] is the **subscript** that specifies that you want a specific element of the array and not the whole thing.

Now that you know how to access individual array elements, it’s time to put that knowledge to use.

► Change the body property to the following:

```
var body: some View {
    NavigationView {
        List {
            Text(checklistItems[0])
            Text(checklistItems[1])
            Text(checklistItems[2])
            Text(checklistItems[3])
            Text(checklistItems[4])
        }
    }
}
```

```
        .navigationBarTitle("Checklist")
    }
}
```

Just so that there isn't any confusion at this point, the complete code for `ContentView` should look like this:

```
struct ContentView: View {
    var checklistItems = ["Walk the dog", "Brush my teeth", "Learn
    iOS development", "Soccer practice", "Eat ice cream"]

    var body: some View {
        NavigationView {
            List {
                Text(checklistItems[0])
                Text(checklistItems[1])
                Text(checklistItems[2])
                Text(checklistItems[3])
                Text(checklistItems[4])
            }
            .navigationBarTitle("Checklist")
        }
    }
}
```

► Before we get into a discussion about array syntax, run the app to see what the new code does. You should see this:



The list, using an array

Changing array elements and responding to taps on list items

Changing an array element is simple: You put the array element you want to change on the left side of an = and the value you want to change it to on the right side. For example, to change the contents of the first item in checklistItems to “Take the dog to the vet”, you’d use this code:

```
checklistItems[0] = "Take the dog to the vet"
```

Now, change the list so that tapping on the first item in the list, **Walk the dog**, changes the item to **Take the dog to the vet**. You’ll take advantage of a method built into every view object: onTapGesture(), which causes code to be executed whenever a user taps the view.

► Update body so that it looks like this:

```
var body: some View {
    NavigationView {
        List {
            Text(checklistItems[0])
                .onTapGesture {
                    self.checklistItems[0] = "Take the dog to the vet"
                }
            Text(checklistItems[1])
            Text(checklistItems[2])
            Text(checklistItems[3])
            Text(checklistItems[4])
        }
    }
}
```

Note that you use `self.checklistItems[0]` instead of just plain `checklistItems[0]`. That’s because the code in the braces after `.onTapGesture` is a closure — a self-contained bit of code that you can pass around as if it were a value like an integer or a string. That means it needs to use the `self` keyword to refer to `checklistItems`.

Even with the use of `self`, Xcode still reports an error: **Cannot assign through subscript: 'self' is immutable...**

```
11 struct ContentView: View {  
12  
13     var checklistItems = ["Walk the dog", "Brush my teeth", "Learn iOS development",  
14         "Soccer practice", "Eat ice cream"]  
15  
16     var body: some View {  
17         NavigationView {  
18             List {  
19                 Text(checklistItems[0])  
20                     .onTapGesture {  
21                         self.checklistItems[0] = "Take the dog to the vet"  
22                     }  
23                 Text(checklistItems[1])  
24                 Text(checklistItems[2])  
25                 Text(checklistItems[3])  
26                 Text(checklistItems[4])  
27             }  
28             .navigationBarTitle("Checklist")  
29         }  
30     }  
}
```

 Cannot assign through subscript: 'self' is immutable

The 'self is immutable' error message

Oh look, it's another technically correct but not-so-helpful message from Xcode! What does it mean?

To keep things simple here, just remember that *by default*, code inside a `struct` object is not allowed to change the values of that `struct`'s properties. *Normally*, only code outside the `struct` is allowed to do that.

Note the emphasized words in the previous paragraph: *by default* and *normally*. There *are* a couple of ways for code inside a `struct` to change the values of that `struct`'s own properties — and better yet, you've already used one of them!

That way is the `@State` attribute, which marks a property as a state variable. Remember, state variables determine what happens in the app, and the user's actions often affect them, and the view must respond to those changes. Marking a property with `@State` makes it exempt from the rule that code inside a `struct` can't change that `struct`'s own properties.

So go ahead and add the `@State` attribute to the declaration of `checklistItems`. While you're at it, reformat `checklistItems` so that it's easier to read and change.

- Update the declaration of checklistItems to the following:

```
@State var checklistItems = [  
    "Walk the dog",  
    "Brush my teeth",  
    "Learn iOS development",  
    "Soccer practice",  
    "Eat ice cream",  
]
```

Now that it's marked as a state variable, checklistItems is exempt from the “structs can't modify their own properties” rule and the onTapGesture() code can now change the value inside checklistItems[0]. As a result, the error message should disappear.

Note the new formatting, which makes the checklistItems array easier to read. It's also easier to get a sense of how many items are in the array. Swift ignores most “white space” — spaces, tabs, new lines and so on — which allows you to format the code for maximum legibility.

Note that the last item, **Eat ice cream**, has a , after it, even though it's not followed by another item. That's not an error; that's a deliberate addition that makes it easy to add another item after it, should it become necessary. Good code is code that's easy to update.

- Run the app and tap the “Walk the dog” item in the list. By default, SwiftUI is a little fussy and won't register the tap unless you tap right on the **Walk the dog** text. When tapped, the item should change to “Take the dog to the vet”:



The list, with the first item changed to 'Take the dog to the vet' after the user tapped on it

The limits of the current approach

Suppose you added an additional item — “Learn every martial art” — to `checklistItems`:

```
@State var checklistItems = [
    "Walk the dog",
    "Brush my teeth",
    "Learn iOS development",
    "Soccer practice",
    "Eat ice cream",
    "Learn every martial art",
]
```

If you were to run the app, it wouldn’t display the newly-added item. That’s because `List` in the body is currently set to display `Text` views for `checklistItems[0]` through `checklistItems[4]`:

```
List {
    Text(checklistItems[0])
        .onTapGesture {
            self.checklistItems[0] = "Take the dog to the vet"
        }
    Text(checklistItems[1])
    Text(checklistItems[2])
    Text(checklistItems[3])
    Text(checklistItems[4])
}
```

What you need is a way for the list to display the complete contents of `checklistItems` without having to make any changes to `body` as the array changes. To help you do that, it’s time to introduce a concept that goes hand-in-hand with arrays (and many other aspects of programming): loops.

Loops

Up to this point, you’ve experienced two different kinds of **flow control**, or ways of executing your code. Let’s look at them, and then you’ll learn a new kind of flow control: **Looping**.

Flow control

The first kind of flow control that you’ve seen is **sequence**, which is simply performing instructions in the order in which they appear.



Here's an example from Bullseye:

```
func startNewGame() {  
    score = 0  
    round = 1  
    resetSliderAndTarget()  
}
```

In `startNewGame()`, the code performs its instructions in order. First, the value of `score` is set to 0, then the value of `round` is set to 1 and finally, it executes `resetSliderAndTarget()`.

You've also seen the second kind of flow control: **Branching**, which some computer science people also like to call **selection**. This is the “decision-making” flow control, which offers two or more courses of action, depending on some kind of test. Here's an example from Bullseye:

```
func alertTitle() -> String {  
    let title: String  
    if sliderTargetDifference == 0 {  
        title = "Perfect!"  
    } else if sliderTargetDifference < 5 {  
        title = "You almost had it!"  
    } else if sliderTargetDifference <= 10 {  
        title = "Not bad."  
    } else {  
        title = "Are you even trying?"  
    }  
    return title  
}
```

In `alertTitle()`, the code performs different instructions based on the value of `sliderTargetDifference`. Each of these instructions is a branch in a structure of multiple choices called a **decision tree**.

It's time to introduce a third kind of flow control: **Looping**, which some computer science people also call **iteration**. This is the “repetition” flow control, where instructions are performed over and over again, either indefinitely or until some condition is met.

You're going to use looping to go through the elements in `checklistItems`, one at a time, in order, to display each one onscreen. You can also call this process **looping through** or **iterating through** each element.

for loops

You'll start by displaying all the elements in `checklistItems` in Xcode's debug console. Another way of saying this is "For every item in `checklistItems`, print its name," which is pretty close to the way you'd code it in Swift:

```
for item in checklistItems {  
    print(item)  
}
```

The code above goes through every item in `checklistItems`, starting with the item in index 0 and ending after the last item in the array. `item` is a temporary variable that exists only as long as we're still in the loop. The first time through the loop, `item` is set to the first item in the array, "Walk the dog".

The code inside the loop is then executed: It's `print(item)`, which prints "Walk the dog" in Xcode's debug console. This completes the first iteration of the loop.

The program returns to the beginning of the loop, and there are still more items in `checklistItems` to go through. `item` is set to the next item in the array, "Brush my teeth", after which the code in the loop executes, printing "Brush my teeth" in Xcode's debug console. The second iteration of the loop is now complete.

The program returns to the beginning of the loop, and the cycle repeats until it's gone through every item in `checklistItems`.

You can see this process in action.

► First, make sure that `checklistItems` has the original five items:

```
@State var checklistItems = [  
    "Walk the dog",  
    "Brush my teeth",  
    "Learn iOS development",  
    "Soccer practice",  
    "Eat ice cream",  
]
```

Take the loop code shown earlier and put it inside a method.

► Enter the following method after the end of body:

```
func printChecklistContents() {  
    for item in checklistItems {  
        print(item)  
    }  
}
```

Finally, you need to do two things with body:

1. Since you won't be displaying the contents of checklistItems onscreen yet, you'll simplify the list that appears onscreen.
2. You'll also use `onAppear()`, which is built into every view, to call `howChecklistContents()` when the list is first drawn onscreen.

► Change body to the following:

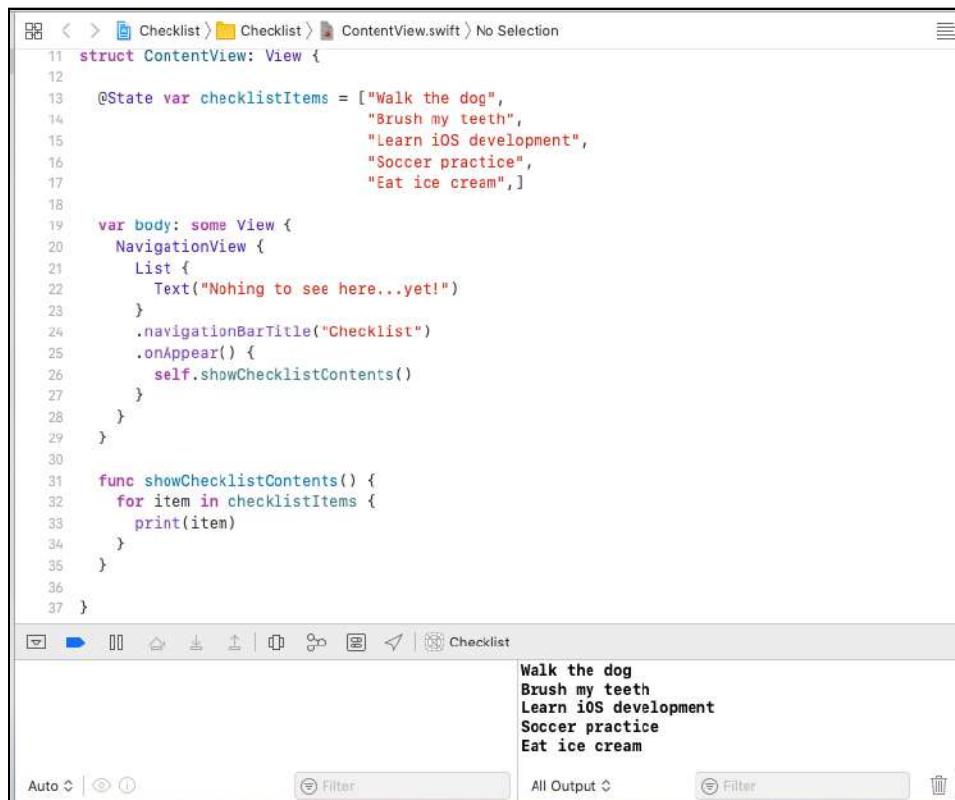
```
var body: some View {
    NavigationView {
        List {
            Text("Nothing to see here...yet!")
        }
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

With all the changes you made, `ContentView` should look like this:

```
struct ContentView: View {
    @State var checklistItems = [
        "Walk the dog",
        "Brush my teeth",
        "Learn iOS development",
        "Soccer practice",
        "Eat ice cream",
    ]
    var body: some View {
        NavigationView {
            List {
                Text("Nothing to see here...yet!")
            }
            .navigationBarTitle("Checklist")
            .onAppear() {
                self.printChecklistContents()
            }
        }
    }
    func printChecklistContents() {
        for item in checklistItems {
            print(item)
        }
    }
}
```



- Run the app but don't bother looking at the simulator. All the action is happening in the Xcode window, in the debug console:



The screenshot shows the Xcode interface. The top part displays the code for `ContentView.swift`. The bottom part shows the debug console with the output of the `print` statements.

```
11 struct ContentView: View {
12
13     @State var checklistItems = ["Walk the dog",
14                                 "Brush my teeth",
15                                 "Learn iOS development",
16                                 "Soccer practice",
17                                 "Eat ice cream"]
18
19     var body: some View {
20         NavigationView {
21             List {
22                 Text("Nothing to see here...yet!")
23             }
24             .navigationBarTitle("Checklist")
25             .onAppear() {
26                 self.showChecklistContents()
27             }
28         }
29     }
30
31     func showChecklistContents() {
32         for item in checklistItems {
33             print(item)
34         }
35     }
36
37 }
```

Walk the dog
Brush my teeth
Learn iOS development
Soccer practice
Eat ice cream

The Xcode window, with the contents of `checklistItems` displayed in the debug console

When `List` first appears on the screen, its `onAppear()` is called, which then calls `printChecklistContents()`. `printChecklistContents()` goes through `checklistItems` in order, printing each item's value in the debug console.

As you've seen before, printing to the debug console doesn't affect the user interface; it's only for the benefit of the programmer. How do you show the contents of the checklist to the user?

The `ForEach` view

SwiftUI has a view called `ForEach` that takes a collection of data, such as an array, and generates views based on that data. This is one of those cases where “Show, don't tell” is the better approach, so take a look at `ForEach` in action first, then you'll look at it in more detail afterward.



- Change body to the following:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems, id: \.self) { item in
                Text(item)
            }
        }
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

- Run the app. You'll see this:



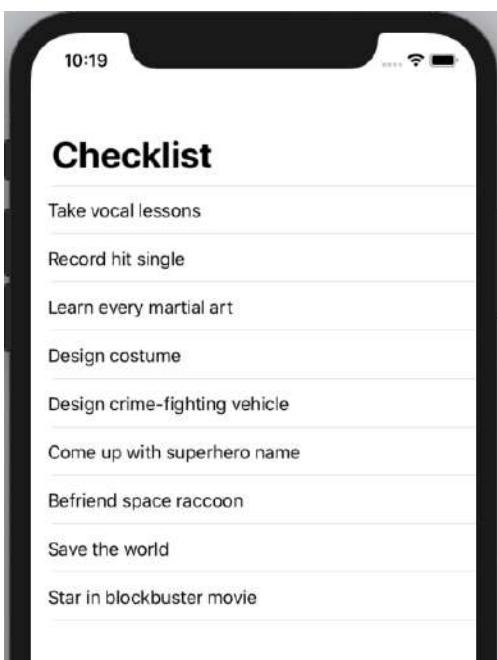
The original five-element array displayed using 'ForEach'

For the user, there's no difference between an app that uses `ForEach` and one that simply uses a `Text` view for every item in `checklistItems`. The benefit of `ForEach` becomes clear when you make changes to `checklistItems`.

- Change the declaration of `checklistItems` to the items on the following page, which has no items in common with the original list.

```
@State var checklistItems = [  
    "Take vocal lessons",  
    "Record hit single",  
    "Learn every martial art",  
    "Design costume",  
    "Design crime-fighting vehicle",  
    "Come up with superhero name",  
    "Befriend space raccoon",  
    "Save the world",  
    "Star in blockbuster movie",  
]
```

- Run the app to see this all-new list:



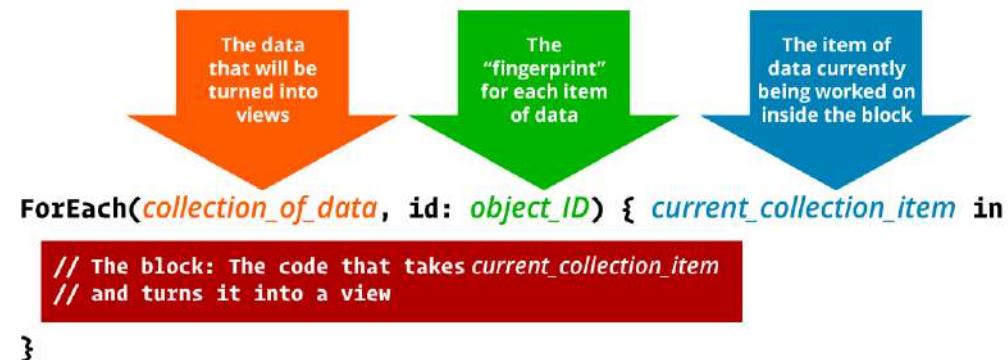
A completely different list displayed using 'ForEach'

With `ForEach`, you don't have to make any changes to the user interface to change the items in the checklist. You only have to change the underlying data.

Take a closer look at `ForEach`, which you added to `body`:

```
ForEach(checklistItems, id: \.self) { item in  
    Text(item)  
}
```

There are four key parts here:



A completely different list displayed using 'ForEach'

First, there's the **collection of data** that you're using `ForEach` to display in a view. In this case, that collection of data is `checklistItems`.

Next is a "**digital fingerprint**" that uniquely identifies each element in the collection of data provided to `ForEach`. That's what `id:` is for. SwiftUI uses it to update the views that `ForEach` creates when a user moves or deletes elements in the collection, and when they add new ones. For this parameter, you use `\.self`, which means: "Use the item's value as its identifier."

Then there's the value representing the **current collection item**, which lets you refer to that item in the code in the `ForEach` block. If you're using the most recent version of `checklistItems`, this value will be "Eat ice cream" during the first pass through `ForEach`, then "Take vocal lessons", then "Record hit single" and so on, until you hit the final item, "Star in blockbuster movie".

And finally, there's the block, which contains the code that specifies the views that will display the collection of data. In this case, it's a `Text` view that contains the current collection item.

The similarity between `for` and `ForEach` is intentional, especially the way the values represent the collection of data that you pass to them and the value that represents the item in the collection that you're currently working on.

The diagram below shows how the two relate to each other:

```
for current_item in array {  
    // Code to execute  
    // for each item in the array  
    // goes here  
}  
  
ForEach(array, id: object_ID) { current_item in  
    // Code to create a view  
    // for each item in the array  
    // goes here  
}
```

The similarities between 'for' and 'ForEach'

Deleting items from the list

Now that the list is based on an array and is no longer a “hardwired” part of the user interface, changing the array’s contents can change the list’s contents.

Adding items to the end of an array (and the checklist)

Since arrays are ordered lists of items, and lists often grow by adding items to the end, arrays have an `append()` method, which lets you add new items to the end of the array.

Suppose you have an array named `myArray` that contains the following items:

- This is an item.
- This is another item.
- Third item!

If you wanted to add another item, “One more item!!!” to the array, you’d use the `append()` method this way:

```
append("One more item!!!")
```

Once that code executes, `myArray` would contain:

- This is an item.
- This is another item.
- Third item!
- One more item!!!

You’d be able to access this new item with `myArray[3]`. Remember, array indexes begin at 0, so the fourth item is at index 3.

Now, try out `append()` in the app. Now that `List` is connected to `checklistItems`, which is a state variable, adding an item to the end of the array will add an item to the end of the `List`.

You’ll change the app so that tapping on a checklist item will add a copy of that item to the end of the list. You’ll use `onTapGesture()` to respond to the user’s taps.

► Start by changing `checklistItems` back to its original contents:

```
@State var checklistItems = [  
    "Walk the dog",  
    "Brush my teeth",  
    "Learn iOS development",  
    "Soccer practice",  
    "Eat ice cream",  
]
```

► Change body to the following:

```
var body: some View {  
    NavigationView {  
        List {  
            ForEach(checklistItems, id: \.self) { item in  
                Text(item)  
                    .onTapGesture {  
                        self.checklistItems.append(item)  
                        self.printChecklistContents()  
                    }  
            }  
        }  
        .navigationBarTitle("Checklist")  
    }  
}
```



```
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

Take a look at the `ForEach` view, which contains the lines of code you just added:

```
ForEach(checklistItems, id: \.self) { item in
    Text(item)
        .onTapGesture {
            self.checklistItems.append(item)
            self.printChecklistContents()
        }
}
```

You added a call to the `Text` view's `onTapGesture()`, which contains code that does two things:

- It uses `append()` to add `item` to the end of `checklistItems`. This means that tapping on the “Walk the dog” list item adds a new “Walk the dog” item to the end of `checklistItems`, which in turn causes a new “Walk the dog” item to appear at the end of the list onscreen.
- It calls `printChecklistContents()` so that you can look at Xcode's debug console to confirm that a new item was added to `checklistItems`.

► Run the app and tap the **Walk the dog** item. You'll see a new **Walk the dog** item at the end of the list...



Adding a new item to the end of the list

If you look at Xcode's debug console, you'll see the output of `printChecklistContents()`, which shows that `checklistItems` contains the following items in the given order:

- Walk the dog.
- Brush my teeth.
- Learn iOS development.
- Soccer practice.
- Eat ice cream.
- Walk the dog.

Later on, you'll change the app so that the user will be able to add new items of their choice instead of simply adding duplicates of existing items. The underlying principle will still be the same, however: You'll use `append()` to add the new, user-provided item to the end of `checklistItems`.

Removing items from an array (and the checklist)

There are several methods that remove an element from an array. You'll try two of them out in your app.

Removing items using `remove(at:)`

The simplest one is `remove(at:)`, which removes the element at the given index. For example, if you wanted to remove the first element of `checklistItems`, you'd use the code `checklistItems.remove(at: 0)`.

Now, use that code by changing the app so that tapping a list item removes the first item from the list. Do this by replacing this line in `onTapGesture()`:

```
self.checklistItems.append(item)
```

with this:

```
self.checklistItems.remove(at: 0)
```

- Incorporate this change by changing body to:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems, id: \.self) { item in
                Text(item)
                    .onTapGesture {
                        self.checklistItems.remove(at: 0)
                        self.printChecklistContents()
                    }
            }
        }
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

- Run the app and start tapping on list items. The list will shrink from the top, with the first item in the list disappearing with each tap until the list is empty:



An empty checklist

Using remove(atOffsets:) to remove list items

Another way to remove list items is `remove(atOffsets:)`. This is a bulk version of `remove(at:)`, which removes a specific range of elements from an array. You specify the starting index and ending index of the items you want to remove using

`IndexSet`. To see it in action, change the app so that tapping a list item removes the elements from index 0 through index 4. You do this by replacing this line in the `onTapGesture()` method:

```
self.checklistItems.remove(at: 0)
```

with this:

```
let indexesToRemove = IndexSet(integersIn: 0...4)
self.checklistItems.remove(atOffsets: indexesToRemove)
```

- Incorporate this change by changing body to:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems, id: \.self) { item in
                Text(item)
                    .onTapGesture {
                        let indexesToRemove = IndexSet(integersIn: 0...4)
                        self.checklistItems.remove(atOffsets:
                            indexesToRemove)
                        self.printChecklistContents()
                    }
            }
            .navigationBarTitle("Checklist")
            .onAppear() {
                self.printChecklistContents()
            }
        }
    }
}
```

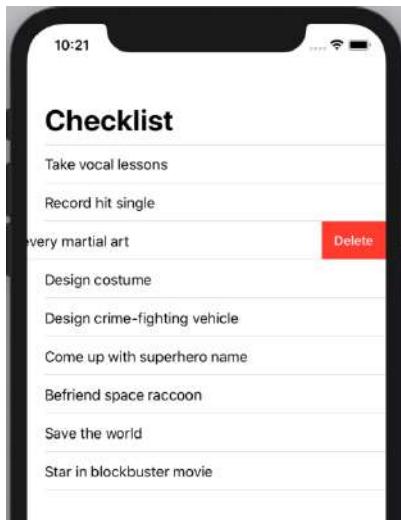
- Run the app and tap on any list item. That will call the `onTapGesture()` method, which will remove elements 0 through to 4 of `checklistItems`. This will empty the array, which in turn will empty the list.

Now that you know how to remove items from an array, it's time to learn how to respond to the “swipe to delete” gesture.

Responding to the “swipe to delete” gesture

Even though many apps use the list control that comes standard with iOS, many people still don't know the standard “swipe to delete” gesture.

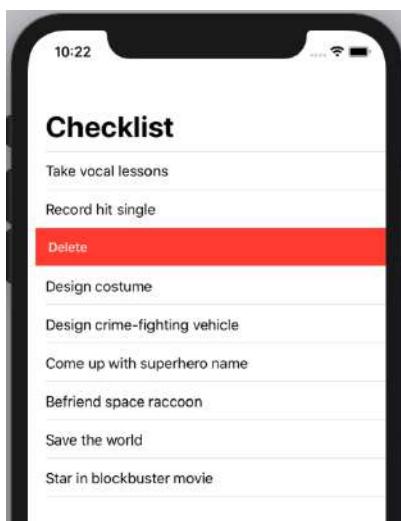
In many apps, you can put your finger on a list item and drag it slightly to the left to reveal a **Delete** button:



The 'Delete' button that appears when you 'slide to delete' a list item

To delete the list item, you can either tap the **Delete** button or continue swiping to the left. As your finger moves leftward, the list item gets moved offscreen and the **Delete** button grows wider until it's half the width of the list.

At that point, it expands to fill the entire width and deletes the list item:



The 'Delete' button at full width

The user can cancel the delete action either by swiping to the right or by tapping on the list item.

Your next step is to add this “swipe to delete” capability to the app. Do this by using `ForEach`’s `onDelete(perform:)`, which is called whenever the user completes the “swipe to delete” gesture.

`onDelete(perform:)` has one parameter, `perform:`, where you specify the name of a method that deletes the data behind the list item that the user just swiped.

`onDelete(perform:)` will pass this method an `IndexSet` that starts and ends with the list item’s index, so that it knows which data to delete.

It’s time to write this method, which you’ll call `deleteListItem`.

► Add this method after `printChecklistContents()`:

```
func deleteListItem(whichElement: IndexSet) {
    checklistItems.remove(atOffsets: whichElement)
    printChecklistContents()
}
```

The method removes the item whose index is contained in `index`, then prints the contents of `checklistItems` in Xcode’s debug console.

Now, add `onDelete(perform:)` to `ForEach`, which will enable “swipe to delete” for the user. You’ll also remove `onTapGesture()` from the `Text` in `ForEach`, just to keep things simple.

► Change body to the following:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems, id: \.self) { item in
                Text(item)
            }
            .onDelete(perform: deleteListItem)
        }
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

- Run the app, then swipe a list item to delete it. Check `printChecklistContents()`'s output in Xcode's debug console to confirm that the item you swiped no longer exists in `checklistItems`.

Now the user can delete items from the checklist!

Moving list items

Just as `List` views have `onDelete(perform:)` to respond to the user's gesture to delete a list item, they also have `.onMove(perform:)` to respond to the user's gesture to move a list item.

Unlike deleting items from a `List`, you can only move list items when the list is in “Edit” mode. To make “Edit” mode available to the user, you need to add the button that enables it to the navigation bar using `navigationBarItems()`.

- Change body to the following:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems, id: \.self) { item in
                Text(item)
            }
            .onDelete(perform: deleteListItem)
        }
        .navigationBarItems(trailing: EditButton())
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

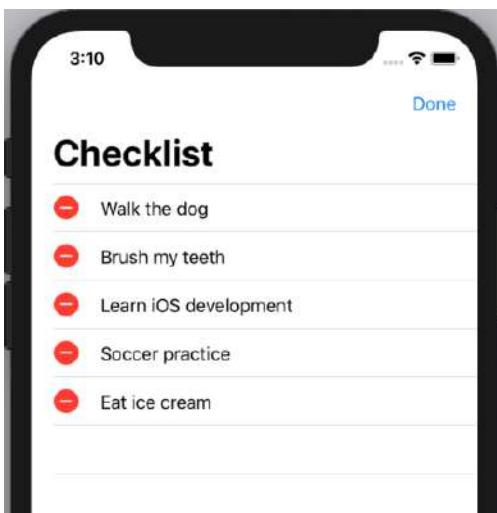
Here, you've added `navigationBarItems()` to `List`. Its parameter adds an `Edit` button to the “trailing” side of the navigation bar. In languages like English, which read from left to right, the trailing side is the right side.

- Run the app. There's now an **Edit** button in the navigation bar:



The app with an 'Edit' button in the navigation bar

- Tap the **Edit** button to switch to edit mode. The screen will look like this:



The app in edit mode, showing 'Delete' buttons

In edit mode, each list item has a **Delete** button, which looks like a minus sign in a red circle, on its left side. You can tap the button to delete the item. There's still no visual indicator for moving list items, because you haven't made use of `.onMove(perform:)` yet.

Like `onDelete(perform:)`, `.onMove(perform:)` has a `perform:` parameter, where you specify the name of a method that moves the data behind the list item that the user just moved. `.onMove(perform:)` will pass this method an `IndexSet` that starts and ends with the list item's index, so that it knows which data to move and an `Int` that indicates where to move it to. Now, write this method, which you'll call `moveListItem`.

- Add this method after `deleteListItem()`:

```
func moveListItem(whichElement: IndexSet, destination: Int) {  
    checklistItems.move(fromOffsets: whichElement, toOffset:  
    destination)  
    printChecklistContents()  
}
```

The first line of `moveListItem(whichElement:destination:)` uses an array method you haven't seen yet: `move(fromOffsets:, toOffset:)`, which moves one or more array elements within the array. You put the indexes of the starting and ending elements that you want to move in `fromOffsets:`, and the index of the place to move them to in `toOffset:..`

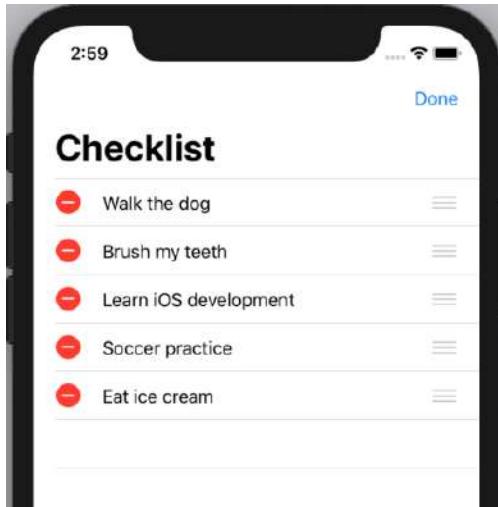
The second line displays the newly-rearranged contents of `checklistItems` so that you can confirm that the array element moved correctly.

Now, make use of `moveListItem` in `.onMove`.

- Change body to the following:

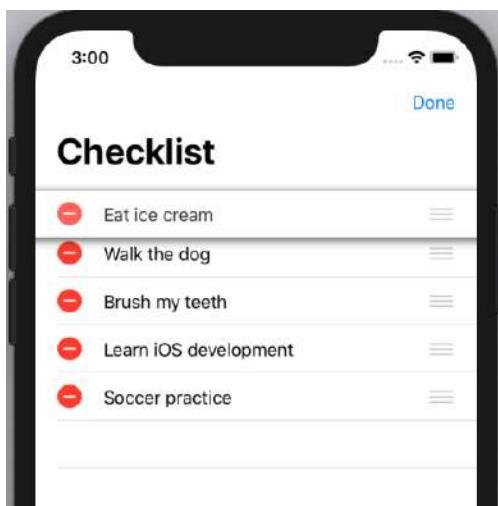
```
var body: some View {  
    NavigationView {  
        List {  
            ForEach(checklistItems, id: \.self) { item in  
                Text(item)  
            }  
            .onDelete(perform: deleteListItem)  
            .onMove(perform: moveListItem)  
        }  
        .navigationBarItems(trailing: EditButton())  
        .navigationBarTitle("Checklist")  
        .onAppear() {  
            self.printChecklistContents()  
        }  
    }  
}
```

- Run the app and tap the **Edit** button. This time, when you enter edit mode, you see the **Delete** buttons on the left side of each list item and **move handles** on the right side:



The app in edit mode, showing 'Delete' buttons and move handles

- Press down on the move handle of any list item and drag it to a new location:



Dragging 'Eat ice cream' to the top of the list

When you let go of the item, it “snaps” to the closest “slot”, changing the order of the list. You can confirm that `checklistItems` reflects the onscreen changes by checking the output of `printChecklistContents()` in Xcode’s debug console.

Key points

In this chapter, you did the following:

- You started with a static list of items that were “hard-wired” into the user interface, using both plain and grouped styles.
- You learned about arrays, the most-used data structure, and loops, one of the key means of flow control in programs.
- You applied your newly-gained knowledge about arrays and loops, as well as `ForEach`, to build a dynamic list. The contents of that list aren’t hard-wired into the user interface; underlying data determine the contents.
- With your dynamic list, you used the power of SwiftUI to give the user the ability to delete items from the list and rearrange the list’s items.

Phew! That was a lot of new stuff to take in, so I hope you’re still with me. If not, then take a break and start at the beginning again. You’re learning a whole bunch of new concepts all at once, and that can be overwhelming.

But don’t worry, it’s OK if everything doesn’t make perfect sense yet. As long as you get the gist of what’s going on, you’re good to go.

If you want to check your work up to this point, you can find the project files for the app under **09 - List Views** in the Source Code folder.

10

Chapter 10: A “Checkable” List

Joey deVilla

Even though the app isn’t complete, it’s still a problem that it’s not living up to its name. It displays a list of items, but it doesn’t show if they’re checked or not. It doesn’t even track if an item is checked or not. And it most certainly doesn’t let the user check or uncheck items!

In this chapter, the goal is to fix these problems by:

- **Creating checklist item objects:** Swift is an object-oriented programming language, which means that sooner or later, you’re going to have to “roll your own” objects. And by “sooner or later,” I mean *now*. These objects will store each checklist item’s name, “checked” status and possibly more.
- **Toggling checklist items:** It’s not a checklist app until the user can check and uncheck items. It’s time to make this app live up to its name!
- **Key points:** A quick review of what you’ve learned in this chapter.



Creating checklist item objects

Arrays: A review

In its current state, the app stores the checklist items in an array named `checklistItems`. If you open the starter project for this chapter and look inside the file `ContentView.swift`, you’ll see this at the start of `ContentView`:

```
@State var checklistItems = [  
    "Walk the dog",  
    "Brush my teeth",  
    "Learn iOS development",  
    "Soccer practice",  
    "Eat ice cream",  
]
```

You may notice that `checklistItems`’s last element, “Eat ice cream,” has a comma , after it. Swift will ignore it, but that trailing comma makes it easier for you to add additional elements to the array in the future. It’s a good idea to use tricks like this that make your programs easy to update.

Arrays are ordered lists of objects, and `checklistItems` is a specific array instance that contains the five items that the checklist contains when the app is launched:



The checklistItems array with its current contents

`checklistItems` is missing information. It stores the name of each item, but it doesn’t store each item’s “checked” status — that is, is the item checked or not? What we need is a `checklistItems` array where each element stores a checklist item’s name and “checked” status:

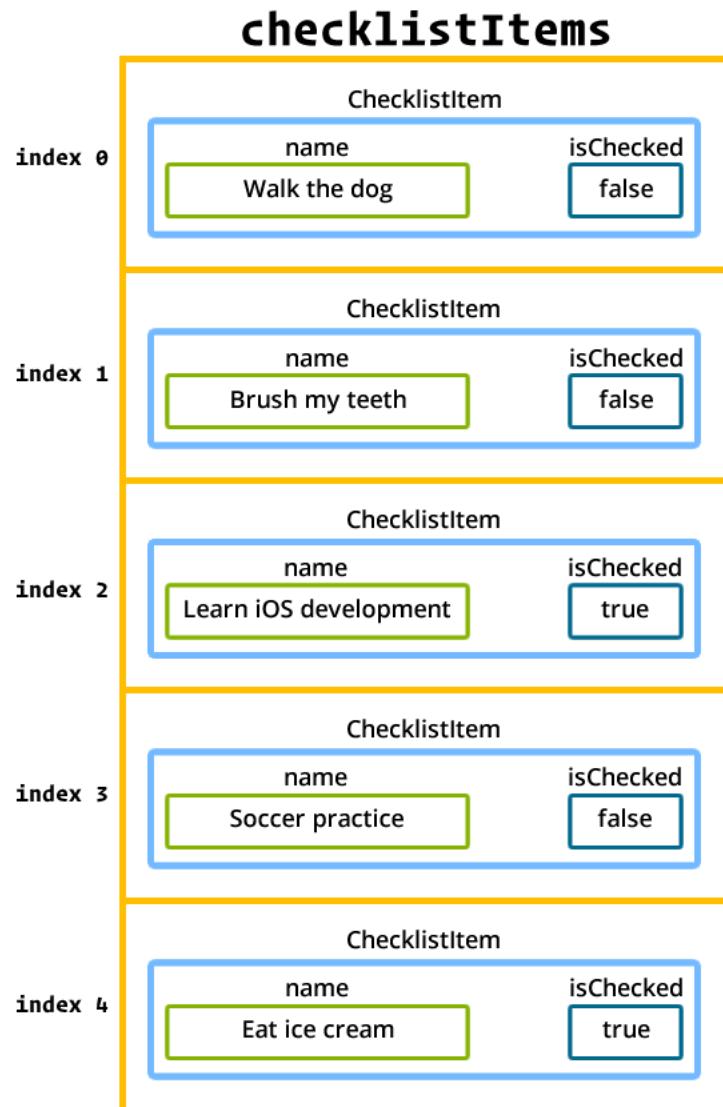
checklistItems		
	name	isChecked
index 0	Walk the dog	false
index 1	Brush my teeth	false
index 2	Learn iOS development	true
index 3	Soccer practice	false
index 4	Eat ice cream	true

The `checklistItems` array with each element holding two values

While an array can hold many items, its individual elements are like regular variables: They can hold only one item at a time.

What we need is a single package for each checklist item that we can use to hold its name, its “checked” status and any other data.

That package, being a single item, could be put into an array element or ordinary variable. Using these packages, the checklistItems array would look like this:



The checklistItems array with each element holding a single package containing all the value for a checklist item

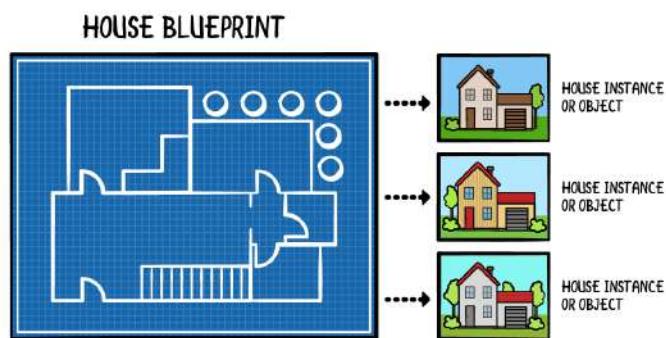
Here's the good news: These things exist, and you've already been using them. They're objects or, more accurately, instances of structs.

structs vs. objects or instances

Until now, I’ve been referring to `structs` as *objects* to keep things simple. This isn’t technically correct, but it was good enough to get you started. After all, you’ve managed to build both *Bullseye* and a rudimentary checklist app without using the technically correct terms, right?

An analogy that gets used all the time in object-oriented programming is the “blueprint and houses” analogy. A blueprint is a set of drawings that describes a house.

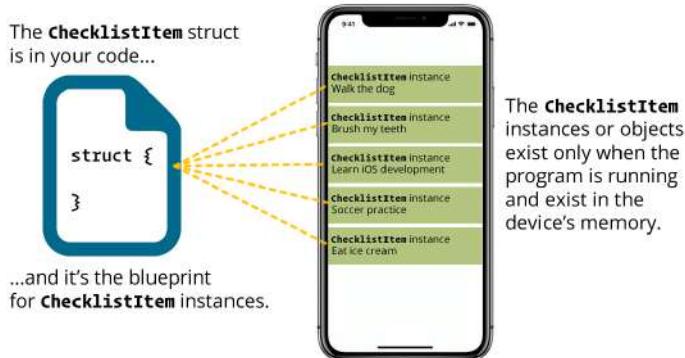
The blueprint is used to build one or more houses. The houses based on the same blueprint are identical in structure. They’re made with the same number of rooms with the same dimensions, but each house is a unique instance and will hold different people, furniture and other possessions.



You should think of a `struct` as the *blueprint* for objects. Just as you don’t live in a blueprint, your app doesn’t create `structs` to be components in your program. Instead, your program creates *objects* or *instances* based on the information in the `struct`.

For this app, you’ll code a `struct` named `ChecklistItem` that will be the blueprint for checklist item objects. When the app runs, instances of `ChecklistItem` will be created for each item in the list: “Walk the dog,” “Brush my teeth,” “Learn iOS development” and so on.

Each of these objects will hold two pieces of data: The checklist item’s name and its “checked” status.



A struct and its object instances

Creating a struct for checklist items

Let’s define the `ChecklistItem` struct. It will specify that its instances will have two properties:

- The name of the checklist item, which we’ll call `name`. Since this will contain text data, this will be a `String` property. The user should be able to change the name of checklist items, so this should be a variable, which we specify with the `var` keyword.
- The “checked” status of the checklist item. This is a true/false value, so it will be a `Bool` (Boolean) property. Since it’s a Boolean property, we’ll follow the convention of giving this property a name that begins with the word “is”: `isChecked`. The user should be able to check and uncheck checklist items, so this should also be a variable.

When someone adds an item to our checklist, we’ll assume the item is incomplete. After all, that’s why checklists exist in the first place. We should set up `ChecklistItem` so that unless otherwise indicated, its initial “checked” status should be “unchecked.”

With this criteria, there’s enough information to define the `ChecklistItem` struct.

Note that `ChecklistItem` starts with an uppercase “C.” In programming, things that act like blueprints typically have names that begin with uppercase

letters. Meanwhile, objects based on those blueprints usually have names that start with lowercase letters. It’s a convention that programmers use to make their code easier to understand.

- Add the following to `ContentView`, just after the `import SwiftUI` line and before the `struct ContentView: View {` line:

```
struct ChecklistItem {  
    var name: String  
    var isChecked: Bool = false  
}
```

The declaration for the `name` property simply says that it’s a `String` property. The declaration for `isChecked` ends with `= false`, which means that `isChecked` has a default value of `false`.

Now that you’ve defined the `ChecklistItem` blueprint, you can create objects based on it. There are a couple of ways you can refer to this process: Creating `ChecklistItem` objects, creating `ChecklistItem` instances or instantiating `ChecklistItem` objects or instances. Any of these are correct. They all describe making a new thing based on a design for that thing.

To create an object or instance of a `struct`, use the `struct`’s name followed by parentheses containing the `struct`’s properties and the values for those properties. For example, to create a `ChecklistItem` object for “Learn iOS development” that is checked, you’d write this line of code:

```
ChecklistItem(name: "Learn iOS development", isChecked: true)
```

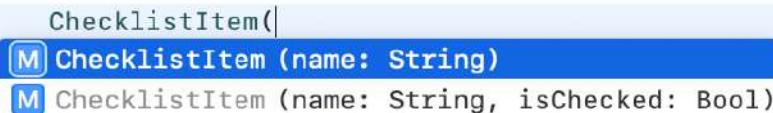
For a `ChecklistItem` object for “Walk the dog” that is unchecked, you can initialize it a couple of ways. First, there’s the complete way:

```
ChecklistItem(name: "Walk the dog", isChecked: false)
```

Since `isChecked` has a default value of `false`, you can simply just provide a value for the `name` parameter and skip providing a value for `isChecked`, which will cause it to default to `false`:

```
ChecklistItem(name: "Walk the dog")
```

When you’re creating a new instance of a struct, Xcode will try to help you by showing you the options. Here’s what it will show you when you’re making a ChecklistItem object:



Xcode will try to help you when you’re instantiating an object

Let’s update the checklistItems array by replacing the Strings that currently fill it with ChecklistItem instances. We want the same item names, and they should have these “checked” statuses:

- Walk the dog – *unchecked*
- Brush my teeth – *unchecked*
- Learn iOS development – *checked*
- Soccer practice – *unchecked*
- Eat ice cream – *checked*

► Edit checklistItems so that it looks like this:

```
@State var checklistItems = [  
    ChecklistItem(name: "Walk the dog"),  
    ChecklistItem(name: "Brush my teeth"),  
    ChecklistItem(name: "Learn iOS development", isChecked: true),  
    ChecklistItem(name: "Soccer practice"),  
    ChecklistItem(name: "Eat ice cream", isChecked: true),  
]
```

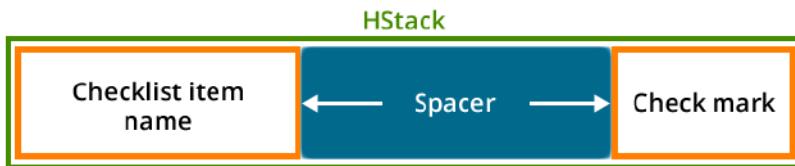
Showing an item’s “checked” status

Now that the checklistItems array is filled with checklistItem instances instead of Strings, we need to update the way that ContentView displays checklist items. Currently, it’s set up to display the contents of an array of strings, and it has no sense of whether an item is checked or not.

Here's the part of `ContentView`'s `body` property that displayed the contents of `checklistItems` when it was an array of strings:

```
List {  
    ForEach(checklistItems, id: \.self) { item in  
        Text(item)  
    }  
    .onDelete(perform: deleteListItem)  
    .onMove(perform: moveListItem)  
}
```

We need to change the contents of the `ForEach` view so that it displays both the name and “checked” status of each checklist item. The name should appear on the left side of the row, while the checkmark should appear on the right side. This sounds like a job for an `HStack`, a couple of `Text` views and a `Spacer` between them, arranged like this:



The `HStack` containing the items in a checklist row

- Change the `ForEach` view in `body` to the following:

```
ForEach(checklistItems, id: \.self) { checklistItem in  
    HStack {  
        Text(checklistItem.name)  
        Spacer()  
        if checklistItem.isChecked {  
            Text("✓")  
        } else {  
            Text("□")  
        }  
    }  
    .onDelete(perform: deleteListItem)  
    .onMove(perform: moveListItem)
```

To get the emoji, type **control+⌘+space** to get the emoji selector and enter **check** into the **Search** text field. To get the character, enter **square** into the emoji selector’s **Search** text field and scroll through the results to find it.

You’re almost ready to run the app and see the results of the changes you made. But first, there’s the matter of this error message:

```
ForEach(checklistItems, id: \.self) { checklistItem in
    HStack {
        Text(checklistItem.name)
        Spacer()
```

What does this error message mean?

Hooray for cryptic error messages! What Xcode is trying to tell you is that it’s running into trouble on this line:

```
ForEach(checklistItems, id: \.self) { checklistItem in
```

The part of the line where it’s running into trouble is `id: \.self`. The `id` parameter of `ForEach` tells SwiftUI how to identify each element in the data provided to it, which in this case is `checklistItems`. When `checklistItems` was an array of strings, we told `ForEach` to simply use the value of the string as a way of distinguishing one element from another, and it worked. Now that `checklistItems` is an array of `ChecklistItem` instances, `\.self` refers to a whole `ChecklistItem` instance, which is a blob of data.

We can fix this by changing the `id` parameter so that SwiftUI distinguishes between each `ChecklistItem` instance by its `name` property.

► Change the `ForEach` line to the following:

```
ForEach(checklistItems, id: \.self.name) { checklistItem in
```

This revised line of code says “loop through all the items in `checklistItems`, using each item’s `name` property to uniquely identify it, and within each loop through `checklistItems`, put the current item inside the `checklistItem` variable.”

You should notice that Xcode’s cryptic error message has disappeared. Will the app compile and run? There’s an easy way to find out:



- Run the app. Items in the list now have a checked and unchecked status:



The app now displays items’ “checked” status

What happens when two checklist items have the same name?

Let’s look at the `ForEach` line again:

```
ForEach(checklistItems, id: \.self.name) { checklistItem in
```

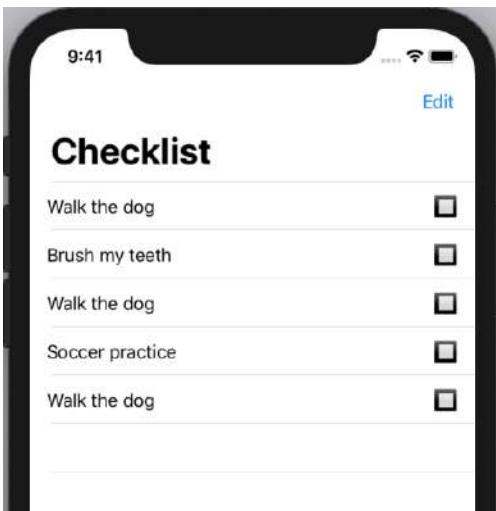
As I said earlier, setting the `id` parameter to `\.self.name` tells `ForEach` to use each item’s `name` property as a way of uniquely identifying it. What happens if two or more items have the same name? Let’s find out.

- Change the declaration of `checklistItems` so that “Walk the dog” appears three times, with two of them checked:

```
@State var checklistItems = [  
    ChecklistItem(name: "Walk the dog"),  
    ChecklistItem(name: "Brush my teeth"),  
    ChecklistItem(name: "Walk the dog", isChecked: true),  
    ChecklistItem(name: "Soccer practice"),  
    ChecklistItem(name: "Walk the dog", isChecked: true),  
]
```

Before you run the app, try to guess what this change will do.

- Run the app. You’ll see this:



The checklist, with multiple “Walk the dog” items, all unchecked

The checklist has three “Walk the dog” items in the right places, but they’re all unchecked. That’s because the app is identifying items by name, and the first “Walk the dog” item it saw was the unchecked one. It thinks that the second and third instances, both of which are supposed to be checked, are the same instance as the first one, so it thinks they’re all unchecked.

If you’ve ever been in a situation with someone with the same name as you and someone called out your name, you know this sort of confusion.

A better identifier for checklist items

There’s a simple fix for this, and it involves giving each `ChecklistItem` instance a unique “fingerprint” so that it can be distinguished from other instances, even those with identical `name` and `isChecked` properties.

- Change the declaration of `ChecklistItem` so that it looks like this:

```
struct ChecklistItem: Identifiable {
    let id = UUID()
    var name: String
    var isChecked: Bool = false
}
```

You just made two changes to `ChecklistItem`. The first is in the first line:

```
struct ChecklistItem: Identifiable {
```

The line still defines a `struct` named `ChecklistItem`. The addition to the end of the line says that `ChecklistItem` is also a kind of `Identifiable`. Remember, in most cases in Swift, the “`:`” character means “is a kind of.”

You’re probably wondering what `Identifiable` is. It’s a protocol, which in case you’ve forgotten, is an agreement for an object blueprint to provide some kind of feature or service by including specific properties and methods. The `Identifiable` protocol is an agreement that an object blueprint *adopts* or *conforms to* to guarantee that all its instances can be uniquely identified — hence the name “`Identifiable`.”

`Identifiable` is a simple protocol. For an object blueprint to adopt it, it needs to do only one thing: Include an `id` property whose value is guaranteed to be different for every object. Luckily, Apple operating systems have a built-in `struct` called `UUID`, which generates a universally unique value (a `UUID`, short for “universally unique identifier”) every time it’s called. And I’m not kidding. By *universally unique*, I mean that if you took billions of `UUID` generators and had them generate billions of `UUIDs` a day for billions of years, the odds of any two of them generating the same `UUID` would still be practically zero.

This brings us to the second change to `ChecklistItem`, which is the addition of this line:

```
let id = UUID()
```

This adds a property named `id` to `ChecklistItem`. The `let` makes it a constant, which means its value can be set only once when the object is created. `UUID()` creates a new instance of `UUID`, which creates a new universally unique identifier value. This value is put into `id`.

`id` is a constant, and its value is set when the `ChecklistItem` instance is created. This means that you don’t have to set it when creating a new `ChecklistItem` instance.

You won’t even get the opportunity to set its value. As this Xcode screenshot shows, there won’t be an option to set `id`’s value during instantiation:

```
ChecklistItem()
[M] ChecklistItem (name: String)
[M] ChecklistItem (name: String, isChecked: Bool)
```

There’s no option to set a predefined constant of a struct during instantiation

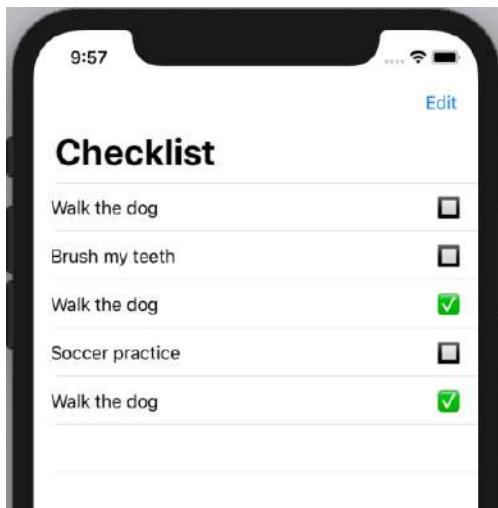
With these changes, we’ve upgraded `ChecklistItem` so it now comes with a “fingerprint” in the form of the `id` property that uniquely identifies every instance. With this change comes a bonus: You no longer have to tell the `ForEach` view how to uniquely identify instances of `ChecklistItem` anymore, because they now conform to the `Identifiable` protocol.

► Change the `ForEach` line to the following:

```
ForEach(checklistItems) { checklistItem in
```

Note the change: It’s now `ForEach(checklistItems)` instead of `ForEach(checklistItems, id: \.self.name)` because each `ChecklistItem` now has the ability to uniquely identify itself to `ForEach`.

► Run the app. Now that each `ChecklistItem` instance has its own unique identifier, the app can properly distinguish between items, even if they have the same name:



The checklist with properly identified multiple “Walk the dog” items

Using a little less code with the ternary conditional operator

Here’s the code in the `ForEach` view that determines whether the checked or unchecked emoji is displayed for a checklist item:

```
if checklistItem.isChecked {  
    Text("✅")  
} else {  
    Text("◻")  
}
```

The pattern — “if this condition is met, use this value; otherwise use this other value” — is one you’ll often use in programming. In fact, it’s used so often that Swift and many other programming languages use a special shorthand that condenses this sort of decision down to a single line. Using this shorthand, replace the code above with the following:

```
Text(checklistItem.isChecked ? "✅" : "◻")
```

This shorthand is called the *ternary conditional operator*, or *ternary operator*.

“Ternary” refers to the fact that it has three parts:

- A condition that is evaluated as either `true` or `false`. This is the part that comes before the `?`.
- The “true” outcome. This is the output if the condition evaluates to `true`, and it appears between the `?` and the `:`.
- The “false” outcome. This is the output if the condition evaluates to `false`, and it appears after `:`.

With the ternary operator, what once took five lines of code now takes just one. When you look at others’ code, which is a great way to learn, you’ll find that many programmers prefer to use the ternary operator whenever possible.

A quick check before moving on

With `Checklist` now able to track the “checked” status of checklist items, you’re a little closer to a working checklist app.



Before moving to the next step — giving the user the ability to check and uncheck items — let’s restore the checklist and review the code. Remember, it has some duplicate items right now.

Restoring the checklist

- Change the declaration for the `ChecklistItems` array back to the original:

```
@State var checklistItems = [
    ChecklistItem(name: "Walk the dog", isChecked: false),
    ChecklistItem(name: "Brush my teeth", isChecked: false),
    ChecklistItem(name: "Learn iOS development", isChecked: true),
    ChecklistItem(name: "Soccer practice", isChecked: false),
    ChecklistItem(name: "Eat ice cream", isChecked: false),
]
```

Reviewing the code

The code in **ContentView.swift**, minus the comments at the start, should look like this:

```
import SwiftUI

struct ChecklistItem: Identifiable {
    let id = UUID()
    var name: String
    var isChecked: Bool = false
}

struct ContentView: View {

    // Properties
    // =====

    @State var checklistItems = [
        ChecklistItem(name: "Walk the dog", isChecked: false),
        ChecklistItem(name: "Brush my teeth", isChecked: false),
        ChecklistItem(name: "Learn iOS development", isChecked: true),
        ChecklistItem(name: "Soccer practice", isChecked: false),
        ChecklistItem(name: "Eat ice cream", isChecked: true),
    ]

    // User interface content and layout
    var body: some View {
        NavigationView {
            List {
```

```
ForEach(checklistItems) { checklistItem in
    HStack {
        Text(checklistItem.name)
        Spacer()
        Text(checklistItem.isChecked ? "✓" : "□")
    }
    .onTapGesture {
        print("checklistitem name: \(checklistItem.name)")
    }
    .onDelete(perform: deleteListItem)
    .onMove(perform: moveListItem)
}
.navigationBarItems(trailing: EditButton())
.navigationBarTitle("Checklist")
.onAppear() {
    self.printChecklistContents()
}
}

// Methods
// =====

func printChecklistContents() {
    for item in checklistItems {
        print(item)
    }
}

func deleteListItem(whichElement: IndexSet) {
    checklistItems.remove(atOffsets: whichElement)
    printChecklistContents()
}

func moveListItem(whichElement: IndexSet, destination: Int) {
    checklistItems.move(fromOffsets: whichElement, toOffset:
destination)
    printChecklistContents()
}

// Preview
// =====

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

If you’re new to programming in general, this may look like a lot of code. You might be surprised to discover that a checklist app that can do as much as this one, even in its incomplete state, would require considerably more code if you did it using UIKit (the previous way of writing iOS apps), on Android or various web application frameworks.

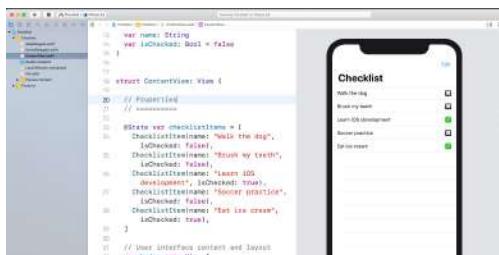
Checking the Canvas

If you haven’t been looking at the app in the Canvas lately, now’s a good time! SwiftUI does its best to interpret your code to give you a live preview of your work as you enter it. If you don’t see the Canvas, show it by selecting it in the menu in the upper right corner of the editor:



Showing the Canvas

Press the **Resume** button, and you should see your app:



Looking at the code and the Canvas

Toggling checklist items

Finding out when the user tapped a list item

The app now tracks each item’s “checked” status and can display it to the user. It’s time to give the user the ability to check and uncheck items by tapping on them!

Let’s look at the `ForEach` view inside `body`:

```
ForEach(checklistItems) { checklistItem in
    HStack {
```

```
    Text(checklistItem.name)
    Spacer()
    Text(checklistItem.isChecked ? "✅" : "◻")
}
.onTapGesture {
    print("checklistitem name: \(checklistItem.name)")
}
.onDelete(perform: deleteListItem)
.onMove(perform: moveListItem)
```

For every item in the bundle of data that you pass to `ForEach`, which in this case is `checkListItems`, it creates a view as defined by the stuff in the braces that follow the `ForEach` keyword, which is this code:

```
{ checklistItem in
    HStack {
        Text(checklistItem.name)
        Spacer()
        Text(checklistItem.isChecked ? "✅" : "◻")
    }
    .onTapGesture {
        print("checklistitem name: \(checklistItem.name)")
    }
}
```

The `checklistItem` at the start of this block of code contains the current checklist item. The first time through the loop, `checklistItem` contains the “Walk the dog” checklist item.

The second time, it contains the “Brush my teeth” item, and so on. As a result, each item in the checklist gets its own view: An `HStack` containing two `Text` views that show the item’s name and “checked” status, and a `Spacer` between them.

There are also calls to methods at the end of the `ForEach` view:

```
.onDelete(perform: deleteListItem)
.onMove(perform: moveListItem)
```

These attach instructions to each item in the `List` for when the user opts to delete and move list items, and we wrote those instructions in the previous chapter.

If there are methods for responding to events where the user deletes or moves a list item, there must be one for responding to the user tapping a list item.

There is, and it's called `onTapGesture()`, and it's a method that all `Views` have. Here's how we'll use it:

```
.onTapGesture {  
    // Code to perform when the user taps a list item goes here  
}
```

Let's start with something simple: We'll print “The user tapped a list item!” to Xcode's debug console whenever the user taps an item.

► Update `body` so that it looks like this:

```
var body: some View {  
    NavigationView {  
        List {  
            ForEach(checklistItems) { checklistItem in  
                HStack {  
                    Text(checklistItem.name)  
                    Spacer()  
                    Text(checklistItem.isChecked ? "✓" : "□")  
                }  
            }  
            .onDelete(perform: deleteListItem)  
            .onMove(perform: moveListItem)  
            .onTapGesture {  
                print("The user tapped a list item!")  
            }  
        }  
        .navigationBarItems(trailing: EditButton())  
        .navigationBarTitle("Checklist")  
        .onAppear() {  
            self.printChecklistContents()  
        }  
    }  
}
```

The change you made was adding a call to `onTapGesture()` immediately after the call to `onMove()`:

```
.onTapGesture {  
    print("The user tapped a list item!")  
}
```

- Run the app and tap some list items. Look at the debug console in Xcode. You should see “The user tapped a list item!” for every time you tapped a list item:

```
44      .onTapGesture {
45          print("The user tapped a list item!")
46      }
47  }
48 .navigationBarItems(trailing: EditButton())
49 .navigationBarTitle("Checklist")
50 .onAppear() {
51     self.printChecklistContents()
52 }
53 }
54 }
```



The user tapped a list item

Finding out which item the user tapped

It's good to know that the user tapped a list item, but it's even better to know *which* item.

The `checklistItem` variable inside the `ForEach` view contains the current list item, so we should be able to use it to identify the tapped item.

- Change `onTapGesture()` so that its `print` function displays the name of the current item:

```
.onTapGesture {
    print("The user tapped \(checklistItem.name).")}
```

Xcode will complain, showing you an error message that says “Use of unresolved identifier ‘`checklistItem`’”:

```
.onTapGesture {
    print("The user tapped \(checklistItem.name).") } ✖ Use of unresolved identifier 'checklistItem'
```

Xcode says that checklistItem is unresolved

This is Xcode's terribly technical way of saying, “I have no idea what you mean by `checklistItem`.” If your response is “But it's *right there!*,” I feel your pain.

If you take a closer look at the code, you’ll see the reason behind Xcode’s confusion. A variable, constant, or method that is defined within a set of braces is “visible” — or as programmers say, *in scope* — only to code within the same set of braces. `checklistItem`’s visibility or *scope* is limited to the `ForEach` braces:

```
ForEach(checklistItems) { checklistItem in
    HStack {
        Text(checklistItem.name)
        Spacer()
        Text(checklistItem.isChecked ? "✓" : "□")
    }
}
.onDelete(perform: deleteListItem)
.onMove(perform: moveListItem)
.onTapGesture {
    print("The user tapped \(checklistItem.name).")
}
```

checklistItem’s scope

The `onTapGesture()` method call lives outside the braces where `checklistItem` is in scope. If we want to know which item the user tapped, we’ll need to use `onTapGesture()` somewhere inside those braces.

The `HStack` that makes up a list row is a `View`, which means that it has an `onTapGesture()` method. Better still, it’s inside the braces where `checklistItem` is in scope. Let’s move the call to `onTapGesture()` there!

► Update body so that it looks like this:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems) { checklistItem in
                HStack {
                    Text(checklistItem.name)
                    Spacer()
                    Text(checklistItem.isChecked ? "✓" : "□")
                }
                .onTapGesture {
                    print("The user tapped \(checklistItem.name).")
                }
            }
            .onDelete(perform: deleteListItem)
            .onMove(perform: moveListItem)
        }
        .navigationBarItems(trailing: EditButton())
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.printChecklistContents()
        }
    }
}
```

```
    }
}
```

Note the change: The call to `onTapGesture()` is now attached to the `HStack` view instead of the `ForEach` view. `onTapGesture` is called whenever the user taps one of the `HStacks` in the list. Don’t take my word for it, though — try it out for yourself!

- Run the app, tap some list items, and look at Xcode’s debug console, which shows you which item the user tapped.

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems) { checklistItem in
                HStack {
                    Text(checklistItem.name)
                    Spacer()
                    Text(checklistItem.isChecked ? "✓" : "□")
                }
                .onTapGesture {
                    print("The user tapped \(checklistItem.name).")
                }
            }
        }
    }
}
```

```
The user tapped Soccer practice.
The user tapped Learn iOS development.
The user tapped Brush my teeth.
The user tapped Walk the dog.
```

Seeing which item the user tapped

Now that you know which item the user tapped, it’s time to check or uncheck it.

Checking and unchecking a checklist item

Tapping an item in the list should change its “checked” status. If the item is unchecked, tapping it should change it to checked. Conversely, tapping a checked item should uncheck it.

The `isChecked` property determines a checklist item’s “checked” status. Setting it to `true` checks the item, and setting it to `false` unchecks it. With that in mind, we could code `onTapGesture()` like so:

```
.onTapGesture {
    if checklistItem.isChecked {
```



```
    checklistItem.isChecked = false
} else {
    checklistItem.isChecked = true
}
```

However, there's a better, shorter way. Boolean (`Bool`) values have a method called `toggle()` which does the same thing, toggling the value from `true` to `false` and vice versa. This reduces all those lines of the `if` statement above to a single line. Lets use that instead.

- Change the call to `onTapGesture()` so that it uses `toggle()` to change the item's `isChecked` property:

```
.onTapGesture {
    checklistItem.isChecked.toggle()
}
```

Once again, Xcode has an issue with what you just did. Lets have a look at what the issue is this time.

```
.onTapGesture {
    if checklistItem.isChecked {
        checklistItem.isChecked.toggle()
    }
}
```

① Cannot use mutating member on immutable value:
'checklistItem' is a 'let' constant

checklistItem is a let constant

Xcode's error message, “Cannot use mutating member on immutable value: ‘`checklistItem`’ is a ‘let’ constant,” is telling you that `checklistItem` is a constant, which means you can't change any of its properties, including `isChecked`. You can only see what its properties contain.

You'll often run into situations like this when programming. You'll come up with a solution that you think *should* work, but when you code it, the compiler throws this kind of seemingly intractable obstacle in your way. This is the time to step back, look at the code, and see if it provides you with another solution.

First, think about what we *can* do with `checklistItem`. We can read its properties, which are:

- `id`: The automatically generated universally unique identifier for the item.
- `name`: The name of the item. We used this earlier to print the name of the tapped item in the Xcode console.
- `isChecked`: The “checked” status of the item.

Then you should ask yourself: Is there another way to access a given checklist item? There might be, by way of the array of checklist items, `checklistItems`. The entire `ContentView` struct is its scope.

It’s declared as a `var` property, which means it’s a variable, and so are its contents. Accessing a given element of `checklistItems` is done using *array notation*: `checklistItems[0]` is the first element of the array, `checklistItems[1]` is the second, `checklistItems[2]` is the third, and so on.

Let’s test this idea by changing `onTapGesture()` so that when the user taps a list item, the first item in the list — `checklistItems[0]` — is toggled.

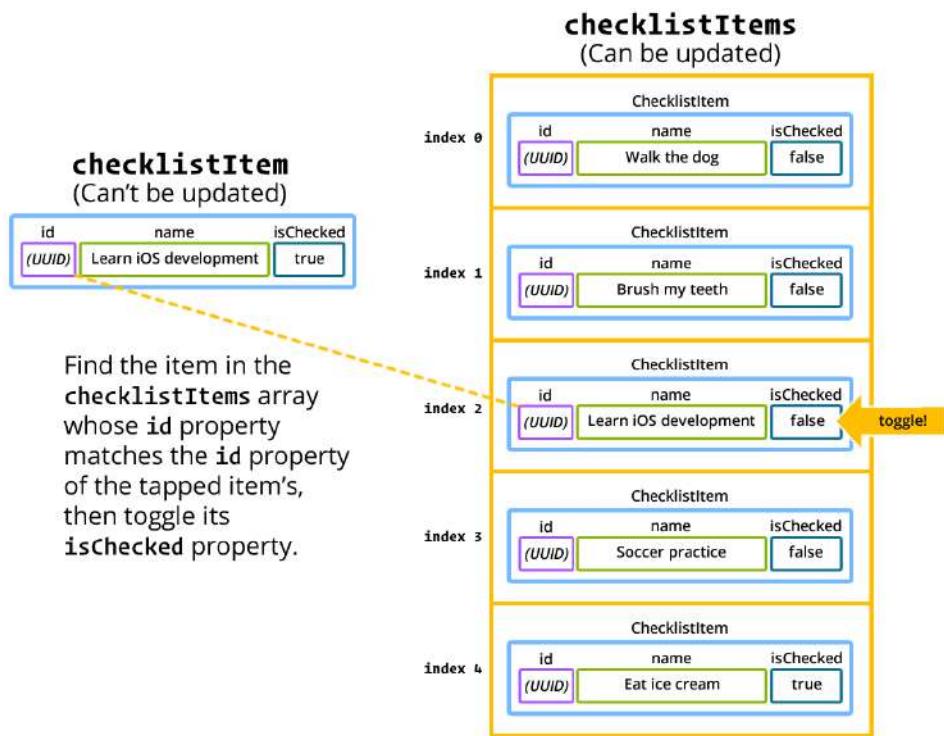
► Change the call to `onTapGesture()` to the following:

```
.onTapGesture {
    self.checklistItems[0].isChecked.toggle()
}
```

► Run the app and tap any list item. The first item in the list, “Walk the dog,” should toggle between checked and unchecked.

Now we’re getting somewhere. We now know that if you know the index (the element number, which starts at 0) of the item in `checklistItems`, you can change the item’s “checked” status. The problem is that there’s no such information inside the single `checklistItem` we get inside the `ForEach` view. All we have are `checklistItems` properties.

Look again at `checklistItem`’s properties. That first one, `id`, uniquely identifies it. Couldn’t we use it as a “search term” to scan through the items in the `checklistItems` array to find the matching item, and then toggle that item’s `isChecked` property?

*Finding the matching item in checklistItems*

Fortunately, the answer is “yes,” and there’s a way to do it in a couple of lines of code.

Swift arrays have many built-in methods for all sorts of purposes, including inspecting their contents, accessing one or more of their elements, adding and removing elements and *finding* specific elements. This final method is the most important for our immediate needs. One of these methods is `firstIndex(of:)`, which gives you the index of the first element in the array that matches the given criteria. Remember, an array index is the number that specifies an element in an array.

Suppose we had an array named `myList` that was defined this way:

```
let myList = [
  "Alpha",
  "Bravo",
  "Charlie",
  "Delta",
]
```

If you wanted to find the index of “Charlie” in `myList`, you’d use this code:

```
let charlieIndex = myList.firstIndex(of: "Charlie")
```

“Charlie” is the third item in `myList`, which means its index is **2**, so the value of `charlieIndex` is set to **2**. As a reminder, array elements begin at **0**.

What happens if you search for an item that *isn’t* in the array, say “Egbert”?

```
let egbertIndex = myList.firstIndex(of: "Egbert")
```

Since “Egbert” isn’t in `myList`, it doesn’t have an index. When this happens, `firstIndex(of:)` returns a value called `nil`, and that’s the value put into `egbertIndex`.

In case you were wondering how I knew about `firstIndex(of:)`, I didn’t spring forth from the womb as a fully-formed Swift programmer. Instead, I looked at the [Arrays](#) section of Apple’s Swift documentation, located at developer.apple.com/documentation/swift/array. If you ever have a question about a Swift language feature, the developer docs are an excellent place to start.

Introducing `nil` and its friend, `if let`

`nil` is a special value, and it means “no value.” `nil` doesn’t mean **0**, because **0** is a value. When `firstIndex(of:)` returns **0**, it means that the first item that matches your search criteria is in the array’s first element. When `firstIndex(of:)` returns `nil`, it means that the array doesn’t have any items that match your search criteria in the array’s first element.

You’ll learn more about `nil` and optional types later on.

You’ll often find yourself writing code that follows this pattern: “If an operation produced a result with a value, do something with that result.” In many programming languages, you’d have to write this sort of code this way:

```
let result = someOperation()
if result != nil {
    // Do something with result
}
```

In case you were wondering, `!=` means “is not equal to.”



In the code above, we’re calling `someOperation()`, which returns a result. If that result is *not nil*, the code inside the braces with the “Do something with result” comment is executed. If the result is `nil`, the code inside the braces is skipped entirely.

Swift has the `if let` construct, which makes this sort of code a little more concise. Here’s how you’d rewrite the code above using `if let`:

```
if let result = someOperation() {  
    // Do something with result  
}
```

Just like the code before it, this code calls `someOperation()` and puts its result in the variable `result`. If `result`’s value is not `nil`, the code inside the braces with the “Do something with result” comment is executed. Otherwise, the code inside the braces is skipped.

We’re going to use `if let` to toggle the “checked” status of a checklist item in `checklistItems` if one whose `id` matches the `id` of the tapped item is found by a close cousin of the `firstIndex(of:)` method.

Finding a specific item in `checklistItems`

The `firstIndex(of:)` method is good for doing simple matches. Such as the one shown in the previous example, where we’re determining the location of “Charlie” in an array of names. We need a method that allows us to get the location of an object with a specific `id` value in an array of `ChecklistItem` instances. That method is the `firstIndex(where:)` method.

The `firstIndex(where:)` method follows this format:

```
result = firstIndex(where: { Code for search criteria that produces a true or false  
result goes here } )
```

Where `firstIndex(of:)` is useful for finding the first exact match in an array, `firstIndex(where:)` lets you get *really* specific. In a really fancy checklist app, you could use it to search for the first checklist item in the list that is checked, entered on a Tuesday, marked as high priority and features a cat picture. In this checklist app, you’ll use it to find the first item in the list with a specific `id` value.

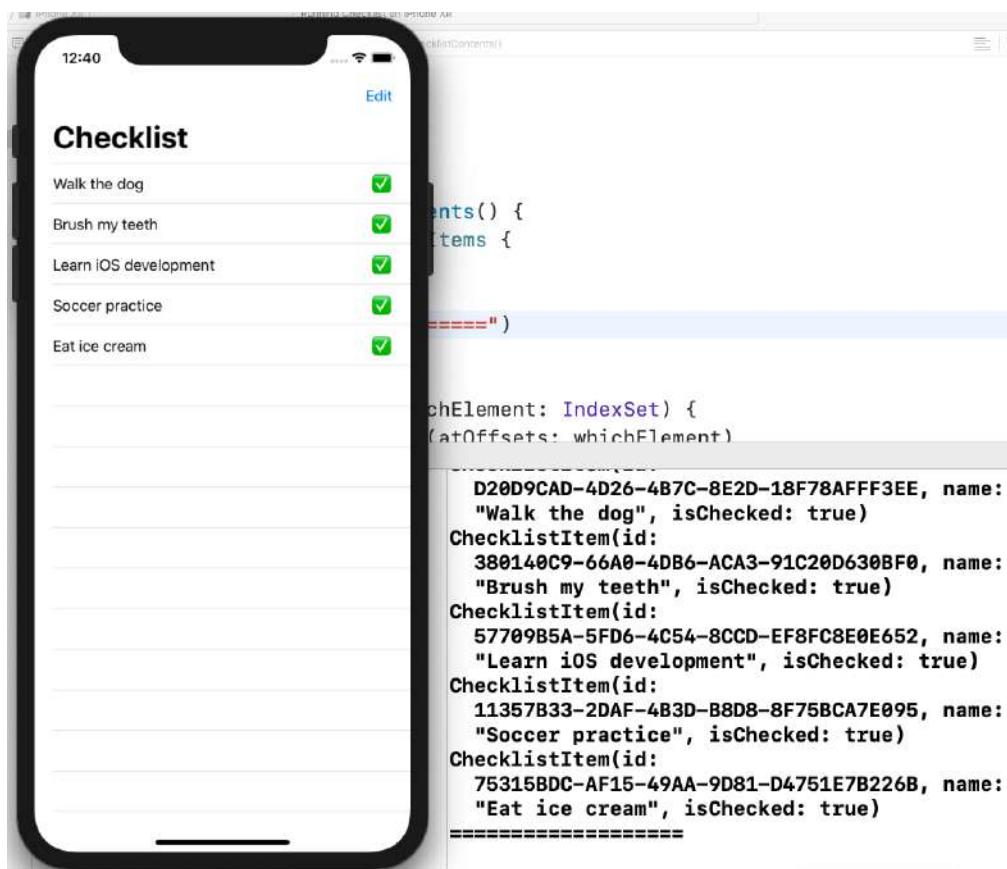
Using `firstIndex(where:)` is another one of those cases where showing it in action first is better than telling you how to use it. So, that’s just what I’ll do:

- Change the call to `onTapGesture()` to this:

```
.onTapGesture {
    if let matchingIndex = self.checklistItems.firstIndex(where: {
        $0.id == checklistItem.id }) {
        self.checklistItems[matchingIndex].isChecked.toggle()
    }
    self.printChecklistContents()
}
```

- Run the app. Tap on any of the item names or checkboxes to check and uncheck them. You might notice that the blank space between the name and checkbox doesn’t respond to taps — we’ll fix that shortly.

You can confirm that the items’ `isChecked` properties are being updated by looking at Xcode’s debug console:



A working checklist, as seen in the Simulator and debug console

Now that it’s possible for the user to check and uncheck items, let’s look at the code that made it possible. Here’s the first line of the new code:

```
if let matchingIndex = self.checklistItems.firstIndex(where:  
{ $0.id == checklistItem.id }) {
```

Let’s take a closer look at funny-looking part of that line, namely:

```
{ $0.id == checklistItem.id }
```

You provide `firstIndex(where:)` with the code in the braces — in case you’ve forgotten, it’s called a *closure* — and it goes through the array, applying that code to each element. The `$0` is shorthand for “the first parameter passed to the closure,” which is the current array element.

The first time that `firstIndex(where:)` applies the code to the array, `$0` represents the “Walk the dog” checklist item, and its `id` property compared to the `id` property of the tapped item. The second time, `$0` represents the “Brush my teeth” checklist item, and the `id` property comparison is made. The third time, `$0` represents the “Learn iOS development” item, and once again, `id` properties are compared. This cycle continues until the code in the closure results in a `true` value or the code has been applied to every element in the array.

If `firstIndex(where:)` finds a checklist item in `checklistItems` whose `id` property matches the `id` property of the tapped checklist item (`checklistItem`), it returns the index of matching item in the `checklistItems` array. This value is stored in the constant `matchingIndex`.

Now that we have the index of the matching item in `checklistItems`, we can toggle its `isChecked` property:

```
self.checklistItems[matchingIndex].isChecked.toggle()
```

We now have a checklist that the user can actually check! It’s always nice when an app lives up to its name. There’s just one little user experience issue that we should fix.

Fixing the “dead zone”

For each row in the list, the space between the item’s name and its checkbox is a “dead zone.” Tapping on it doesn’t check or uncheck the checkbox. That’s an annoying quirk. It might make your user think that your app is broken, that you’re a terrible programmer and perhaps even put a curse on you, the accursed developer

and the seven generations to come after you. Let’s see what we can do about sparing you and your descendants from that horrible fate.

The solution was the result of some experimenting and guessing. Rather than drag you through my experimentation and guesswork, let me simply give you the summary.

Do you know how you can make the whole row tappable, rather than just the visible parts? *Give it a background color.*

I decided to set the row’s background color to white, which I did by adding this method call to the HStack that defines each row:

```
.background(Color.white) // This makes the entire row clickable
```

With this change, the body property should look like this:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklistItems) { checklistItem in
                HStack {
                    Text(checklistItem.name)
                    Spacer()
                    Text(checklistItem.isChecked ? "✓" : "□")
                }
                .background(Color.white) // This makes the entire row
                clickable
                .onTapGesture {
                    if let matchingIndex =
                        self.checklistItems.firstIndex(where: { $0.id ==
                            checklistItem.id }) {
                        self.checklistItems[matchingIndex].isChecked.toggle()
                    }
                    self.printChecklistContents()
                }
                .onDelete(perform: deleteListItem)
                .onMove(perform: moveListItem)
            }
            .navigationBarItems(trailing: EditButton())
            .navigationBarTitle("Checklist")
            .onAppear() {
                self.printChecklistContents()
            }
        }
    }
}
```

Why does this work? It’s because, in its default state, list rows are transparent. The white color of a default list row is actually the white color of the view that contains the whole user interface.

The standard for most user interfaces — not just iOS’ — is that transparent objects aren’t tappable or clickable. Giving the row a color means that its pixels are clickable, and giving it the same color as the background view makes the whole user interface seamless.

As you gain more experience programming, you’ll find that your ability to come up with these flashes of insight will improve. Practice, to twist the expression slightly, makes programmer.

Key points

In this chapter, you did the following:

- You created your first `struct`, and in the process, learned the difference between `structs` and objects or instances.
- You updated the user interface to show each checklist item’s name and “checked” status.
- You learned about the ternary operator.
- You used the `onTapGesture` method that `Views` have to detect when the user tapped on a row.
- You learned about methods for finding the first occurrence of an item in an array that met specific criteria.
- You got a look into the sort of problem-solving that goes hand in hand with writing programs. As you do more programming, you’ll get better at it!

As always, you can find the project files for the app at this stage under **10 - Checkable List** in the Source Code folder.

In the next chapter, we’ll handle the next big piece of missing functionality: Adding and editing checklist items. *Checklist* is beginning to look like a real app, isn’t it?

Chapter 11: The App Structure

Joey deVilla

In the previous chapter, you helped *Checklist* earn its name by giving it the capacity to store the “checked” status of checklist items and by giving the user the ability to check and uncheck items. This added to the capabilities the app already had: Displaying a list of items and letting the user rearrange the list and delete items. Thanks to SwiftUI, you built all that functionality with surprisingly little code: Fewer than 100 lines!

However, the app’s still missing some very important functionality. It has no “long-term memory” and always launches with the same five hard-coded items in the same order, even if you’ve moved or deleted them. There’s no way for the user to add new items or edit existing ones.

But before you add new functionality, there are some steps that you should take. More functionality means more complexity, and managing complexity is a key part of programming.

Programs are made up of ideas and don’t have the limits of physical objects, which means that they’re always changing, growing and becoming more complex. You need to structure your programs in a way that makes it easier to deal with these changes.

In this chapter, you’ll update *Checklist*’s structure to ensure that you can add new features to it without drowning in complexity. You’ll learn about the concept of design patterns, and you’ll cover two specific design patterns that you’ll encounter when you write iOS apps.

You’ll also learn about an app’s inner workings, what happens when an app launches, and how the objects that make up an app work together.



Design patterns: MVC and MVVM

All the code that you've written for *Checklist* so far lives in a single file:

ContentView.swift. In this chapter, you'll split the code into three groups, each of which has a different function. This will make your code easier to maintain in the future. Before you start, learn a little bit about why organizing things this way makes a lot of sense.

Different parts of the code do different things. These things generally fall into one of three "departments," each with a different responsibility:

- **Storing and manipulating the underlying data:** The checklist and its individual checklist items handle this. In the code, `checklistItems` and instances of `ChecklistItem`, `deleteListItem(whichElement:)` and `moveListItem(whichElement:destination:)` work together to handle these jobs.
- **Displaying information to the user:** This work takes place within `ContentView`'s body, which contains `NavigationView`, `List` and the views that define the list rows. Each of these includes each item's name and checkbox.
- **Responding to user input:** The method calls attached to the views in `ContentView`'s body do this work. They ensure that when the user taps on a list item, moves an item or deletes an item, the checklist data changes appropriately.

Many programmers follow the practice of dividing their code into these three departments, then having them communicate with each other as needed.

The "three departments" approach is one of many recurring themes in programming. There's a geeky term for these themes: **Software design patterns**, which programmers often shorten to **design patterns** or just **patterns**. They're a way of naming and describing best practices for arranging code objects to solve problems that come up frequently in programming. Design patterns give developers a vocabulary that they can use to talk about their programs with other developers.

Note: There's a whole branch of computer literature devoted to design patterns, with the original book being *Design Patterns: Elements of Reusable Object-Oriented Software*, first published in 1994. Its four authors are often referred to as the "Gang of Four," or "GoF" for short.

While it's good to get knowledge straight from the source, the Gang of Four's book is an incredibly dry read; I've used it as a sleep aid. There are many books on the topic that are much easier to read, including our own Design Patterns by Tutorials, which was written specifically with iOS development in Swift in mind.

The Model-View-Controller (MVC) pattern

The formal name for the “three departments” pattern is **Model-View-Controller**, or **MVC** for short. Each name represents a category of object:

- **Models:** These objects contain your data and any operations on that data. For example, if you’re writing a cookbook app, the model would consist of the recipes. In a game, it would be the design of the levels, the player score and the positions of the monsters.

Models can interact with each other, and the way in which they interact — the **business rules** or the **domain logic** — is also determined by the models. In a game, the player and opponent models determine how the players and their various opponents interact.

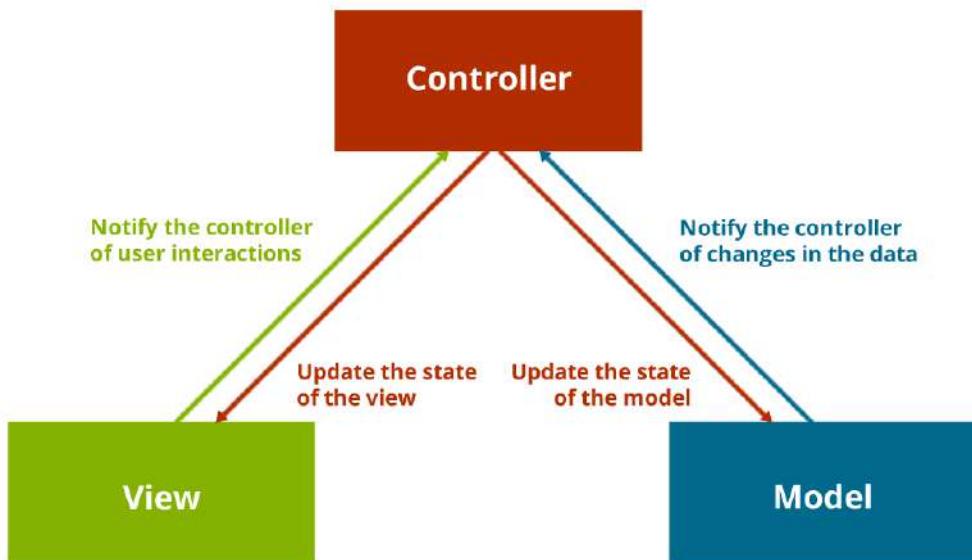
In *Checklist*, the checklists along with their to-do items and the “move” and “delete” operations form the data model. Models are the keepers of the knowledge in the system.

- **Views:** These are the visual part of the app: Text, images, buttons, lists and their rows and so on. In a game, the views form the visual representation of the game world, such as the monster animations or a frag counter.

A view can draw itself and respond to user input, but it shouldn’t handle any app logic. It should be “dumb” in the sense that it only knows how to show data to the user, without knowing anything about that data or making any decisions about what it’s displaying. Many different apps can all use views like Lists, because they’re not tied to a specific data model.

- **Controllers:** A controller is an object that connects your data model objects to the views. It listens to taps on the views, makes the data model objects do calculations in response, and updates the views to reflect the new state of your model.

Here's how model, view and controller objects fit together:



How the Model, View and Controller in MVC fit together

The flow of activity in MVC is circular. The user taps a button or enters information in the view, then the view notifies the controller. The controller interprets the user interaction and then contacts the model for the information it needs to complete the request. The model provides the information to the controller, which relays it to the view, which shows it to the user.

The Model-View-Controller pattern has been around since the 1970s. Most desktop, web and mobile apps that you use are built on this pattern, or something similar to it. That's because it does a good job of separating the various aspects of an app into more manageable chunks. This pattern makes life easier for both the solo developer, who has to manage a lot of code alone and developer teams, who have to work on the same app without getting in each other's way.

You'll use the Model-View-Controller pattern later in this book, when you build apps with the UIKit framework. SwiftUI uses a slightly different pattern, called MVVM.

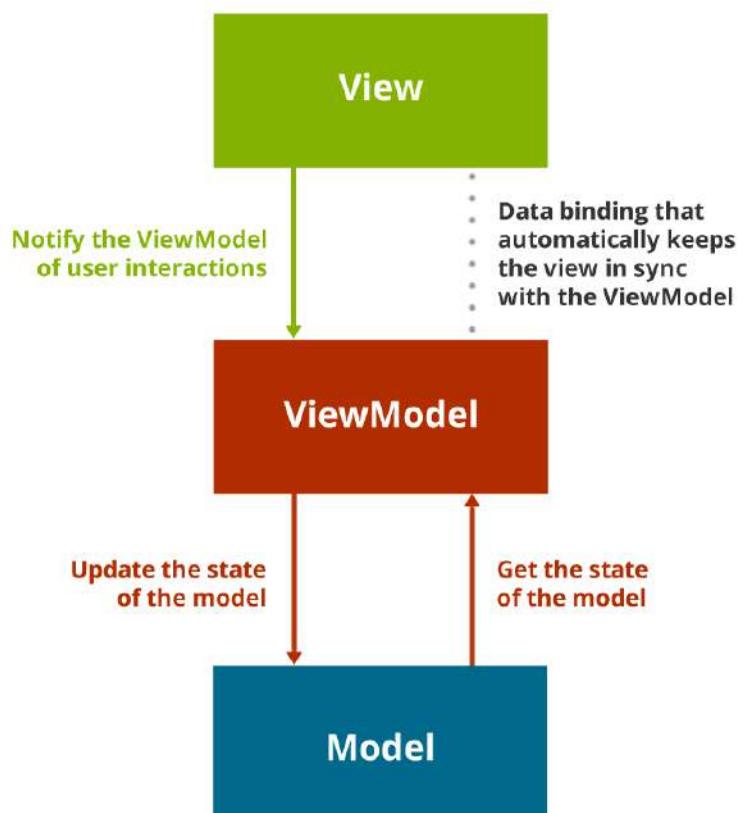
Model-View-ViewModel (MVVM)

Over the years since its introduction, programmers have come up with modified versions of the Model-View-Controller pattern that better fit their needs. One of these is **Model-View-ViewModel**, which is often shortened to *MVVM*.

Like Model-View-Controller, its name suggests that it has three different object categories:

- **Models:** MVVM models, like MVC models, contain the data for the app.
- **Views:** As with MVC, MVVM views present data to the user and accept user input. Unlike MVC, there's very little code in an MVVM view, since they focus on the user interface and nothing else.
- **ViewModels:** “ViewModel” is a clumsy name, but since the brightest minds in computer science haven't come up with a better one, we're stuck with it. The ViewModel gives the view the data and functionality it needs from the model, and nothing more. You can think of the ViewModel as the model's “customer service representative” and the view as the “customer.”

Here's how model, view and ViewModel objects fit together:



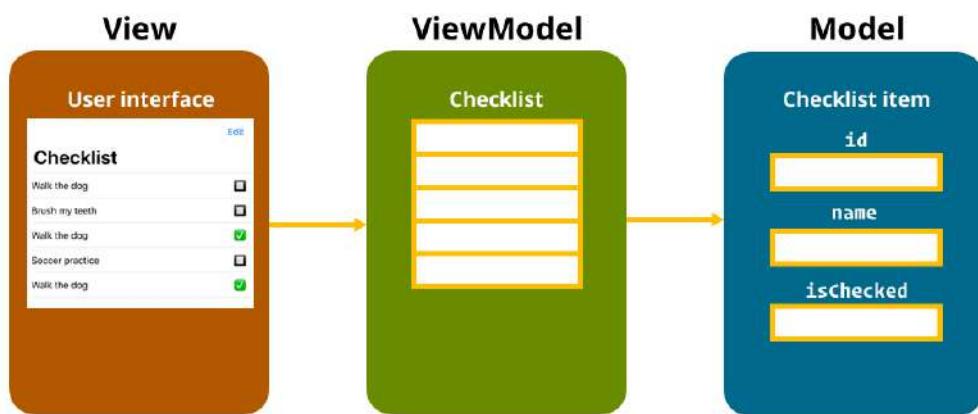
How the Model, View and ViewModel in MVVM fit together

The flow of activity in MVVM is often described as more linear than in MVC. In MVC, the Controller acts as a central hub and has to know about both the view and the model. In MVVM, the view knows only about the ViewModel, and the ViewModel knows only about the model.

This linear arrangement of one-way relationships makes it easier to maintain, modify and even swap out components. The flexibility that MVVM offers is one reason why it's popular with Windows and web app developers, and it's why Apple adopted this approach for the SwiftUI framework.

Using MVVM with Checklist

Here's how you'll split up *Checklist*'s code:



The model, view and ViewModel in Checklist

- The existing `ContentView` already acts as the app's view. To adopt MVVM for *Checklist*, you'll extract the code that makes up the ViewModel and the model and put each set of code into its own file.
- The ViewModel is an object that contains the properties and methods that the view needs to show data to the user and to respond to the user's actions. You'll extract the list of items and the methods that manipulate the list from `ContentView` and put them into their own ViewModel object, which will live in its own file.
- The model is an object representing individual checklist items. You'll extract `ChecklistItem` from `ContentView` and put it into its own file.

The end result will be an app that appears the same to the user, but to programmers, it'll be better organized and easier to maintain and to add features to.

Along the way, you'll learn about breaking up a project into easier-to-manage pieces, dive deeper into how objects work and learn what happens "under the hood" when the user launches an app.

There's a lot to do in this chapter, so go ahead and get started!

Renaming the view

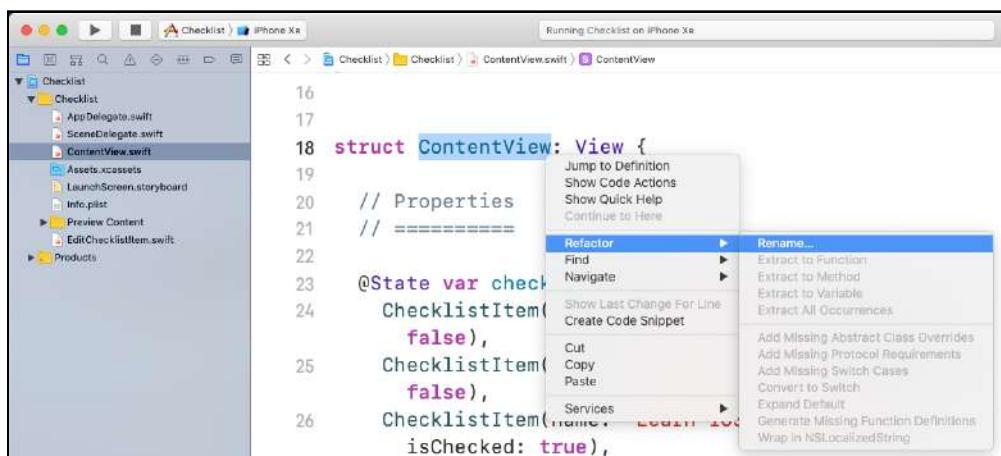
Both *Bullseye* and *Checklist* are based on Xcode's **Single View App** project template. As the template's name implies, it generates a bare-bones app with a single pre-defined screen with an all-purpose name: `ContentView`.

It's a good enough name for an app with a single screen, but a bit too generic for an app that will have multiple screens.

Since you're in the process of rearranging the app to fit the Model-View-ViewModel pattern, give the app's main view a more fitting name: `ChecklistView`.

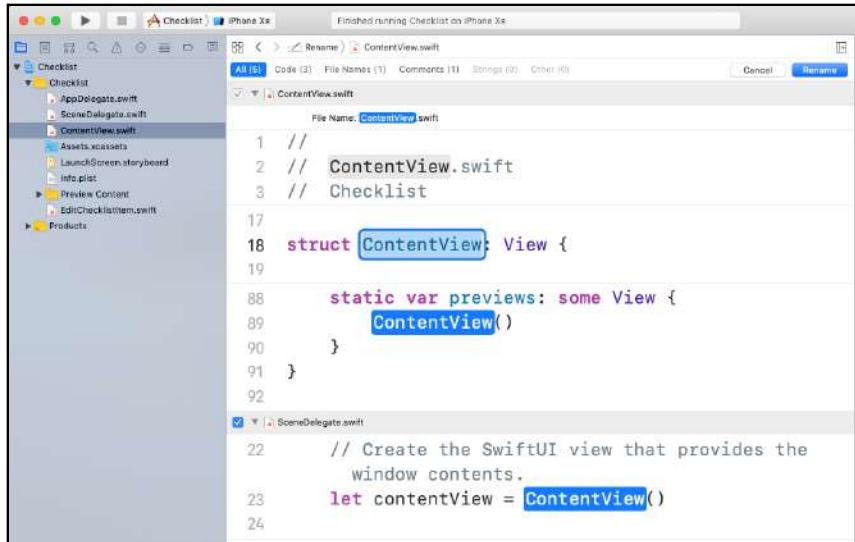
You can do this manually, by searching for `ContentView` throughout the project's files and renaming the `ContentView.swift` file itself, but that's a tedious and error-prone process. Instead, you'll rename it automatically using Xcode's refactoring tools.

► Open `ContentView.swift`. Find the opening line of `ContentView`. Select `ContentView` and right-click or control-click it. Select **Refactor** from the menu that appears, then select **Rename...** from the next menu:



The first step in renaming `ContentView`

Xcode will scan all the project files for any occurrence of the name `ContentView`. It will then display all these occurrences in the Editor pane so you can easily rename them.

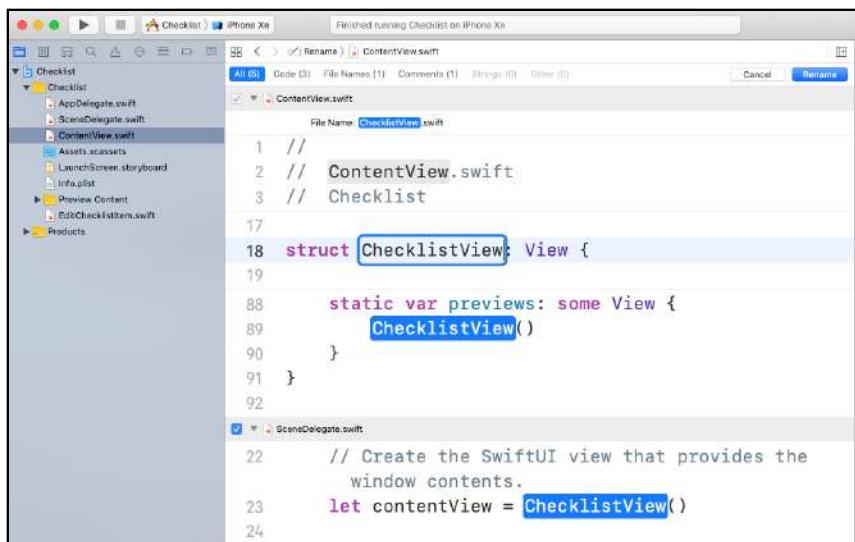


The screenshot shows the Xcode interface with the "Checklist" project selected. In the center, the "ContentView.swift" file is open in the editor. A "Rename" dialog is displayed over the code, with the current file name "ContentView.swift" highlighted. Below it, a new file name "ContentItem.swift" is typed. The code itself contains several instances of "ContentView". The "SceneDelegate.swift" file is also partially visible at the bottom of the editor.

```
// ContentView.swift
// Checklist
// ContentItem.swift
1 // ContentView.swift
2 // Checklist
3
4 struct ContentView: View {
5
6     static var previews: some View {
7         ContentView()
8     }
9 }
10
11 let contentView = ContentView()
12
13
14
15
16
17
18
19
20
21
22
23
24
```

Xcode shows you all the instances of the name `ContentView`

- Type `ChecklistView`. This change will be reflected in most of the instances of `ContentView`:



The screenshot shows the Xcode interface with the "Checklist" project selected. In the center, the "ContentView.swift" file is open in the editor. A "Rename" dialog is displayed over the code, with the current file name "ContentItem.swift" highlighted. Below it, a new file name "ChecklistView.swift" is typed. The code has been updated to reflect this change, with "ContentItem" instances replaced by "ChecklistView". The "SceneDelegate.swift" file is also partially visible at the bottom of the editor.

```
// ChecklistView.swift
// Checklist
// ContentItem.swift
1 // ContentItem.swift
2 // Checklist
3 // ChecklistView.swift
4
5 struct ChecklistView: View {
6
7     static var previews: some View {
8         ChecklistView()
9     }
10
11 }
12
13
14
15
16
17
18
19
20
21
22
23
24
```

Changing `ContentView`'s name to `ChecklistView`

- Click **Rename**. All the instances of `ContentView` that matter — in the code and in filenames — will be changed to `ChecklistView`.

There's a bug in Xcode's refactoring that causes it to fail to update the one instance of `ContentView` that appears in the comments. It won't affect the app, but you should change that instance manually, if only to be consistent.

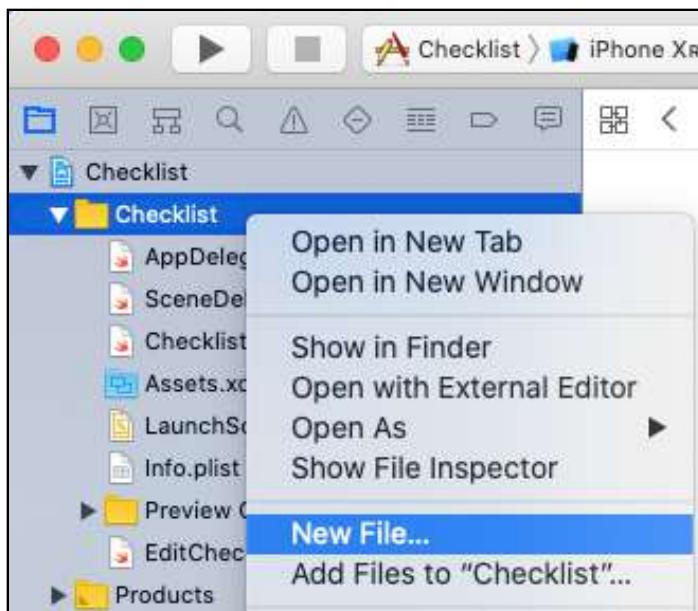
- Build and run to reassure yourself that you didn't break anything during the name change.

Adding a file for the model

Creating a file for the model

Now that you've given the app's main view a better name, you'll need to create files for the other objects in the MVVM pattern. You'll start by creating a file for the model's code.

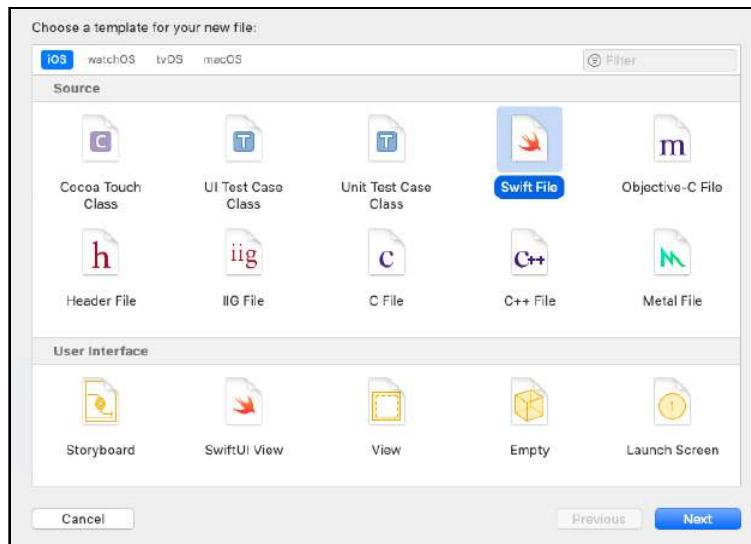
- Add a new file to the project by right-clicking or control-clicking on the **Checklist** folder in Xcode's Project Explorer. Select **New File...** from the menu that appears:



Add the second new file to the project

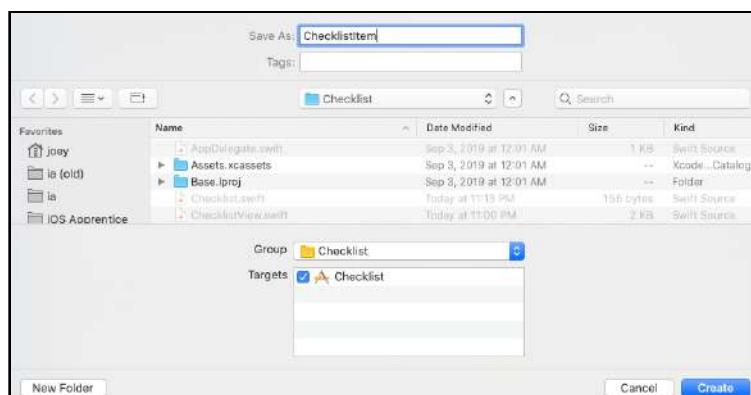
This time, you don't want to add a new view to the app, but rather to a class. This calls for adding a different kind of file to the project.

► In the window that appears, make sure that you've selected **iOS** then select **Swift File**, and *not SwiftUI View*. Unlike the SwiftUI View template, which comes with code to generate a simple “Hello World!” screen, this option gives you a mostly empty file. Click **Next**:



Select Swift File

► Enter **ChecklistItem** into the **Save As:** field. Make sure that you've selected **ChecklistItem** in the **Group** menu and in the **Targets** menu, then click **Create**:



Name the second new file 'ChecklistItem'

The project now has a new file named **ChecklistItem.swift**.

Moving the model code to the file

Now that you have a new file for the model, it's time to move its code there. Luckily, this is a simple process.

- Open **ChecklistView.swift** and cut the declaration of ChecklistItem:

```
struct ChecklistItem: Identifiable {
    let id = UUID()
    var name: String
    var isChecked: Bool = false
}
```

- Then paste it into **ChecklistItem.swift** so that the code in the file looks like this:

```
import Foundation

struct ChecklistItem: Identifiable {
    let id = UUID()
    var name: String
    var isChecked: Bool = false
}
```

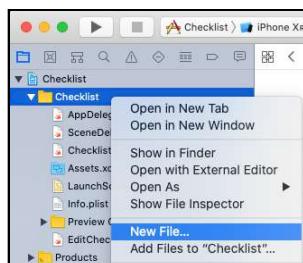
- Build and run the app. It still works because you haven't changed any code; you merely relocated the definition of ChecklistItem into its own file.

That takes care of the model code. It's now time to tackle the ViewModel!

Adding a file for the ViewModel

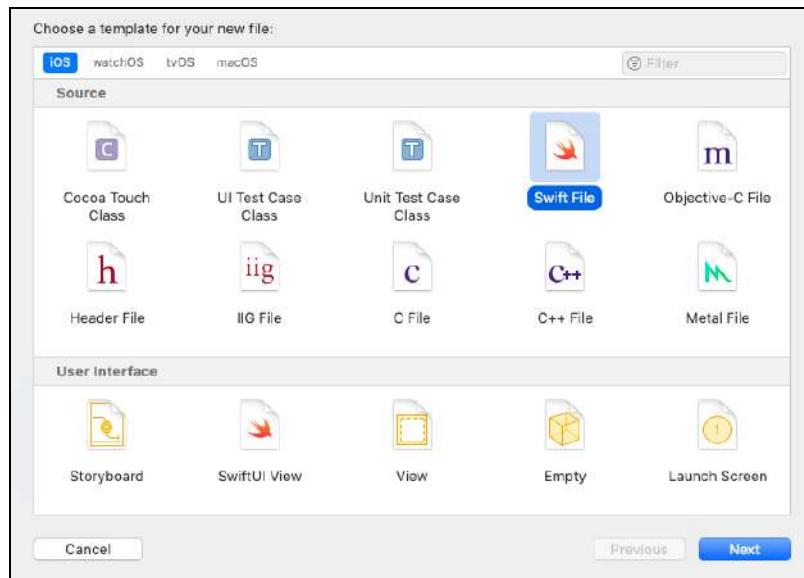
Your next step is to create a file for the ViewModel.

- Add another new file to the project by right-clicking or control-clicking the **Checklist** folder in Xcode's Project Explorer. Select **New File...** from the menu:



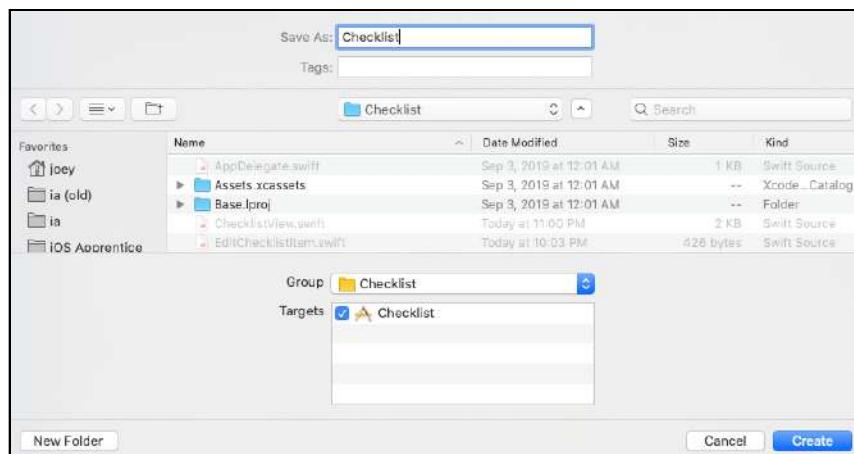
Add the first new file to the project

- Once again, make sure that you've selected **iOS** in the new window, then select **Swift File** and click **Next**:



Select Swift File

- Enter **Checklist** into the **Save As:** field. Make sure that you've selected **Checklist** in the **Group** menu and in the **Targets** menu, then click **Create**:



Name the first new file 'Checklist'

The project now has a new file named **Checklist.swift**. You just need to add the ViewModel code to it.

Moving the ViewModel code to the file

Add the following to **Checklist.swift**, just after the `import Foundation` line:

```
class Checklist: ObservableObject {  
}
```

A class is another kind of blueprint for objects. Like a struct, it has properties and methods and you can create class instances. In the spirit of “show, don’t tell,” continue with moving the ViewModel code into this class and see how it works in action before getting into a deeper discussion about classes.

You may not know much about classes yet, but you should have enough of a grasp of Swift syntax to know that you just added the definition of a class named `Checklist` and that it’s a kind of `ObservableObject`.

`ObservableObject` is a protocol, and as its name implies, when you use it on an object, another object can observe it for changes. That other object is called an observer.

In the MVVM pattern, the view observes the ViewModel. This observer/observed relationship binds the view and ViewModel so that when data that’s displayed to the user is updated in the ViewModel, the view updates itself automatically.

The ViewModel should contain all the functionality that the view needs to present information to the user. So far, that functionality is:

1. Displaying the checklist items.
2. Deleting an item.
3. Moving an item.
4. Toggling an item between “checked” and “unchecked”.

Start by moving the code for the first item on that list: Displaying the checklist items. The checklist items are stored in `checklistItems` in `ChecklistView`.

► Open **ChecklistView.swift**. Cut `checklistItems` in `ChecklistView`:

```
@State var checklistItems = [  
    ChecklistItem(name: "Walk the dog", isChecked: false),
```

```
ChecklistItem(name: "Brush my teeth", isChecked: false),
ChecklistItem(name: "Learn iOS development", isChecked: true),
ChecklistItem(name: "Soccer practice", isChecked: false),
ChecklistItem(name: "Eat ice cream", isChecked: true),
]
```

Xcode will show some error messages. Ignore them for now.

- Paste checklistItems into Checklist in **Checklist.swift**. Change @State to @Published.

The class should now look like this:

```
class Checklist: ObservableObject {

    @Published var checklistItems = [
        ChecklistItem(name: "Walk the dog", isChecked: false),
        ChecklistItem(name: "Brush my teeth", isChecked: false),
        ChecklistItem(name: "Learn iOS development", isChecked:
true),
        ChecklistItem(name: "Soccer practice", isChecked: false),
        ChecklistItem(name: "Eat ice cream", isChecked: true),
    ]
}
```

Marking a property in an `ObservableObject` as `@Published` means that making changes to that property notifies any observing objects. By marking `checklistItems` as `@Published`, any changes to it — toggling an item, deleting an item or moving an item — will update any views that are observing the `ViewModel`.

That takes care of all the `ViewModel` properties. It's time to move the `ViewModel` methods into `Checklist`.

- Open **ChecklistView.swift**. Cut the methods from `ChecklistView`:

```
func printChecklistContents() {
    for item in checklistItems {
        print(item)
    }
    print("====")
}

func deleteListItem(whichElement: IndexSet) {
    checklistItems.remove(atOffsets: whichElement)
    printChecklistContents()
}

func moveListItem(whichElement: IndexSet, destination: Int) {
```

```
    checklistItems.move(fromOffsets: whichElement, toOffset:  
destination)  
    printChecklistContents()  
}
```

- Then go back to **Checklist.swift** and paste the methods into Checklist, just after the checklistItems.

Checklist should now look like this:

```
class Checklist: ObservableObject {  
  
    @Published var checklistItems = [  
        ChecklistItem(name: "Walk the dog", isChecked: false),  
        ChecklistItem(name: "Brush my teeth", isChecked: false),  
        ChecklistItem(name: "Learn iOS development", isChecked:  
true),  
        ChecklistItem(name: "Soccer practice", isChecked: false),  
        ChecklistItem(name: "Eat ice cream", isChecked: true),  
    ]  
  
    func printChecklistContents() {  
        for item in checklistItems {  
            print(item)  
        }  
        print("=====")  
    }  
  
    func deleteListItem(whichElement: IndexSet) {  
        checklistItems.remove(atOffsets: whichElement)  
        printChecklistContents()  
    }  
  
    func moveListItem(whichElement: IndexSet, destination: Int) {  
        checklistItems.move(fromOffsets: whichElement, toOffset:  
destination)  
        printChecklistContents()  
    }  
}
```

The ViewModel, Checklist, now has the following:

- A property called checklistItems, which contains the checklist items.

These methods are all that the view currently needs to present information to the user and respond to the user's actions – exactly what a ViewModel provides to a view.



At this point, you have a file for the model that includes its object blueprint, **ChecklistItem.swift**, ChecklistItem, a file for the ViewModel with its object blueprint, **Checklist.swift** and Checklist. You put these together from bits and pieces extracted from **ChecklistView.swift** and ChecklistView.

Now, take a look at what's left of the view file and its object blueprint.

- Open **ChecklistView.swift** and look at ChecklistView. There's a lot less code and a few more errors:

```
struct ChecklistView: View {  
  
    // Properties  
    // ======  
  
    // User interface content and layout  
    var body: some View {  
        NavigationView {  
            List {  
                ForEach(checklistItems) { checklistItem in ① Use of unresolved identifier 'checklistItems'  
                    HStack {  
                        Text(checklistItem.name)  
                        Spacer()  
                        Text(checklistItem.isChecked ? "✓" : "◻")  
                    }  
                    .background(Color.white) // This makes the entire row clickable  
                    .onTapGesture {  
                        if let matchingIndex =  
                            self.checklistItems.firstIndex(where: { $0.id == checklistItem.id }) {  
                                self.checklistItems[matchingIndex].isChecked.toggle()  
                            }  
                            self.printChecklistContents()  
                        }  
                    }  
                    .onDelete(perform: deleteListItem) ② Use of unresolved identifier 'deleteListItem'  
                    .onMove(perform: moveListItem) ③ Use of unresolved identifier 'moveListItem'  
                }  
                .navigationBarItems(trailing: EditButton())  
                .navigationBarTitle("Checklist")  
                .onAppear() {  
                    self.printChecklistContents()  
                }  
            }  
        }  
    }  
}
```

ChecklistView's code and error messages

It appears that the view will need a little tweaking before Checklist's new MVVM setup will work. The problem is that there isn't a connection between the view, ChecklistView and the ViewModel, Checklist.

You'll establish that connection in a while, after you learn a little more about objects.

Structs and classes

Until this chapter, the only kind of object blueprint you've worked with was a struct. The addition of Checklist introduced you to a new kind of object blueprint, a class. How are classes and structs the same, and how are they different?

Both are used to create instances or objects. Both have properties, which are what the objects know, and methods, which are what the objects do.

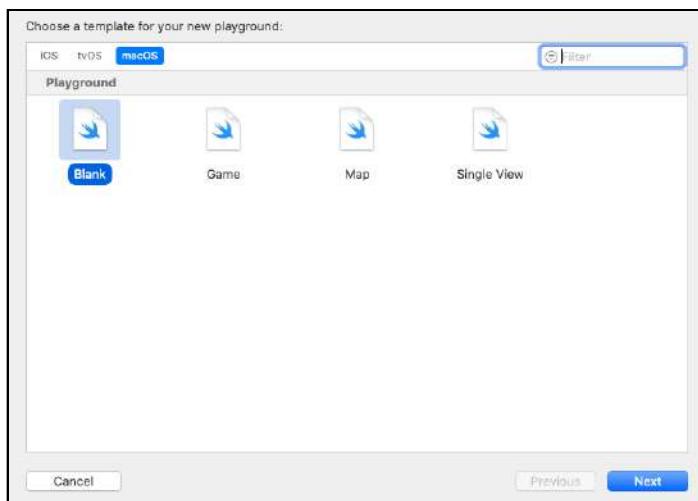
They also differ in a few ways, the most notable one being that structs are **value types** and classes are **reference types**. Rather than give you a dry technical definition of what these are, or confuse you with an analogy, your next step is to play with them using an Xcode feature called **playgrounds**.

Starting a new playground

A playground is a type of Xcode project that lets you experiment with Swift code and see the results immediately. Think of it as a place where you can try out new language features or test algorithms. Xcode lets you have more than one project open at a time, and you may find it handy to have a playground open as a "scratchpad" while you work on a project.

- In Xcode's **File** menu, select **New...**, and then **Playground**.

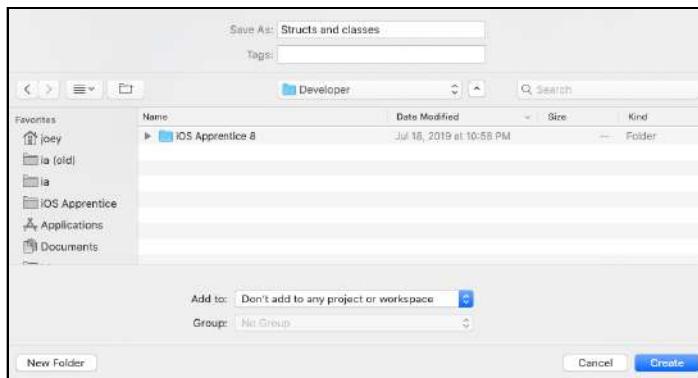
You'll see a pop-up where you select options for the playground you want to create:



Options for creating a new playground

I've found that the best kind of playground for playing with Swift language features is the blank macOS playground. That's because it doesn't load all the extra material that iOS and tvOS programming require, and it crashes less often. Here are the options to choose to create this kind of playground.

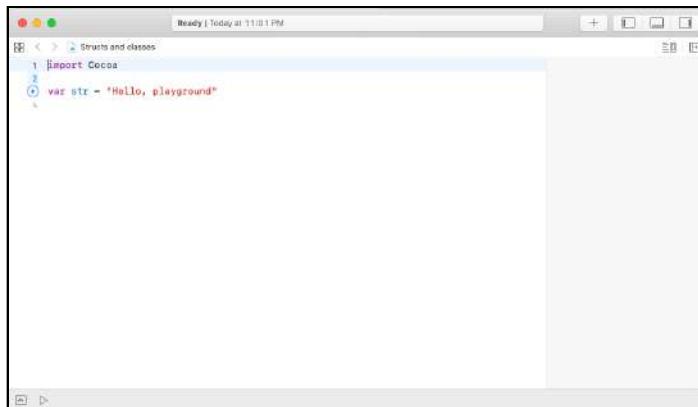
- In the pop-up, select **macOS**, highlight the **Blank** playground type, then click **Next**. You'll see a **Save As:** dialog;



Choosing a place to save the playground

- Enter a name for the playground; I used **Structs and classes**. In the **Add to:** menu, select **Don't add to any project or workspace**. Once you've done that, click the **Create** button.

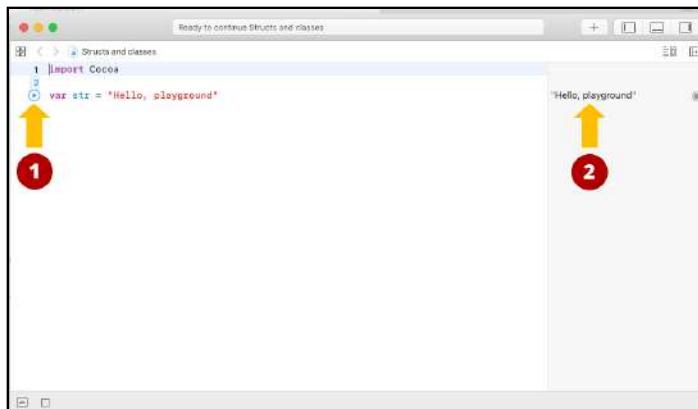
Xcode will create a new playground, which will look like this:



The newly-created Xcode playground

You can see what all the code up to and including a particular line in the playground does by moving the cursor over its line number and pressing the "Play" button that appears. The results will appear in the live view column on the right.

- Move the cursor over the number for line 3 in the playground and click the “Play” button:



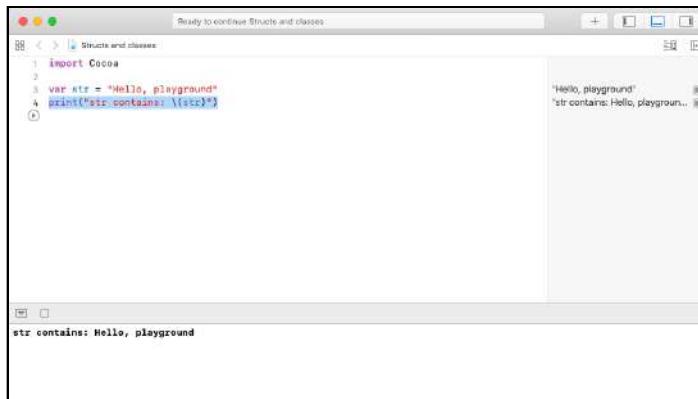
Running a line of code in the playground

You should see “Hello, playground” appear in the live view column. That’s the value that was assigned to the variable `str`.

- Add the following line to the playground:

```
print("str contains: \(str)")
```

- Move the cursor over the line number of the line you just entered and click the “Play” button. You should see this:



Printing in the playground

`print()` works in playgrounds just like it does in iOS projects: It prints to the debug console.

Now that you've covered playgrounds and their basics, it's time to use yours to learn about value types.

Value types

A “value type” is a type of data where each instance keeps its own copy. In Swift, numbers are value types. Play with a couple of numbers so you can see what this means.

- Add the following to the playground, after the code you entered previously:

```
var firstNumber = 5
var secondNumber = firstNumber
print("firstNumber contains \(firstNumber) and secondNumber
contains \(secondNumber)")
```

- Move the cursor over the line number for the last line and click the “Play” button.

The debug console should show the text: “firstNumber contains 5 and secondNumber contains 5”. This makes sense; the line `var secondNumber = firstNumber` copies the contents of `firstNumber` into `secondNumber`.

- Add the following to the playground, after the code you entered previously:

```
secondNumber = 10
print("firstNumber contains \(firstNumber) and secondNumber
contains \(secondNumber)")
```

- Move the cursor over the line number for the last line and click the “Play” button.

The debug console should show the text “firstNumber contains 5 and secondNumber contains 10”. Changing the value of `secondNumber` did not change the value of `firstNumber`. This is what “each instance keeps its own copy” means.

Structs are also **value types**. Next, you'll define a simple struct and play with it, like you just did with numbers.

- Add the following to the playground, after the code you entered previously:

```
struct PetValueType {
    var name: String = ""
    var species: String = ""
}
```



Now, create an instance of `PetValueType` and a copy of that instance.

- Add the following to the playground, after the code you entered previously:

```
var pet1 = PetValueType()  
pet1.name = "Fluffy"  
pet1.species = "cat"  
var pet2 = pet1  
print("pet1: \(pet1.name) is a \(pet1.species)")  
print("pet2: \(pet2.name) is a \(pet2.species)")
```

- Move the cursor over the line number for the last line and click the “Play” button.

The output in the debug console shows that both `pet1` and `pet2`’s `name` properties are set to “Fluffy”, and their `species` properties are both set to “cat”.

- Add the following to the playground after the code you entered previously:

```
pet2.name = "Spot"  
pet2.species = "dog"  
print("pet1: \(pet1.name) is a \(pet1.species)")  
print("pet2: \(pet2.name) is a \(pet2.species)")
```

- Move the cursor over the line number for the last line and click the “Play” button.

From the output in the debug console, you should see that `pet1`’s `name` and `species` properties are still “Fluffy” and “cat”, but `pet2`’s `name` and `species` properties are now “Spot” and “dog”. `pet1` and `pet2` are two separate values. Use value types for data where each instance is guaranteed to be its own thing and independent of any other instance. The individual checklist items in your app should be separate entities, which is why the `ChecklistItem` object blueprint is a value type — a struct.

Reference types

Classes are **reference types**. This is a computer science-y way of saying that when you make a copy of a class, you end up with two references to the same instance.

Once again, take a look at a code example. Define an object blueprint with the same properties as `PetValueType`, but as a `class` rather than a `struct`.

- Add the following to the playground, after the code you entered previously:

```
class PetReferenceType {  
    var name: String = ""  
    var species: String = ""  
}
```

Now, create an instance of `PetValueType` and a copy of that instance.

- Add the following to the playground after the code you entered previously:

```
var pet3 = PetReferenceType()  
pet3.name = "Tonkatsu"  
pet3.species = "pot-bellied pig"  
var pet4 = pet3  
print("pet3: \(pet3.name) is a \(pet3.species)")  
print("pet4: \(pet4.name) is a \(pet4.species)")
```

- Move the cursor over the line number for the last line and click the “Play” button.

In the debug console, you’ll see that both `pet3` and `pet4`’s `name` properties are set to “Tonkatsu”, and their `species` properties are set to “pot-bellied pig”.

Now, check what happens when you change the values for `pet4`.

- Add the following to the playground, after the code you entered previously:

```
pet4.name = "Sashimi"  
pet4.species = "goldfish"  
print("pet3: \(pet3.name) is a \(pet3.species)")  
print("pet4: \(pet4.name) is a \(pet4.species)")
```

Note the output in the debug console: “pet3: Sashimi is a goldfish” and ”pet4: Sashimi is a goldfish”. Both `pet3` and `pet4` are references to the same thing, which means that changing one changes the other.

Use reference types for data that different parts of an app share, or if the data needs the features that only a class offers.

The array of checklist items in your app is a shared resource that different screens will use. For this reason, the `Checklist` object blueprint is a reference type — a class.

It’s time to switch away from the playground and turn your attention to the last component of your app’s Model-View-ViewModel pattern: The view.

Connecting the view to the ViewModel

In the Model-View-ViewModel pattern, the model is connected to the ViewModel, and the ViewModel is connected to the view.



- To see the connection between the model and ViewModel, open **Checklist.swift** and look at the `checklistItems` array.

The connection between model and ViewModel is `checklistItems`, which is a property of `Checklist`, the ViewModel. Each element of `checklistItems` contains an instance of `ChecklistItem`, the model object.

I've mentioned it before, but we've been going over so much new material that it's worth repeating: The key to connecting the ViewModel to the view is in the first line of `Checklist`:

```
class Checklist: ObservableObject {
```

And in the first line of the declaration of `checklistItems`:

```
@Published var checklistItems = [
```

`Checklist` adopts the `ObservableObject` protocol, which means that an observer can constantly watch its `@Published` properties and be notified if their values change. Now, you need to set up the view, `ChecklistView`, as an observer of `Checklist`.

- Open **ChecklistView.swift**. Here's what the code for `ChecklistView` should look like:

```
struct ChecklistView: View {
    // Properties
    // =====

    // User interface content and layout
    var body: some View {
        NavigationView {
            List {
                ForEach(checklistItems) { checklistItem in
                    HStack {
                        Text(checklistItem.name)
                        Spacer()
                        Text(checklistItem.isChecked ? "✓" : "□")
                    }
                    .background(Color.white) // This makes the entire row
                    .clickable
                        .onTapGesture {
                            if let matchingIndex =
                                self.checklistItems.firstIndex(where: { $0.id == checklistItem.id }) {
                                self.checklistItems[matchingIndex].isChecked.toggle()
                            }
                        }
                }
            }
        }
    }
}
```



```
        }
        self.printChecklistContents()
    }
}
.onDelete(perform: deleteListItem)
.onMove(perform: moveListItem)
}
.navigationBarItems(trailing: EditButton())
.navigationBarTitle("Checklist")
.onAppear() {
    self.printChecklistContents()
}
}

// Methods
// =====

}
```

You've reduced ChecklistView to a single property, body, which describes the user interface. This is typical for views in SwiftUI — they contain only those things which define the user interface, and that's done entirely with properties.

Back when the entire app lived in this file, the List view that displayed the checklist items got its data from checklistItems, an array that was both a @State and a property of the original ContentView. That array still exists; it's a @Published property of Checklist, which is an ObservableObject.

You need the view to create an instance of Checklist and then observe it.

Add this line to ChecklistView, after the “Properties” comment and before the declaration for body:

```
@ObservedObject var checklist = Checklist()
```

This adds a new property to ChecklistView named checklist. The = sign means “put whatever is on the right side of me into checklist,” and Checklist() means “create a new instance of Checklist.” This is ChecklistView’s connection to Checklist — the view’s connection to the ViewModel. As an @ObservedObject, checklist will always keep the view up-to-date with any changes to its @Published properties.

Now that you’ve made the connection to Checklist, you just need to update body so that it refers to its required properties and methods in the ViewModel.



The array that `body` used to refer to, `checklistItems`, is now the `checklistItems` property of the `checklist` instance. Next, you'll use Xcode's "Find and Replace" feature to replace any occurrence of `checklistItems` in `ChecklistView` with `checklist.checklistItems`.

- In the **Find** menu, select **Find and Replace**.... You can also use the keyboard shortcut, **Command+Option+F**.

The "Find and Replace" function will appear at the top of the editor:



'Find and Replace' at the top of the editor

- Enter `checklistItems` into the **Replace** field and `checklist.checklistItems` into the **With** field, then click the **All** button.

There are also a couple of calls to `printChecklistContents()`, which was also moved to `Checklist`. Once again, "Find and Replace" will fix this.

- Enter `printChecklistContents()` into the **Replace** field and `checklist.printChecklistContents()` into the **With** field, then click the **All** button.

The last of the error messages will disappear, and the code for `ChecklistView` should look like this:

```
struct ChecklistView: View {  
  
    // Properties  
    // ======  
  
    @ObservedObject var checklist = Checklist()  
  
    // User interface content and layout  
    var body: some View {  
        NavigationView {  
            List {  
                ForEach(checklist.checklistItems) { checklistItem in  
                    HStack {  
                        Text(checklistItem.name)  
                        Spacer()  
                        Text(checklistItem.isChecked ? "✓" : "□")  
                    }  
                .background(Color.white) // This makes the entire row
```

```
clickable
    .onTapGesture {
        if let matchingIndex =
            self.checklist.checklistItems.firstIndex(where:
{ $0.id == checklistItem.id }) {
            self.checklist.checklistItems[matchingIndex].isChecked.toggle()
            self.checklist.printChecklistContents()
        }
    }
    .onDelete(perform: checklist.deleteListItem)
    .onMove(perform: checklist.moveListItem)
}
.navigationBarItems(trailing: EditButton())
.navigationBarTitle("Checklist")
.onAppear() {
    self.checklist.printChecklistContents()
}
}

// Methods
// =====

}
```

► Build and run. It works as before, but it'll be easier to maintain and upgrade now that it's been neatly divided into model, ViewModel, and view components.

Refactoring once more

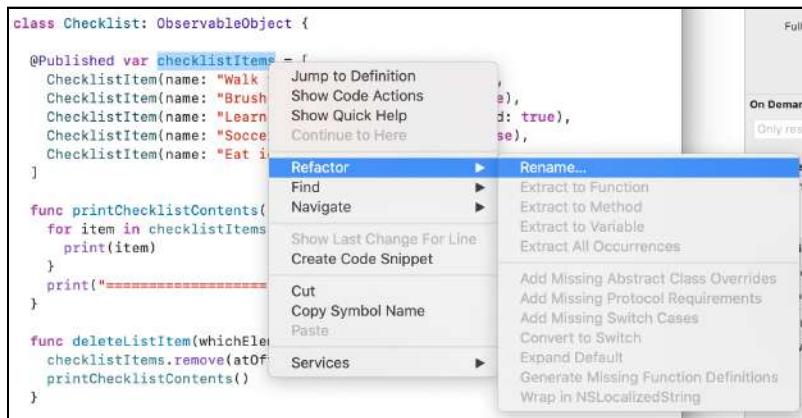
You still have one more change to the code to make...

checklistItems's name comes from the time when it was a property of the old ContentView. Now that it's a property of Checklist, the code in ChecklistView accesses it using the unnecessarily wordy checklist.checklistItems.

Let's change Checklist's checklistItems property's name to items.

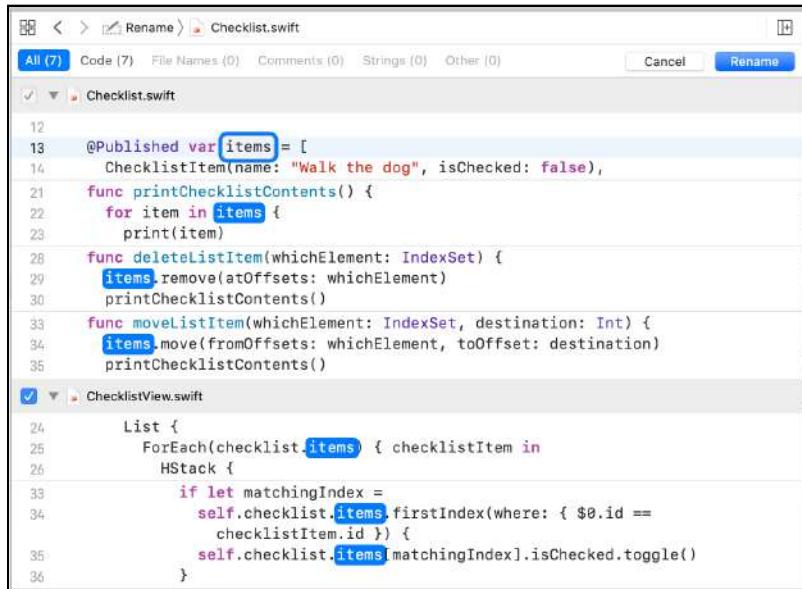


- In **Checklist.swift**, select `checklistItems`, right-click or control-click on it, select **Refactor** ► and then **Rename...**:



Select 'checklistItems' for renaming

- Type **items** and click the **Rename** button to rename the property across all the code in the project:

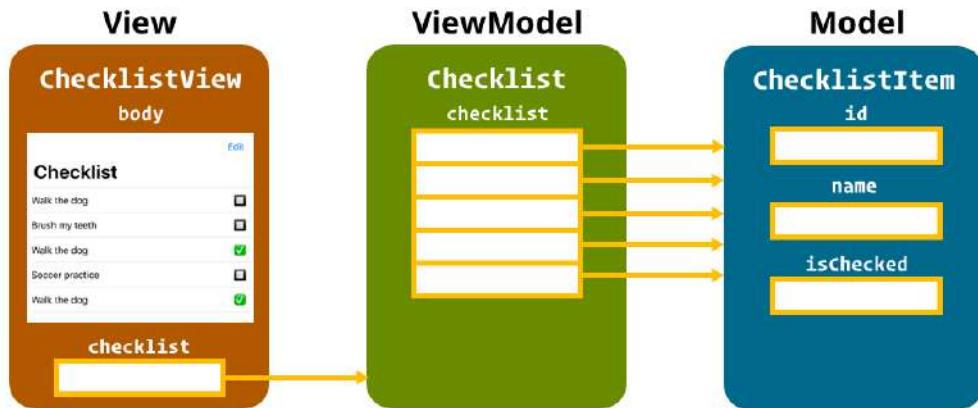


Renaming 'checklistItems' to "items"

- Build and run to confirm that this change didn't break it.

What happens when you launch an app?

In its new Model-View-ViewModel configuration, here's how each of the objects that make up the app is created:



The View, ViewModel, and Model creation order

When the app launches, the view object, `checklistView`, is created first. The view has two properties: `body`, which defines the user interface, and `checklist`, which is the connection to the `ViewModel`.

The `Checklist()` part of this line in `ChecklistView`:

```
@ObservedObject var checklist = Checklist()
```

creates an instance of `Checklist`, which brings the app's `ViewModel` into existence.

In `Checklist`, the declaration of the array you renamed to `items`:

```
@Published var items = [
    ChecklistItem(name: "Walk the dog", isChecked: false),
    ChecklistItem(name: "Brush my teeth", isChecked: false),
    ChecklistItem(name: "Learn iOS development", isChecked: true),
    ChecklistItem(name: "Soccer practice", isChecked: false),
    ChecklistItem(name: "Eat ice cream", isChecked: true),
]
```

Creates an instance of `ChecklistItem` for each instance of an item in the list.

Simply put, an instance of `ChecklistView` creates an instance of `Checklist`, which in turn creates a number of instances of `ChecklistItem`.

But what starts the process? What creates the instance of `ChecklistView`?

The app delegate and scene delegate

As you learned back in Chapter 2, “The One-Button App,” an Xcode project includes a number of source files that contain code to support the app you’re writing. This code handles all the behind-the-scenes details necessary to make a mobile app work, freeing you to focus on the code that’s specific to your app.

You may have noticed two of these files in both the *Bullseye* and *Checklist* projects: **AppDelegate.swift** and **SceneDelegate.swift**. They appear in every iOS app that uses the SwiftUI framework.

Note: When you switch to building apps with the UIKit framework later in this book, you’ll only see **AppDelegate.swift**. UIKit predates SwiftUI, and all its startup code lives in **AppDelegate.swift**. With SwiftUI, the Apple developers decided to separate that code into two separate files, where **AppDelegate.swift** contains code for managing the overall app and **SceneDelegate.swift** is the place for code that manages the app’s individual windows, or scenes.

Every program, regardless of programming language and platform, has an **entry point**. It’s the start of the program — the first set of instructions that execute when the program launches. For iOS apps, the entry point is in **app delegate** whose code is in **AppDelegate.swift**.

You can think of the app delegate as your app’s “root object.” It manages your app at the system level, which includes initializing your app’s user interface.

► Open **AppDelegate.swift**. The code inside may look incomprehensible to you right now, but you should take note of a few things. The first one is this line:

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

You should interpret this line as “this is a class named `AppDelegate`, and it’s a kind of `UIResponder` and `UIApplicationDelegate`”.



In case you’re wondering, a `UIApplicationDelegate` defines what an app delegate does and a `UIResponder` is an object that responds to user interface events such as the user tapping the screen or wiggling their phone.

Here’s `AppDelegate`’s first method:

```
func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions:  
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {  
    // Override point for customization after application launch.  
    return true  
}
```

This method’s name is `application(_:didFinishLaunchingWithOptions:)`. It performs some tasks when the app has finished launching. One of those tasks is to put your app’s user interface on the screen, which it does by creating a **scene delegate** whose code is defined in `SceneDelegate.swift`.

On Apple platforms, a **scene** is an instance of an app’s user interface. On a macOS desktop app, which can have multiple windows, each window is contained within a scene. iOS apps have only one window, and therefore have only one scene.

Each scene has a scene delegate, which manages what happens to the scene under different circumstances. Take a look at those circumstances now.

► Open `SceneDelegate.swift`. As with `AppDelegate.swift`, the details of the code might not be clear to you, but you should note a few things.

First, it defines a class named `SceneDelegate`, which is a kind of `UIResponder` and `UIWindowSceneDelegate`. `UIWindowSceneDelegate` defines what an app delegate does and, like the app delegate, also responds to user interface events.

It also has a number of methods, whose names mostly begin with different circumstances that could arise while an app is running. These include **sceneWillEnterForeground** and **sceneDidEnterBackground**. Most of these methods contain nothing but comments; they’re there for advanced programmers to add code for custom behaviors to handle different circumstances.

However, the first method in the class *does* contain code:

```
func scene(_ scene: UIScene, willConnectTo session:  
UISceneSession, options connectionOptions:  
UIScene.ConnectionOptions) {  
    // Use this method to optionally configure and attach the  
    // UIWindow `window` to the provided UIWindowScene `scene`.  
    // If using a storyboard, the `window` property will  
    // automatically be initialized and attached to the scene.
```



```
// This delegate does not imply the connecting scene or
// session are new (see
`application:configurationForConnectingSceneSession` instead).

// Create the SwiftUI view that provides the window contents.
let contentView = ContentView()

// Use a UIHostingController as window root view controller.
if let windowScene = scene as? UIWindowScene {
    let window = UIWindow(windowScene: windowScene)
    window.rootViewController = UIHostingController(rootView:
contentView)
    self.window = window
    window.makeKeyAndVisible()
}
}
```

Remember: A scene is a container for a window in your app. In iOS apps, there's only one window and it takes up the entire screen. This method determines which view is the first screen your app shows, and it does so with this code:

```
// Create the SwiftUI view that provides the window contents.
let contentView = ChecklistView()
```

Now, look at this code in detail. The first thing it does is declare a constant called `contentView`. As a constant, you can fill it with a value only once. Once filled, you can't change it to another value as long as the app is running.

The `=` specifies that you're going to fill `contentView` with a value, and that value is the thing that follows. In this case, it's `ChecklistView()`.

As a reminder, a capitalized name followed by parentheses, `()`, typically means that you're creating an object or instance. That's what's happening with `ChecklistView()`: It tells Swift to create a new instance of `ChecklistView`, which is defined inside the `ChecklistView.swift` file. This new instance is stored inside `contentView`.

Note: The convention is that names for constants, variables, methods and functions start with lowercase letters, and names for data types and object blueprints like structs and classes start with uppercase letters.

Now that `contentView` has been defined as an instance of a `ChecklistView` screen, the next bit of code puts the contents of `contentView` into the scene's window:

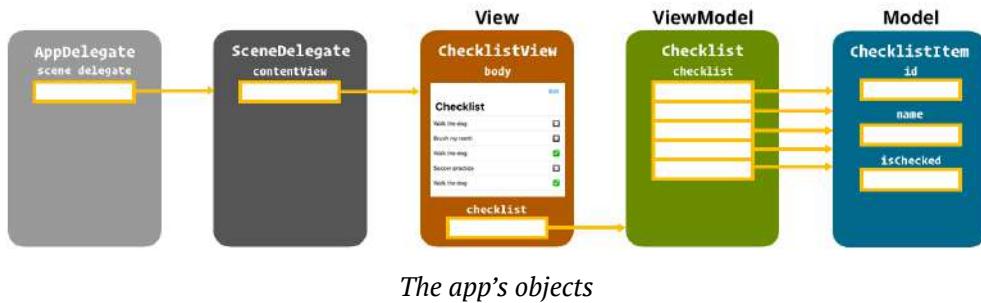
```
// Use a UIHostingController as window root view controller.  
if let windowScene = scene as? UIWindowScene {  
    let window = UIWindow(windowScene: windowScene)  
    window.rootViewController = UIHostingController(rootView:  
contentView)  
    self.window = window  
    window.makeKeyAndVisible()  
}
```

Here's the line in the code above that matters:

```
window.rootViewController = UIHostingController(rootView:  
contentView)
```

This line makes `contentView` the first screen that the app displays. Since `contentView` contains an instance of `ChecklistView`, that first screen is your list of to-do items!

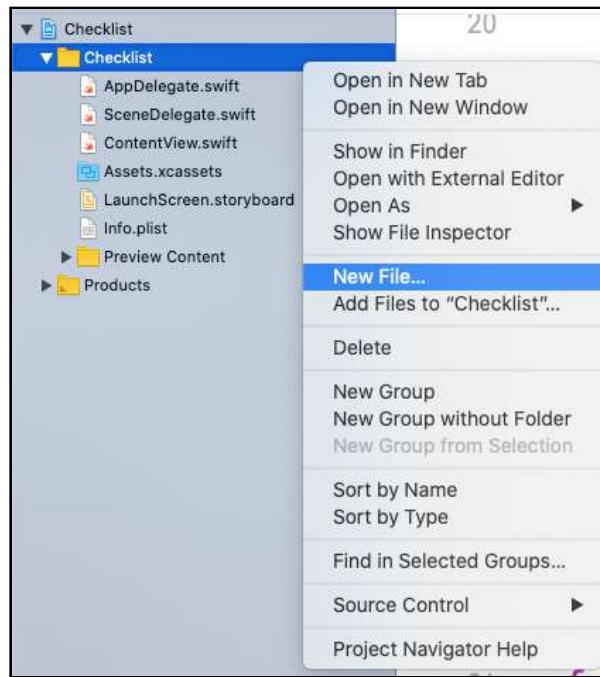
Now that you know about the app delegate and the scene delegate, here's a more complete view of the objects in the app and how they're created:



Changing the app's first screen

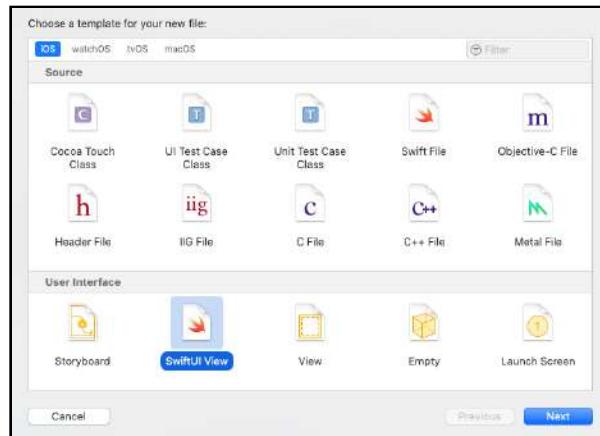
Your next step is to change the app so that it starts with a screen other than `ContentView`. The app will need a couple of additional screens anyway, so add a screen that you'll eventually use to edit items in the list.

- Add a new file by right-clicking or control-clicking on the **Checklist** folder in Xcode's Project Explorer. Select **New File...** from the menu that appears:



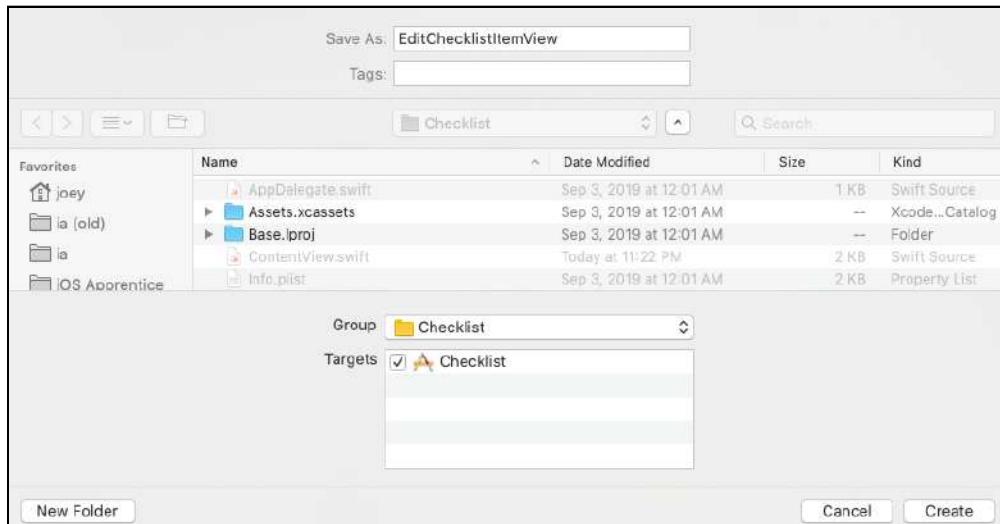
Creating a new file

- You want to add a new view to the app. So in the window that appears, make sure you've selected **iOS**, then select **SwiftUI View** and click **Next**:



Selecting SwiftUI View

- Enter the name of the new view, **EditChecklistItemView**, into the **Save As:** field. Make sure that you've selected **Checklist** in the **Group** menu and in the **Targets** menu, then click **Create**:



Save the file as 'EditChecklistItemView'

The project now has a new file, **EditChecklistItemView.swift**. If you open it, you'll see the code for a new view, **EditChecklistItemView**. Here's the interesting part of the file:

```
struct EditChecklistItemView: View {
    var body: some View {
        Text("Hello World!")
    }
}
```

You've seen this before — this is the default content for a new view created by Xcode, which is an empty screen with the text "Hello World!" in the center. Your next step is to change **SceneDelegate** so that the app opens with it, instead.

- Open **SceneDelegate.swift** and find this line in **scene(_:willConnectTo:options:)**:

```
let contentView = ChecklistView()
```

- Change the line to the following:

```
let contentView = EditChecklistItemView()
```

- Build and run. Instead of seeing the checklist, you'll see the "Hello World!" screen from `EditChecklistItemView`:



The initial `EditChecklistItem` screen

You actually want `ChecklistView` to be the app's first screen, so change the code back to what it was.

- Find this line:

```
let contentView = EditChecklistItemView()
```

- And change it back to this:

```
let contentView = ChecklistView()
```

- Build and run to confirm that `ChecklistView` is the screen that the app shows when it launches.

Congratulations! You've learned a lot about how your app works, and made some improvements to it along the way, including how best to utilise design patterns to better structure your code.

Key points

In this chapter, you did the following:

- You learned about design patterns in general, and specifically about two key patterns that you'll use as an iOS developer: Model-View-Controller (MVC) and Model-View-ViewModel (MVVM).
- You renamed the app's view from the default `ContentView` to one that better fits this project: `ChecklistView`.
- You changed the architecture of the app so that all the code no longer lives in a single file, but in a model file, a `ViewModel` file and a view file.
- You learned about the app and scene delegate objects and about what happens when an app launches.

You'll find the project files for the app at this stage under **11 - App Structure** in the Source Code folder.

In the next chapter, you'll add the next major feature: Adding and editing checklist items.



12

Chapter 12: Adding Items to the List

Joey deVilla

Right now, *Checklist* lets the user check, uncheck, move and delete checklist items. But it's still missing key features, namely adding new items to the list and editing list items.

Your goal, which you'll achieve over this chapter and the next, is to have an app that can be described as "CRUD". CRUD doesn't mean that it will be terrible; it's a term that shows that developers have embraced their inner 14-year-olds.

CRUD is shorthand for the tasks that most record-keeping apps perform. It's made up of the first letters of those tasks:

- **Create** a new record. For *Checklist*, this means creating a new checklist item. You'll add this capability to the app in this chapter.
- **Report** all records. Your app already does this by presenting the list of all items.
- **Update** an existing record. In *Checklist*, this is the ability to edit an existing checklist item. The app can't do this yet, but it will by the end of the chapter.
- **Delete** a record. The app already has this capability.



Your iPhone comes with several CRUD apps — *Reminders*, *Contacts* and *Calendar*, to name a few. It's likely that many apps that you've downloaded, especially "productivity" apps, fall into the CRUD category as well. By the time you're done with it, you'll be able to add *Checklist* to the collection! In this chapter, you'll enable the "C" in CRUD: creating a new checklist item.

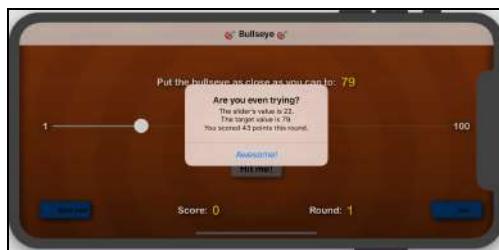
You might be surprised to learn that adding an item to the list requires just *one* line of code. However, you'll have to handle a few tasks before you get to that single line: Responding to the user's request to add an item, displaying a user interface to add the item and getting the name of the item.

Setting up the user interface

To add an item to the list, the user should be able to indicate that they want to add an item. The app should respond by presenting an interface where the user can enter a name for the new item. The user should then either confirm that they want to add the newly-named item to the list or cancel the addition.

The property that starts the process

If you think back to those long-ago days when you were coding *Bullseye*, you might remember that an alert pop-up appears when the user presses the **Hit me!** button:



The Bullseye app displays its alert pop-up

`alert(isPresented:)` made this possible. It defines an alert pop-up complete with a title, a message and a button to dismiss the alert. It also makes use of a Boolean property that determines whether the pop-up is visible.

For *Checklist*, you'll use the same technique to present the user with a pop-up where they can enter the name of the item that they want to add, then either confirm the addition or cancel it.

Now, you'll create that Boolean property for the checklist view, name it

`newChecklistItemViewIsVisible` and add it to `ChecklistView`.

- Open `ChecklistView.swift` and add the following line to `ChecklistView`'s **Properties** section, just after the `checklist` property and before the `body` property:

```
@State var newChecklistItemViewIsVisible = false
```

This property, when `true`, will cause the **Add item** pop-up to appear. You don't want the pop-up to appear until the user presses the **Add item** button, which is why its initial value is `false`. When the user wants to add an item to the list, they'll perform an action that changes the property's value to `true`. Once they've added an item, something should happen to cause the value to revert to `false`.

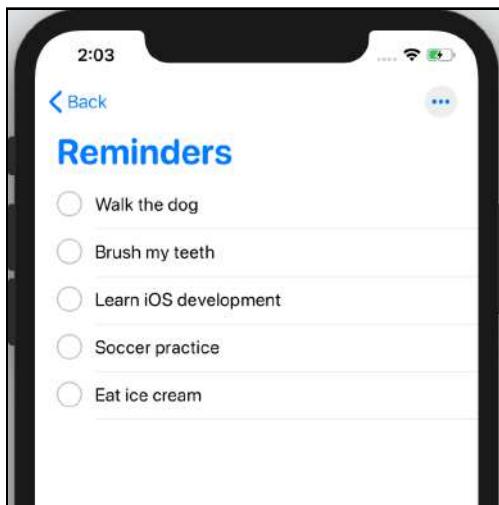
Now that we have the property, it's time to give the user a way to change it.

Adding the “Add item” button

What should the user do to create a new item in *Checklist*?

It's a good idea to look at other people's apps for inspiration, especially if those apps do something similar to the one you're writing. You'll often get good user interface ideas, learn from other developers' design mistakes and get insight into the kinds of features and functionality that users expect from an app.

Look at how users add new items to lists in iOS' built-in checklist app. Here's a list from the iOS 13 version of *Reminders*:



The Reminders app on iOS 13



To add a new item to a list in *Reminders*, the user presses the **New Reminder** button located at the bottom of the screen. You'll use a similar button in *Checklist*.

On any given list screen in *Reminders*, the navigation bar is already fully occupied with controls on either side: the **Back** button on the left and a button for options on the right. That's why the **New Reminder** button is at the bottom of the screen.

Now, look at *Checklist*'s user interface:



Where the navigation bar buttons go

Only the right-hand side of the navigation bar contains a control: the **Edit** button. The left side is available, and that's where you'll put the **Add item** button. It will follow the same format as the **New Reminder** button in *Reminders*: a “plus” sign in a circle and some text that explains the button. Our text will say: “Add item.”

Before you add a new button to the navigation bar, check to see how you added the one that's already there. You put it there with this call to one of *List*'s methods — a modifier — attached to end of the *List* in the Checklist view's body property:

```
.navigationBarItems(trailing: EditButton())
```

This line of code adds an **EditButton**, a built-in user interface element, to the *trailing* side of the navigation bar. When the device's language is set to a left-to-right language, such as English, the *right* side is the trailing side. In a right-to-left language like Hebrew, the *left* side is the trailing side.

The opposite of the trailing side is the **leading** side, which is on the left for devices set to a left-to-right language. We'll put the **Add item** button there.

► Open **ChecklistView.swift** and update the `navigationBarItems` modifier to this:

```
.navigationBarItems(  
    leading: Button(action: { self.newChecklistItemViewIsVisible =  
        true }) {  
        HStack {  
            Image(systemName: "plus.circle.fill")  
            Text("Add item")  
        }  
    },  
    trailing: EditButton()  
)
```

The modifier now has *two* parameters: `leading:`, for the button on the leading side of the navigation bar, and `trailing:`, for the button on the trailing side. The button on the trailing side remains the same.

This code defines the **Add item** button on the leading side:

```
Button(action: { self.newChecklistItemViewIsVisible = true }) {  
    HStack {  
        Image(systemName: "plus.circle.fill")  
        Text("Add item")  
    }  
}
```

This code creates a button based on two parameters:

- The `action:` parameter, which contains code that defines what should happen when a user presses the button. This code sets the value contained in the `newChecklistItemViewIsVisible` property to `true`.
- The parameter within the { and } characters, which contains `View` objects that define what the button looks like. These objects could be `Text`, `Image` or any other object that's a kind of `View`.

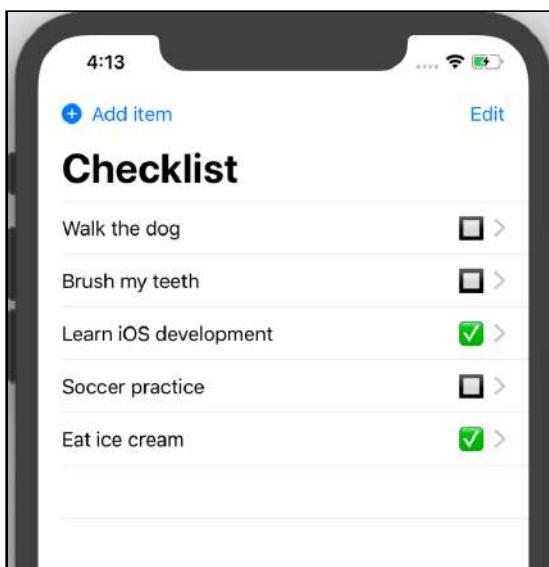
In this case, you'll use an `HStack` to create a button appearance that's a combination of an `Image` view followed by `Text`, just like the **New Reminder** button in *Reminders*.

You probably noticed that you used a slightly different method, `Image(systemName:)`, to create the icon for the **Add item** button. This method makes an image based on **SF Symbols**, a pre-defined set of over 1,500 symbols that you can use in any app running on iOS 13 or later. `Image(systemName:)` lets you call up any of the symbols' images by name. The symbol named "plus.circle.fill" is a + sign drawn in a filled circle.

Note: You can browse the complete set of SF Symbols in the [SF Symbols desktop app](#), which is available at Apple's Developer site.

It's time to see the button in action!

- Run the app. You should now see the **Add item** button on the left, or leading, side of the navigation bar. You can try pressing it, but nothing will happen... yet!



The app, now featuring the 'Add item' button

Displaying a pop-up

In *Bullseye*, when the user presses the **Hit me!** button, this action activates the code in the button's action: parameter, which sets the `alertVisible` property to `true`. `alert(isPresented:)` is also attached to the button, and connects to the `alertVisible` property. The modifier displays an alert pop-up if its `isPresented` parameter — `alertVisible` — is `true`.

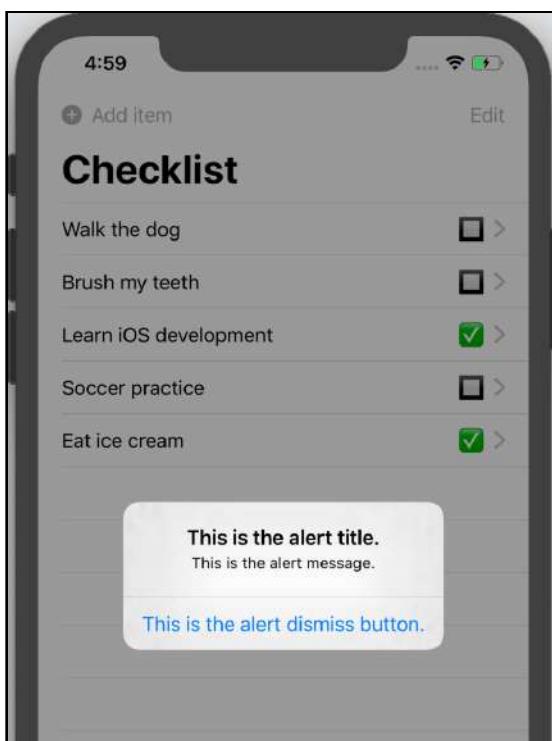
Here's the code for the modifier:

```
.alert(isPresented: self.$alertVisible) {
    Alert(title: Text(alertTitle()),
        message: Text(scoringMessage()),
        dismissButton: .default(Text("Awesome!")) {
            self.startNewRound()
        }
    )
}
```

```
    }  
}
```

You're going to do something similar with the **Add item** button. You've already done some of the work: You've added a Boolean property that will control the appearance of a pop-up screen, and you've added a button that controls the value in the Boolean property. The next step is to create a pop-up window that the Boolean property controls.

An Alert pop-up would be a little too small:



The app, displaying an 'Alert' pop-up

Whatever kind of pop-up you use should provide lots of space — not just enough space for the user to enter a name for the new checklist item, but enough space for additional information that you might decide to include with an item in later editions of the app.

That kind of pop-up is called a sheet. It's much larger than an alert; in fact, it takes up nearly the entire screen. Here's an example of a sheet in action:



The app, displaying an 'Sheet' pop-up

Next, you'll write the code to display a basic sheet with the message, "New item screen coming soon!" when the `newChecklistItemViewIsVisible` property is true, which happens when the user presses the **Add item** button.

► Add the following code after `NavigationView`'s closing brace:

```
.sheet(isPresented: $newChecklistItemViewIsVisible) {  
    Text("New item screen coming soon!")  
}
```

The `sheet(isPresented:)` method takes two arguments:

- `isPresented`: displays the sheet if its value is `true`.
- The second argument, which you'll find within the brackets, is a view defining the content of the sheet. The code above sets that content to a `Text` view displaying the text "New item screen coming soon!".

- Run the app and press the **Add item** button. You'll see this:



The app, displaying a sheet that says 'New item screen coming soon!'

- Swipe down on the sheet. This will dismiss it, returning you to the checklist view.

Defining the sheet

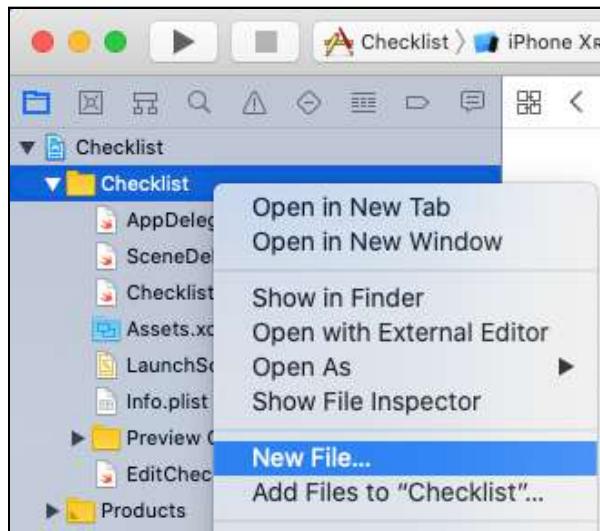
Checklist is an app, not a web site from the 1990s, so we can't leave it in a state where it shows a blank screen promising an upcoming feature. Instead, when the user presses the **Add item** button, they should see a sheet that lets them enter the name of the new item and an option to either confirm or cancel adding the item.

You'll set up the sheet so that it displays the following:

- The title “Add new item”.
- A text field where the user can enter the name of the new item.
- A button that the user can press to confirm that they want to add the new item to the list.
- A text prompt that tells the user to swipe down to cancel adding an item to the list.

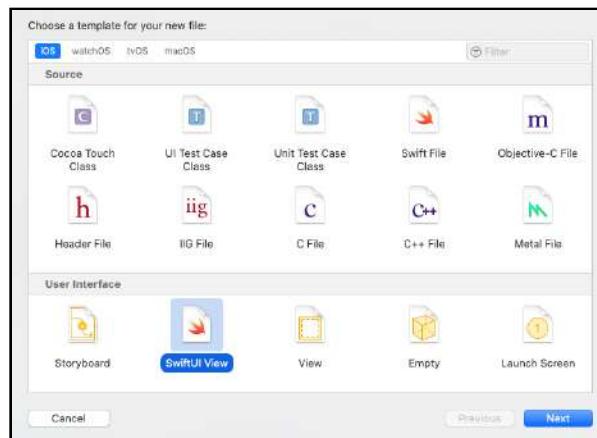
You'll define this screen in its own view, `NewChecklistItemView`, which will live in its own file, `NewChecklistItemView.swift`.

- Add a new file to the project by right-clicking or control-clicking on the **Checklist** folder in Xcode's Project Explorer. Select **New File...** from the menu that appears:



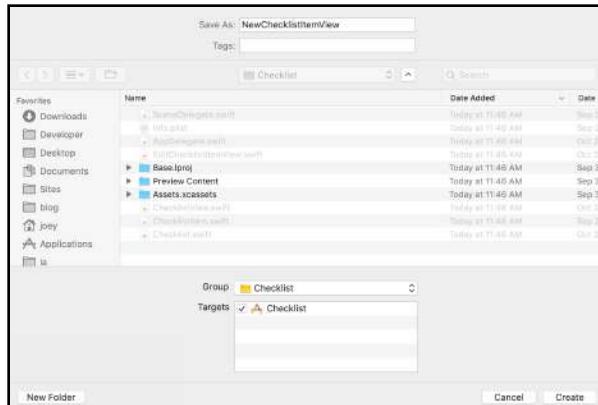
Add a new file to the project

- In the window that appears, make sure that you've selected **iOS** then select **SwiftUI View**. Click **Next**:



Select the 'SwiftUI View' template

- Enter **NewChecklistItemView** into the **Save As:** field. Make sure that you've selected **ChecklistItem** in the **Group** menu and in the **Targets** menu, then click **Create**:



Name the file 'NewChecklistItemView'

The project now has a new file named **NewChecklistItemView.swift**.

- Open **NewChecklistItemView.swift**. Its body defines the default “Hello World!” view:

```
var body: some View {
    Text("Hello World!")
}
```

- Update **NewChecklistItemView**'s body to the following:

```
var body: some View {
    VStack {
        Text("Add new item")
        Text("Enter item name")
        Button(action: {
            }) {
            HStack {
                Image(systemName: "plus.circle.fill")
                Text("Add new item")
            }
        }
        Text("Swipe down to cancel.")
    }
}
```

To see your new screen, you need to change the line of code that defines the content of the sheet that appears when the user presses the **Add item** button on the checklist screen.

- Open **ChecklistView.swift** and change the code that opens the sheet to the following:

```
.sheet(isPresented: $newChecklistItemViewIsVisible) {  
    NewChecklistItemView()  
}
```

- Run the app and press the **Add item** button. You'll see this:



The Add new item' sheet

That's not quite the look you're going for. Next, you'll try to fix it.

Fixing the sheet's layout

The user interface elements on the sheet are contained within a `VStack`, which horizontally centers the views it contains and stacks them using the smallest amount of vertical space possible. It then vertically centers itself. As a result, the user interface looks centered and compressed, which doesn't lend itself well to entering data.

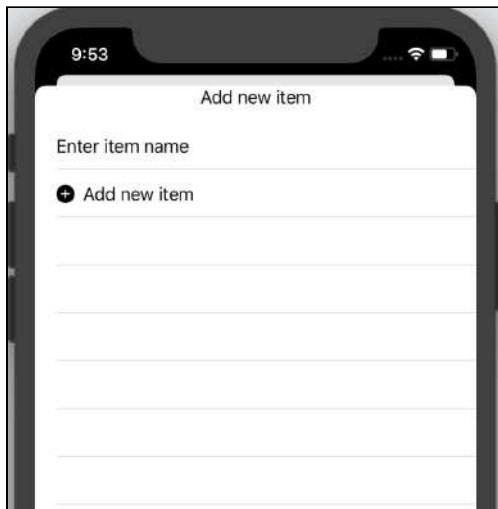
You could use `Spacer` views to fix this user interface, just as you did with *Bullseye*, but it's worth looking at a couple of other approaches.

The `List` view organizes the views it contains into a vertical stack, just as a `VStack` does. Let's put the **Enter item name** text and **Add new item** button into a `List` and see what happens.

- Update the body property to the following:

```
var body: some View {
    VStack {
        Text("Add new item")
        List {
            Text("Enter item name")
            Button(action: {
            }) {
                HStack {
                    Image(systemName: "plus.circle.fill")
                    Text("Add new item")
                }
            }
        }
        Text("Swipe down to cancel.")
    }
}
```

- Run the app and press **Add item**. You'll see this:



The 'Add new item' sheet, using a List

A **List** aligns the views it contains to the leading side, which is the left side for left-to-right languages such as English. Unlike the **VStack**, which takes only the vertical space it needs, the **List** expands to take as much vertical space as possible, filling it with empty cells.

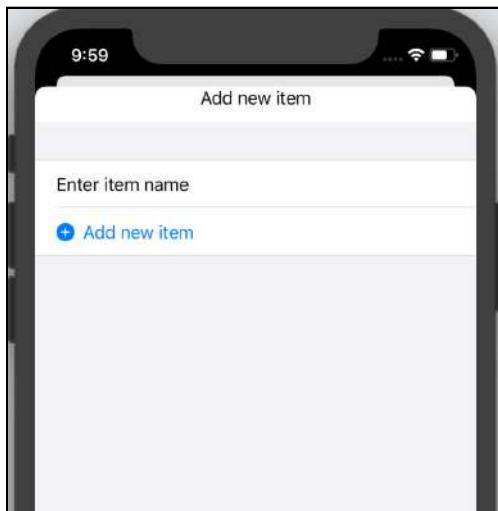
A **List** is a better container for the **Enter item name** text and **Add new item** button than a **VStack**, but it's still not quite right. The empty cells at the bottom of the list suggest that the screen might present more information, but that's not the case.

It's time to introduce a new SwiftUI view: the `Form`. The official documentation describes it as, "A container for grouping controls used for data entry, such as in settings or inspectors." Rather than comment on this description, change your `List` view into a `Form` and see what happens.

- Update body to the following:

```
var body: some View {
    VStack {
        Text("Add new item")
        Form {
            Text("Enter item name")
            Button(action: {
            }) {
                HStack {
                    Image(systemName: "plus.circle.fill")
                    Text("Add new item")
                }
            }
        }
        Text("Swipe down to cancel.")
    }
}
```

- Run the app and press **Add item**. You'll see this:

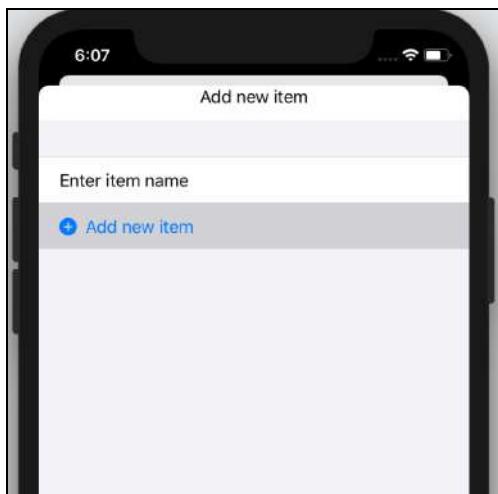


The 'Add new item' sheet, using a Form

`Form` was designed with data entry in mind. The way it separates itself from the views above and below it is a visual hint to the user that says, "You're going to enter information here." It aligns the views appropriately based on the user's language

settings, making it easier for them to enter information. It also provides clear divisions between the views it contains, clearly showing the user each piece of information they must provide, and subtly giving them an idea of how much information they're expected to enter.

Some views, when put into a **Form**, adapt themselves for data entry. For example, the **Button** view expands its tappable area to take up the full width of the form:



Buttons expand their tappable areas when inside a Form

For these reasons, you'll use **Form** as the view to contain the **Enter item name** text and the **Add new item** button.

Collecting user input

Now that you've settled on the layout of the **Add new item** sheet, you can make it functional.

- Run the app and press **Add item**. Try tapping on **Enter item name** to enter the name of an item to add to the list.

Nothing happens, because you tapped on a **Text** view. It's a view that only *outputs* data. In order to collect data from the user, you'll need to use a different view.

You've already used a view that allows the user to input data: the **Slider** view in *Bullseye*. You also created a property to store the slider's current position and set up a two-way binding between the slider and the property. With the binding in effect, the property updated whenever the user moved the slider, and any changes made to the property updated the slider's position.

You'll do something similar to get the name of the item to add to the list. You'll set a `TextField` view where the user can enter the name and a property to store the name. You'll also bind the two together so that changes to the property will change the content of the text field and changes to the text field will change the content of the property.

Now, go ahead and create the property to store the name.

► Add the following to `NewChecklistItemView`, before body:

```
@State var newItemName = ""
```

This creates a new property, `newItemName`, which is initially set to an empty string — that's what the two double-quote characters ("") with nothing between them means.

Now, give the user a `TextField` where they can enter the new item name and connect it to `newItemName`.

► Find the following line in the body property:

```
Text("Enter item name")
```

And change it to this:

```
TextField("Enter new item name here", text: $newItemName)
```

This line initializes a new `TextField`. It takes two arguments:

- Some **hint text**: Light-colored text that tells the user what the text field is for or what information to enter into it. This tutorial uses the text, “Enter new item name here” for the hint text, but feel free to customize it however you like.
- A **binding**: A two-way connection to a property. This tutorial sets this value to `$newItemName`, which connects the text field to `newItemName`. Remember: `newItemName` refers to the value stored in the property, and `$newItemName` is a two-way connection to that value.

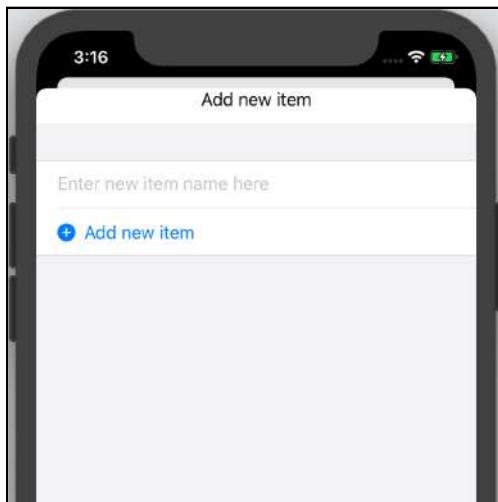
As the user changes text inside the text field, the value stored in `newItemName` will change to match. Conversely, if code changes the values stored in `newItemName`, the contents inside the text field will change to match.

With the changes you just made, the code for `NewChecklistItemView` should now look like this:

```
struct NewChecklistItemView: View {  
    @State var newItemName = ""  
  
    var body: some View {  
        VStack {  
            Text("Add new item")  
            Form {  
                TextField("Enter new item name here", text:  
$newItemName)  
                Button(action: {  
}) {  
                    HStack {  
                        Image(systemName: "plus.circle.fill")  
                        Text("Add new item")  
                    }  
                }  
                Text("Swipe down to cancel.")  
            }  
        }  
    }  
}
```

Now it's time to see these changes in action!

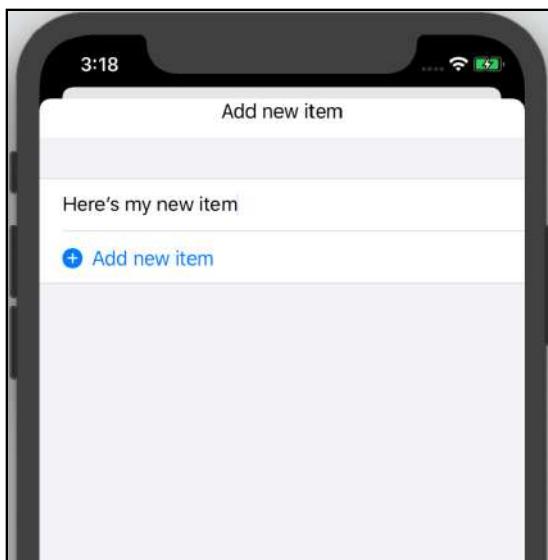
- Run the app and press **Add item**. You should see this:



The 'Add new item' screen with a text field

As promised, the `TextField` displays the hint text. Now, you can try entering some text.

- Tap the text field and try typing in it. You can now type a name for a new checklist item:



Entering text into the text field

If you also pressed the **Add new item** button, you saw that nothing happened. You'll take care of that next.

Adding a new item to the list

When the user presses the **Add new item** button, the following should happen:

1. The app should create a new checklist item. Its name should be whatever the user typed into the text field, and `isChecked` should be the default value, `false`.
2. The newly-created checklist item should appear in the list.
3. The **Add new item** sheet should disappear, returning the user to the checklist view.

Before you write the code that performs these tasks, check out where it will go.

Take a look at the code in body that defines the **Add new item** button:

```
Button(action: {  
}) {  
    HStack {  
        Image(systemName: "plus.circle.fill")  
        Text("Add new item")  
    }  
}
```

There's code that defines what the button *looks like*:

```
HStack {  
    Image(systemName: "plus.circle.fill")  
    Text("Add new item")  
}
```

And there's a place where you'll add code to define what the button *does* when pressed — the Button's `action:` parameter:

```
Button(action: {  
})
```

Start by adding the code to perform the first task: creating the new checklist item.

Creating a new checklist item

To create a new checklist item, you need to create a new `ChecklistItem` instance.

You may not realize it, but you've been creating new instances for some time now. You've been doing it by using the **initializer** of the `struct` or `class` that you wanted to instantiate, which is a special method that creates a new instance of a `struct` or `class` and can also set values for that instance's properties.

When you define a new `struct` or `class`, Swift automatically defines at least one initializer for it. You can also define your own additional initializers to set up instances in very specific ways.

An initializer takes its name from `struct` or `class` that it initializes. Whenever you see a capitalized name followed by parentheses (the `(` and `)` characters), you're probably looking at an initializer. If you look at the body of any view you've worked on so far, you'll see that it's full of calls to initializers: things like `Text` and `Button`.

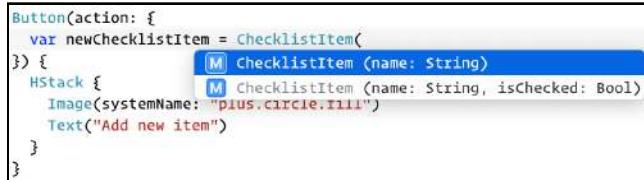
Now, you'll use `ChecklistItem`'s initializer to create a new checklist item.



- Add a line to the Button's action: parameter so that the part of body that defines the button looks like this:

```
Button(action: {
    var newChecklistItem = ChecklistItem(name: self.newItemName)
}) {
    HStack {
        Image(systemName: "plus.circle.fill")
        Text("Add new item")
    }
}
```

As you typed in the new line to create a new ChecklistItem instance, Xcode suggested a couple of initializers:



Xcode suggests initializers for ChecklistItem

That's because ChecklistItem has not one, but *two* initializers:

- **ChecklistItem(name:)**: Creates a new ChecklistItem instance, given only a name for the new item.
- **ChecklistItem(name:isChecked:)**: Also creates a new ChecklistItem instance, but requires that you specify both a name for the new item and whether it's checked or not.

ChecklistItem has two initializers because its isChecked property has a default value of false. Swift detected this default value and automatically created two initializers: One where you don't have to provide a value for isChecked, and one where you do. Since you're treating all new checklist items as unchecked, you'll use the initializer that doesn't require a value for isChecked.

The new line declares a new variable named newChecklistItem and assigns a new ChecklistItem instance to it. The item's name is set to the contents of NewChecklistView's newItemName. newItemName is bound to the text field, so its contents are the text field's contents. With this single line, you've taken care of the first step of adding a new item to the list.

Now that you have a new checklist item, it's time to add it to the list.

Getting access to the checklist

You can't add the item to the list without accessing the list. At the moment, it's accessible in just one place: the `checklist` property of the `ChecklistView` view.

To work with the list from within `NewChecklistItemView`, you'll need to do a couple of things:

- Set up a property within `NewChecklistItemView` that will hold the list.
- Have `ChecklistView` give the list to `NewChecklistItemView`, which will store the list inside the property mentioned above.

Now, go ahead and set up the property.

► Add the following line of code just before the line where you declare `newItemName`.

```
var checklist: Checklist
```

This new line of code is pretty straightforward. It declares a property named `checklist` that can hold `Checklist` instances.

Moments after you add this new line of code, Xcode will show an error message in the code that generates the preview:



```
struct NewChecklistItemView_Previews: PreviewProvider {
    static var previews: some View {
        NewChecklistItemView() // Missing argument for parameter 'checklist' in call
    }
}
```

A new error message appears

Adding the `checklist` property to `NewChecklistItemView` changed its initializer. It now has a parameter, `checklist`:, which you'll use to pass the checklist from `ChecklistView` to `NewChecklistItemView`. The preview is making a call to the old initializer, `NewChecklistItemView()`, which no longer exists.

To fix the problem, you'll change the call so that it passes the value of `checklist` to the preview.

► Change the preview code to the following:

```
struct NewChecklistItemView_Previews: PreviewProvider {
    static var previews: some View {
        NewChecklistItemView(checklist: Checklist())
    }
}
```

The code on the previous page tells the preview that its content is an instance of `NewChecklistItemView` and that it should use a new instance of `Checklist` for its checklist. Xcode then uses this information to generate the preview. [I changed this sentence from passive to active voice. Is Xcode the correct actor.]

Now that you've fixed the error in the preview, you can return to the task of passing the checklist from `ChecklistView` to `NewChecklistItemView`.

- Open `ChecklistView.swift` and change the code in `ChecklistView`'s body that displays the sheet to the following:

```
.sheet(isPresented: $newChecklistItemViewIsVisible) {
    NewChecklistItemView(checklist: self.checklist)
}
```

The complete body should now look like this:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklist.items) { checklistItem in
                HStack {
                    Text(checklistItem.name)
                    Spacer()
                    Text(checklistItem.isChecked ? "✓" : "□")
                }
                .background(Color.white) // This makes the entire row
                .clickable
                .onTapGesture {
                    if let matchingIndex =
                        self.checklist.items.firstIndex(where: { $0.id ==
                            checklistItem.id }) {
                        self.checklist.items[matchingIndex].isChecked.toggle()
                    }
                    self.checklist.printChecklistContents()
                }
                .onDelete(perform: checklist.deleteListItem)
                .onMove(perform: checklist.moveListItem)
            }
            .navigationBarItems(
                leading: Button(action:
                { self.newChecklistItemViewIsVisible = true }) {
                    HStack {
                        Image(systemName: "plus.circle.fill")
                        Text("Add item")
                    }
                },
                trailing: EditButton()
            )
        }
    }
}
```

```
        )
    .navigationBarTitle("Checklist")
    .onAppear() {
        self.checklist.printChecklistContents()
    }
}
.sheet(isPresented: $newChecklistItemViewIsVisible) {
    NewChecklistItemView(checklist: self.checklist)
}
```

- Run the app and press the **Add item** button. It appears to work as before, but under the hood, `NewChecklistItemView` has information that it didn't have before: It now has access to the checklist data.

A quick reminder: Checklist is a class

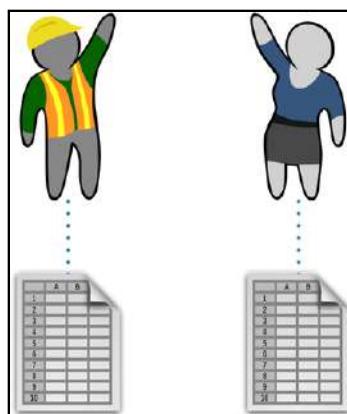
It's time for a quick reminder about `Checklist`, which uses a slightly different kind of object blueprint.

- Open `Checklist.swift`. Near the top of the file you'll see this line:

```
class Checklist: ObservableObject {
```

Unlike most of the other object blueprints in the project, `Checklist` is a **class** and not a **struct**. I covered this in the previous chapter, but rather than make you look back, I'll provide a quick refresher.

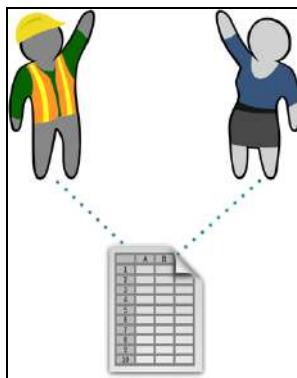
A **struct** is a **value type**, where each instance keeps its own copy. Think of value types as being like a spreadsheet file that you email to your co-workers:



A value type is like a spreadsheet file

With this arrangement, each co-worker has their own copy of the spreadsheet. Any change that a co-worker makes to their spreadsheet will appear only in that co-worker's spreadsheet and nowhere else.

On the other hand, a class is a **reference type**, where all instances refer to the same copy. Think of reference types as being like a Google Docs spreadsheet that you share with your co-workers:



A reference type is like a Google doc

In this scenario, every co-worker is working on the same copy of the doc. Any change that a co-worker makes to the doc will be seen by every other co-worker, since they're all viewing the same thing.

By declaring Checklist as a `class`, you made it possible for different views to access the same checklist. When ChecklistView passes the checklist to NewChecklistItemView, it's giving NewChecklistItemView access to the same checklist that it uses. That's what makes it possible for NewChecklistItemView to add an item to the checklist.

Adding the new item to the list

Now that you've gone through all that setup, plus a quick review of value and reference types, it's time to add the newly-created checklist item to the list!

► In `NewChecklistItemView.swift`, add a couple of lines to the Button's `action:` parameter so that the part of body that defines the button looks like this:

```
Button(action: {  
    var newItem = ChecklistItem(name: self.newItemName)  
    self.checklist.items.append(newItem)  
    self.checklist.printChecklistContents()  
}) {
```

```
HStack {  
    Image(systemName: "plus.circle.fill")  
    Text("Add new item")  
}
```

As I mentioned near the start of the chapter, it takes a single line of code to add a new item to the checklist. It's this line:

```
self.checklist.items.append(newChecklistItem)
```

Remember that `items` is a property of the checklist and that it's an array that holds all its items. `append()` is an array method that takes a given object, adds an element to the array and puts the object in the new element. `append(newChecklistItem)` adds `newChecklistItem` to the checklist.

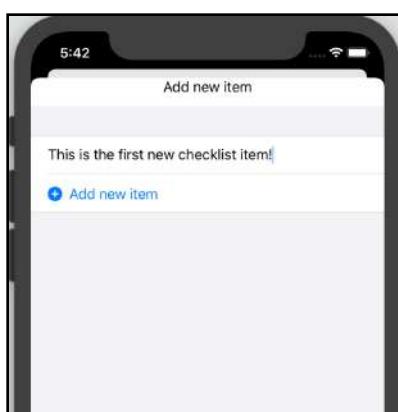
To confirm that the app is really adding newly-created objects to the checklist, you included this line of code:

```
self.checklist.printChecklistContents()
```

This uses the `printChecklistContents()` method in `Checklist` to display what's in the checklist.

Now, try adding items to the list!

- Run the app. Press the **Add item** button, which will display the **Add new item** sheet.
- Type something into the text field and press the **Add new item** button:



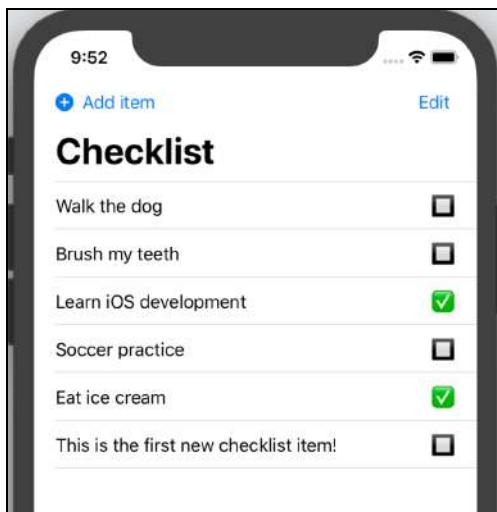
Entering the first new checklist item

Look at Xcode's debug console; you should see that the item has been added to the list:

```
ChecklistItem(id: F7F60F48-13C9-405C-9FE1-84752849DB58,
    name: "Walk the dog", isChecked: false)
ChecklistItem(id: C58F4509-A903-40A1-892F-56B1F5242639,
    name: "Brush my teeth", isChecked: false)
ChecklistItem(id: 74ADC696-23BC-4C91-A143-F13AF26F1143,
    name: "Learn iOS development", isChecked: true)
ChecklistItem(id: 3132033D-6FE6-4926-9538-6AB581716AA1,
    name: "Soccer practice", isChecked: false)
ChecklistItem(id: 0E57C488-0CB1-4417-B806-7D4B9393810E,
    name: "Eat ice cream", isChecked: true)
ChecklistItem(id: 32994B78-024B-4A1F-8355-AD6B6495FD16,
    name: "This is the first new checklist item!", isChecked:
false)
=====
=====
```

The new checklist item in the Xcode console

- Swipe down on the **Add new item** sheet. It will disappear, revealing the checklist screen, which will contain the newly-added item:



The checklist with the newly-added checklist item

It works! The user can now add items to the checklist, bringing you one step closer to having a fully-CRUD app. (Don't forget, in this case, CRUD is a good thing. :])

Dismissing the Add new item sheet automatically

The user should only swipe down on the **Add new item** sheet to cancel adding an item. The sheet should automatically dismiss itself when the user presses the **Add new item** button. With just two lines of code, you can make this happen.

The first line of code will create a property that allows you to control the way the current view presents itself.

- Add the following just below the line where you declared the `newItemName` property:

```
@Environment(\.presentationMode) var presentationMode
```

While you may not know exactly what this line of code does, you've probably seen enough Swift syntax to know that it creates a variable property named `presentationMode`, which is a kind of `@Environment(\.presentationMode)`, whatever that is.

In Swift, anything that begins with the @ character is a hint to the operating system that it's something special that it needs to pay particular attention to.

You've already seen something that begins with the @ character: `@State`. In *Checklist*, you used it in both `ChecklistView` and `NewChecklistItemView` to mark certain properties as state properties. In `ChecklistView`, you marked the `checklist` property as a `@State` property so that changes to `checklist` would be immediately and automatically reflected in the `List` view. In `NewChecklistItemView`, you marked the `newItemName` property so that its contents would be reflected in the "Add new item" text field and vice versa.

`@Environment` marks a property as one that can access a specific system setting related to the operating system *environment*, hence the name. It lets you find useful settings, such as whether the user's language is left-to-right or right-to-left, which calendar system is appropriate for the user's locale settings, the current color scheme and other system-wide settings.

The content of the parentheses (the (and) characters) that follow `@Environment` specifies the kind of setting to access. In this case, you're accessing a setting called `presentationMode`. It's an object that has a method that dismisses the current view.

Now that you have the `presentationMode` property, you can call on one of the methods nested deep within it to dismiss the sheet when the user presses the **Add new item** button.

- In `NewChecklistItemView.swift`, add a new line of code to the Button's `action:` parameter so that the part of the body property that defines the button looks like this:

```
Button(action: {
    var newChecklistItem = ChecklistItem(name: self.newItemName)
    self.checklist.items.append(newChecklistItem)
    self.checklist.printChecklistContents()
    self.presentationMode.wrappedValue.dismiss()
}) {
    HStack {
        Image(systemName: "plus.circle.fill")
        Text("Add new item")
    }
}
```

- Run the app. Press the **Add item** button, which will display the **Add new item** sheet. Enter a name for a new item and press the **Add new item** button.

The sheet will dismiss itself and you'll return to the checklist, which is the way the app should work.

Dealing with a SwiftUI bug

The perils of new platforms

Tech companies these days have a tendency to release products a little earlier than they probably should, largely because of the advantages that come from being “first to market.” Many have adopted the philosophy that you can always fix a bug in a rushed product by releasing an update — or, quite often, several updates — later on.

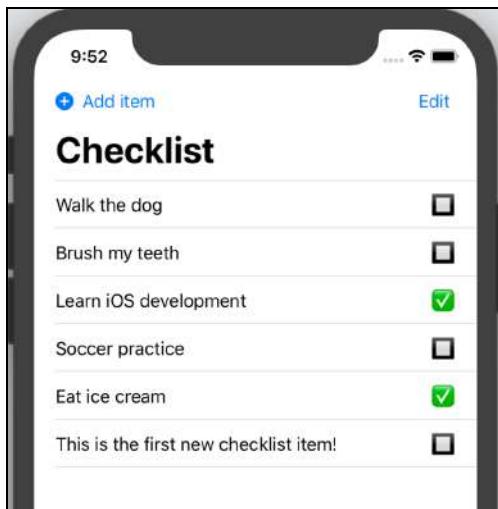
This “release early, release often” approach is doubly true for developer tools and platforms. Unlike consumer products, which usually have a small set of use cases, developers use their tools in many different ways, and there’s no way to predict how they’ll use any given feature. The vendors who make developer tools often find it more practical to treat their users as “gamma testers” — in the Greek alphabet, gamma is the next letter after beta — and rely on their feedback to find out where the bugs are.



SwiftUI is a brand-new platform, and as one of the earliest developers to use it, you should expect to encounter some bugs. If you've been playing with the app, you may have already encountered the one that you're about to deal with next.

Fixing the navigation bar button bug

- Run the app. Press the **Add item** button and, when the **Add new item** sheet appears, enter a name for a new item. Press the **Add new item** button, which will return you to the checklist. So far, so good:



The checklist with a newly-added checklist item

The bug becomes apparent when you try to add another item.

- On the checklist screen, press the **Add item** button again. This time, it doesn't respond to your touch, nor does it take you to the **Add new item** sheet. If you press the **Edit** button, you'll find that it also doesn't work.

I struggled with this bug for a few minutes and discovered that checking or unchecking any item in the list un-sticks the buttons in the navigation bar:

- Tap on any item in the checklist to check or un-check it. Then try pressing the **Add item** or **Edit** button. You'll see that they work now.

I looked at the code that defined the buttons in the navigation bar and the code that dismissed the **Add new item** sheet after the user added a new item and couldn't see anything that would cause this strange behavior.

With the help of Adam Rush, the technical editor for this book, I found a strange workaround. For some reason, changing the style of the title in the navigation bar fixes the bug:

- Open **ChecklistView.swift**. In the body of ChecklistView, change the line that defines the navigation bar title from this:

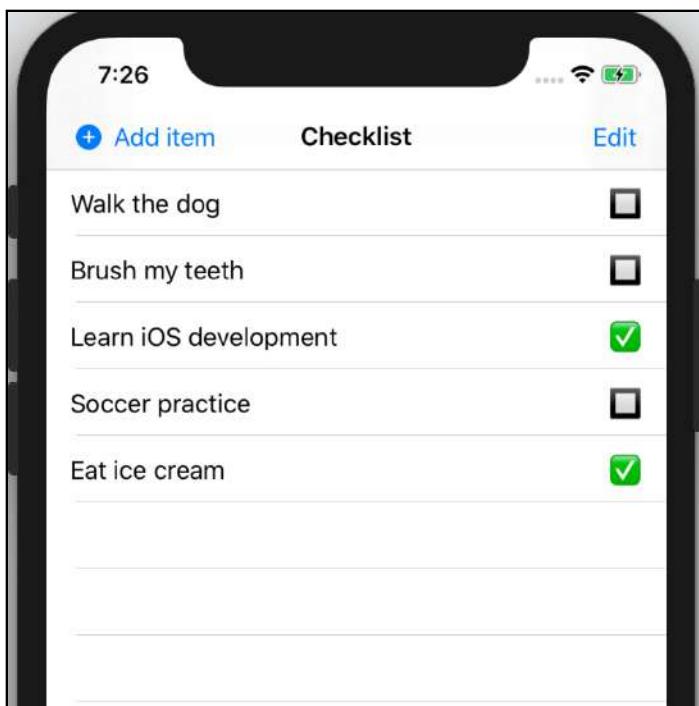
```
.navigationBarTitle("Checklist")
```

To this:

```
.navigationBarTitle("Checklist", displayMode: .inline)
```

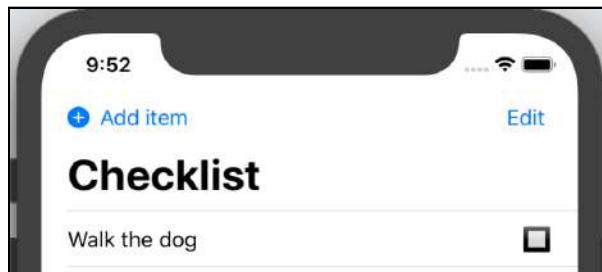
Want to see what this change does? Run the app and find out.

- Run the app. The checklist screen will look like this:



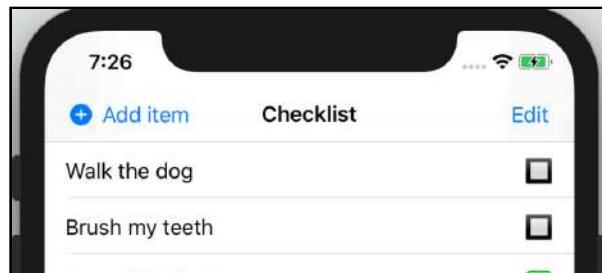
The checklist with an inline navigation bar title

This alteration to the code changes the style of the navigation bar title from the default style...



Navigation bar with the default style title

...to the **inline** style, where the navigation bar's title is on the same line as its buttons:



Navigation bar with the inline style title

While I prefer the default style, I prefer a properly-working app even more.

Apple will eventually address this bug with an iOS update, and when that happens, this fix will no longer be necessary.

How to deal with SwiftUI bugs on your own

Just as developer tool vendors often rely on the developer community to find bugs, you can also rely on the developer community to help you find ways around them. Programming has a strong tradition of sharing information, and you'll find that iOS programmers are generally an energetic and friendly bunch. They're quick to discover bugs in iOS, figure out solutions or workarounds and publish their findings. If you frequent their online hangouts, you're quite likely to find answers to your questions.

Whenever you find yourself stuck while working on a project in *The iOS Apprentice*, the first place you should go for help is [the online forum for this book](#). It's the perfect

place to ask questions if anything we've written has you confused or isn't working for you. A team of moderators keeps an eye on the forums, ensuring when you ask a question, you're not just screaming into the void. When they see a new question, they alert the authors — Yours Truly included — and we'll gladly help out.

Note: We'll update this edition of *The iOS Apprentice* when Apple updates iOS 13 and makes fixes, and those updates are included in the cost of the book! Make sure that you check this book's page on the raywenderlich.com site for new versions.

There are other places online that you may find useful — not just for dealing with SwiftUI bugs, but also for getting answers for your iOS programming questions and learning about other aspects of iOS programming. Here are some good starting points:

- [raywenderlich.com's forums](#): These cover not just iOS topics, but Android, Unity and Flutter development as well.
- [Stack Overflow](#): This is the best-known of all the developer forum sites out there. If you're a programmer, you'll eventually end up here looking for (or possibly dispensing) answers. You'll probably peruse the questions tagged [iOS](#), [Swift](#), and [SwiftUI](#) often.
- [Reddit's iOSProgramming subreddit](#): The biggest collection of forums online has one dedicated to iOS programming. It's so active that collectively, its readers have filed over 60,000 "radars" — that's Apple's term for bug reports and requests for features or enhancements.

Improving the user interface

Before closing out this chapter, make one more improvement to *Checklist*'s user interface that will make it more usable.

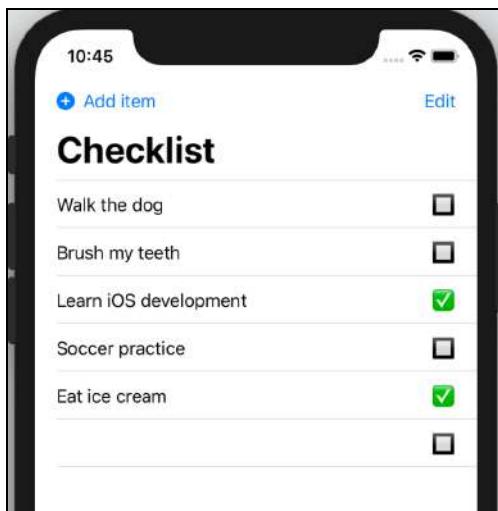
Disabling the “Add new item” button

Here's a question: What happens if you create a new checklist item without providing a name? It's time to be empirical and try it out.

► Run the app. Press the **Add item** button, which will display the **Add new item** sheet. **Without** entering a name for a new item, press the **Add new item** button.



Here's what you'll see when the **Add new item** sheet dismisses itself:



The checklist with an unnamed item

As you can see, the app currently lets the user add items without names to the list. The app shouldn't allow this; each item should have at least one character in its name.

You can fix this easily by using one of Button's methods: `disabled`, which accepts a single Boolean parameter. Setting this parameter to `true` disables the button so pressing it has no effect. Use it now to disable the button when the text field in the **Add new item** sheet is empty.

► Open **NewChecklistItemView.swift** and update **NewChecklistItemView**'s body to this:

```
var body: some View {
    VStack {
        Text("Add new item")
        Form {
            TextField("Enter new item name here", text: $newItemName)
            Button(action: {
                var newChecklistItem = ChecklistItem(name: self.newItemName)
                self.checklist.items.append(newChecklistItem)
                self.checklist.printChecklistContents()
                self.presentationMode.wrappedValue.dismiss()
            }) {
                HStack {
                    Image(systemName: "plus.circle.fill")
                    Text("Add new item")
                }
            }
        }
    }
}
```

```
        }
    .disabled(newItemName.count == 0)
}
Text("Swipe down to cancel.")
}
```

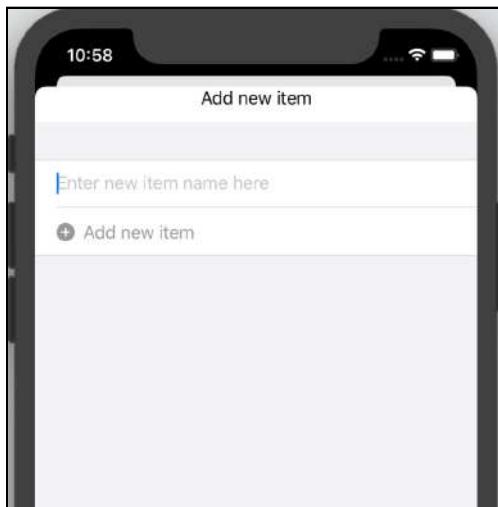
In making the change, you added one line to the end of the section that defines the button:

```
.disabled(newItemName.count == 0)
```

This line disables the button if `newItemName`'s `count` property is equal to zero. `newItemName` is a string property, and for properties that contain strings, the `count` property contains the number of characters in that string. An empty string — that is, a string that doesn't contain any characters — has a `count` of zero.

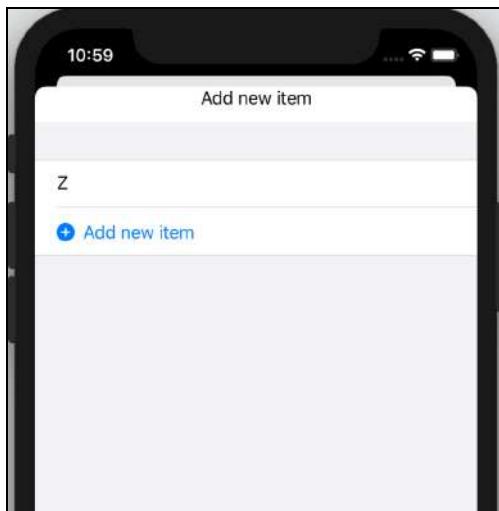
Note: To determine if one value is equal to another, use the “double-equals” operator, `==`. Don’t use the “single>equals”, `=`, which is for assigning values to constants and variables.

- Run the app. Press the **Add item** button, which will display the **Add new item** sheet. Note that the **Add new item** button is disabled:



The 'Add new item' sheet, with an empty text field and a disabled button

- Enter some text — any text — into the text field. The **Add new item** button will enable:



The 'Add new item' sheet, with a non-empty text field and an enabled button

If you delete all the text from the text field, the button will be disabled again. This has the effect you want: The user won't be able to add an item to the list unless it has at least one character in its name.

With *Checklists* now able to add new items to the list, you're one step closer to a fully-CRUD app!

Key points

- You learned about CRUD apps, and what CRUD stands for: Create, Report, Update and Delete.
- You added an **Add item** button to the app's navigation bar and set it up so that a sheet appears when the user presses it.
- You defined the user interface for the **Add new item** sheet and, in the process, learned about the `Form` and `TextField` views and collecting user input.
- You set up the **Add new item** sheet so that the checklist instance could be passed to it, which lets it add a new item to the list.
- You had a quick review of value and reference types.

- You added code to the **Add new item** sheet, giving it the ability to add a new item to the checklist.
- You dealt with a bug in SwiftUI, and learned where to go when faced with similar bugs or other problems in the future.
- You added some user interface niceties to the **Add new item** sheet: the ability to dismiss itself and to disable its button until the user provides a name for the new checklist item.

You'll find the project files for the app at this stage under **12 – Adding Items** in the Source Code folder.

In the next chapter, you'll make *Checklists* fully-CRUD and give it the ability to edit checklist items.

Chapter 13: Editing Checklist Items

Joey deVilla

In the previous chapter, you added a key feature to *Checklist*: The ability to add items to the list. You're no longer stuck with the five default items.

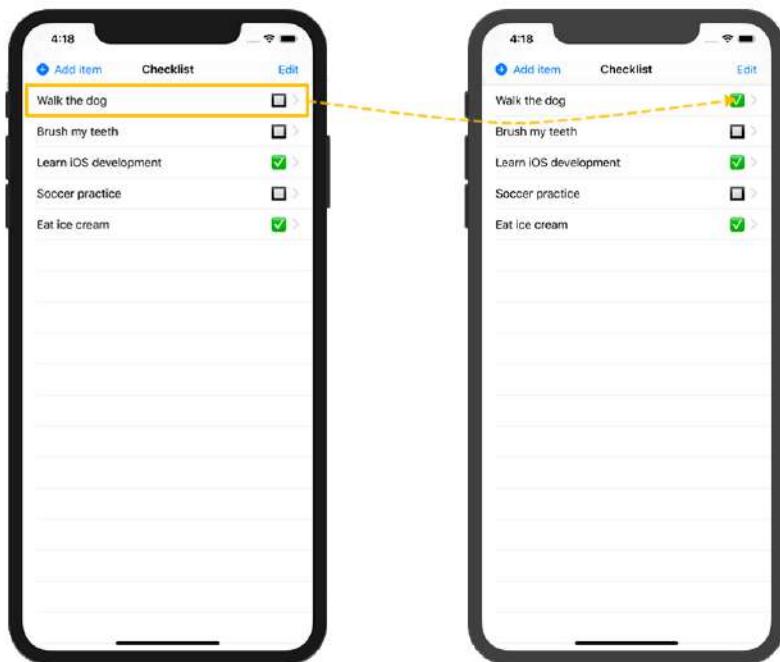
However, you still can't fully edit an item. You can change its status from checked to unchecked, and vice versa, but you can't change its name.

In this chapter, we'll make checklist items fully editable, allowing the user to change both their names and checked status.



Changing how the user changes checklist items

Right now, when the user taps on a checklist item to toggle the item's checked status. Tapping an unchecked item checks it, and tapping on a checked item unchecks it:



Tapping on a checklist item toggles its checked status

We're going to give the user the ability to change either the name of a checklist item or its checked status. This will require making changes to how the app works.

Let's look at the *Reminders* app that Apple includes on every iOS device as an example.

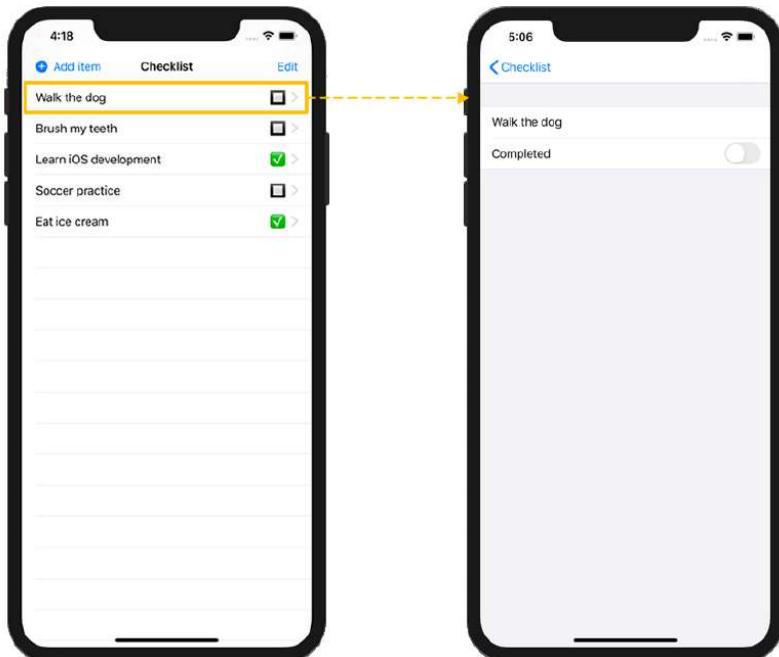
Here, tapping on an item's name allows you to edit the name, while tapping on an item's checkbox toggles its checked status:



Ideally, the user would tap on an item's name to edit it, and tap on its checkbox to check or uncheck it

Building this kind of user interface, as nice as it is, adds more complexity than an introductory tutorial should have. It would require changing the code in `ChecklistView` to support both showing the contents of the checklist and editing any given checklist item.

Instead, when the user taps a checklist item, we'll take them to an edit screen that allows them to edit both its name and checked status:



Tapping on a checklist item will take the user to an edit screen

The edit screen, which you'll code in this chapter, will contain a Form view similar to the one you included in the **Add new item** screen. This Form will contain a view that allows the user to change the checklist item's name and another view that allows the user to change its checked status.

With the changes that you'll make, you'll have a fully CRUD app by the end of this chapter. *Checklist* will be able to create, report, update and delete checklist items.

With that goal in mind, let's get started!

Giving checklist rows their own view

First, we should look at the way that ChecklistView draws the list of checklist items onscreen. Here's ChecklistView's body property:

```
// User interface content and layout
var body: some View {
    NavigationView {
        List {
            ForEach(checklist.items) { checklistItem in
                HStack {
                    Text(checklistItem.name)
                    Spacer()
                    Text(checklistItem.isChecked ? "✓" : "□")
                }
                .background(Color.white) // This makes the entire row
                .clickable
                .onTapGesture {
                    if let matchingIndex =
                        self.checklist.items.firstIndex(where: { $0.id ==
                            checklistItem.id }) {
                        self.checklist.items[matchingIndex].isChecked.toggle()
                    }
                    self.checklist.printChecklistContents()
                }
                .onDelete(perform: checklist.deleteListItem)
                .onMove(perform: checklist.moveListItem)
            }
            .navigationBarItems(
                leading: Button(action:
                { self.newChecklistItemViewIsVisible = true
                }) {
                    HStack {
                        Image(systemName: "plus.circle.fill")
                        Text("Add item")
                    }
                }
            )
        }
    }
}
```



```
        }
    },
    trailing: EditButton()
)
.navigationBarTitle("Checklist", displayMode: .inline)
.onAppear() {
    self.checklist.printChecklistContents()
}
.sheet(isPresented: $newChecklistItemViewIsVisible) {
    NewChecklistItemView(checklist: self.checklist)
}
```

There's a lot going on in this property. It:

- Draws each checklist item, including its name and checked status.
- Responds to presses on checklist items.
- Responds to the user moving a checklist item.
- Responds to the user deleting a checklist item.
- Draws the navigation bar and its items, including the **Add item** button, the **Edit** button, and the title.
- Responds to the user pressing the **Add item** button.

That's already a lot of responsibilities in one place, and that means a lot of complexity.

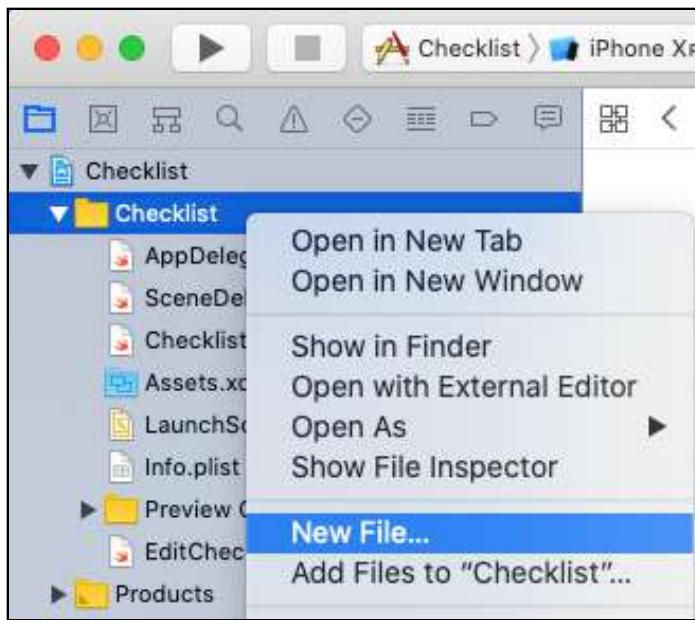
Consider a term I used a couple of times in Section 1 of this book: **Functional decomposition**. It's a fancy academic term that means “breaking down a big complex task into a set of smaller, simpler tasks.” We’re going to apply this principle to ChecklistView to simplify it. We’ll do this by splitting ChecklistView’s set of responsibilities into two groups:

We’ll do this by defining a new view that will be responsible for drawing individual checklist rows. We’ll then call on this view from ChecklistView.

Defining the new row view

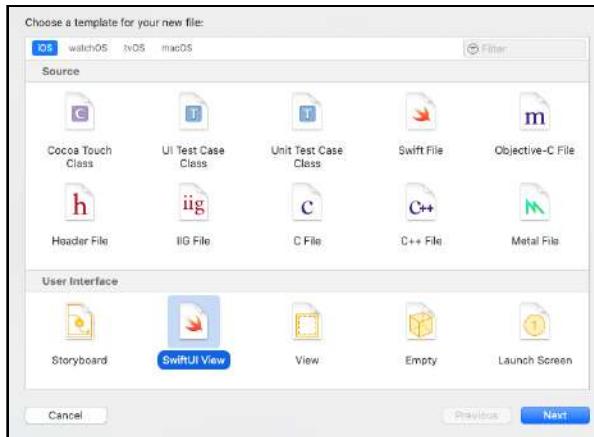
We’ll call this new view RowView, and we’ll put it in its own file, **RowView.swift**.

- Add a new file to the project by right-clicking or control-clicking on the **Checklist** folder in Xcode's Project Explorer. Select **New File...** from the menu that appears:



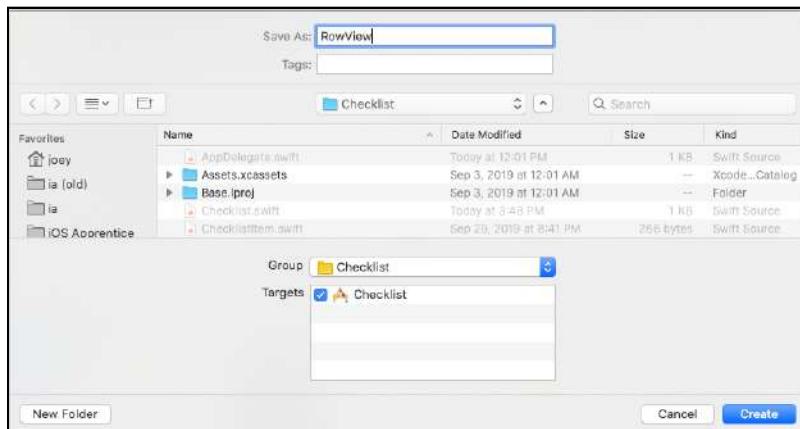
Add a new file to the project

- In the window that appears, make sure that you've selected **iOS**, then select **SwiftUI View** and click **Next**:



Select the 'SwiftUI View' template

- Enter **RowView** into the **Save As:** field. Make sure that you've selected **ChecklistItem** in the **Group** menu and in the **Targets** menu, then click **Create**:



Name the file 'RowView'

The project now has a new file named **RowView.swift**.

- Open **RowView.swift** and change the current definition of RowView to the following:

```
struct RowView: View {
    @State var checklistItem: ChecklistItem
    var body: some View {
        HStack {
            Text(checklistItem.name)
            Spacer()
            Text(checklistItem.isChecked ? "✅" : "◻")
        }
    }
}
```

As soon as you make this change, Xcode will report an error: **Missing argument for parameter 'checklistItem' in call...**

```
struct RowView_Previews: PreviewProvider {
    static var previews: some View {
        RowView() // ✎ Missing argument for parameter 'checklistItem' in call
    }
}
```

An error appears in the preview code

Let's fix the error first, then look at why it came up.

- In **RowView.swift**, change the preview section of the code to the following:

```
struct RowView_Previews: PreviewProvider {
    static var previews: some View {
        RowView(checklistItem: ChecklistItem(name: "Sample item"))
    }
}
```

With this change, the error message will disappear. Let's find out why.

Initializing structs

If you look through the structs that make up the app, you'll see that most of them have pre-defined properties. Let's look at the first struct that you defined for this app.

- Open **ChecklistView.swift** and look at its properties: `checklist`, `newChecklistItemViewIsVisible` and `body`. You'll see this. For brevity's sake, only the first few lines of `body` are shown:

```
@ObservedObject var checklist = Checklist()
@State var newChecklistItemViewIsVisible = false

// User interface content and layout
var body: some View {
    NavigationView {
        List {
            ...
        }
    }
}
```

All three of `ChecklistView`'s properties have initial values assigned to them:

- `checklist` is assigned the value `Checklist()`, which returns a new instance of `Checklist`.
- `newChecklistItemViewIsVisible` is assigned the value `false`.
- `body` is assigned a `NavigationView` that defines the user interface of the checklist screen.

Now, let's look at the app's newest struct: `RowView`.

- Open **RowView.swift** and look at its property: `checklistItem`:

```
@State var checklistItem: ChecklistItem
```

In this case, the property **doesn't** have a value assigned to it. `checklistItem` is declared as a variable that holds instances of `ChecklistItem`, but it doesn't have an initial value. It sits there, waiting for one.

With that observation, let's now take a look at the change to the preview section of `RowView.swift` that made the error disappear. It was a change from this:

```
struct RowView_Previews: PreviewProvider {
    static var previews: some View {
        RowView()
    }
}
```

To this:

```
struct RowView_Previews: PreviewProvider {
    static var previews: some View {
        RowView(checklistItem: ChecklistItem(name: "Sample item"))
    }
}
```

More precisely, you changed this line of code, which simply says, “Create a new instance of `RowView`”:

```
RowView()
```

To this:

```
RowView(checklistItem: ChecklistItem(name: "Sample item"))
```

This line also creates a new instance of `RowView`. It also specifies a value to be assigned the new `RowView` instance's `checklistItem` property: A new instance of `ChecklistItem`, with `name` property set to “Sample item.”

Since `RowView` doesn't assign an initial value to its `checklistItem` property, you have to provide an initial value whenever you create a new instance. That's why `RowView()` results in an error, but `RowView(checklistItem: ChecklistItem(name: "Sample item"))` doesn't.

This isn't the first time you've assigned values to `struct` instances while creating them. You also did it when you created the initial set of items for the checklist.

► Open `Checklist.swift` and look at its `items` property:

```
@Published var items = [
    ChecklistItem(name: "Walk the dog", isChecked: false),
```

```
ChecklistItem(name: "Brush my teeth", isChecked: false),  
ChecklistItem(name: "Learn iOS development", isChecked: true),  
ChecklistItem(name: "Soccer practice", isChecked: false),  
ChecklistItem(name: "Eat ice cream", isChecked: true),  
]
```

To create each of the checklist items in `items`, use `ChecklistItem(name:isChecked:)` to create a new instance of `ChecklistItem` and specify both its `name` and `isChecked` properties. For example, the line:

```
ChecklistItem(name: "Walk the dog", isChecked: false)
```

Says, “Create a new `ChecklistItem` whose `name` property is ‘Walk the dog’ and whose `isChecked` property is `false`.”

Since we’re instantiating `ChecklistItem` instances, let’s take a look at its properties.

► Open `ChecklistItem.swift` and look at its properties:

```
let id = UUID()  
var name: String  
var isChecked: Bool = false
```

Note the following:

- The first property, `id`, is assigned an initial value using the `let` keyword instead of a `var`. This means that it is a constant, and its value can’t be changed. It’s what’s called a **read-only** property; its value can be read, but not rewritten. You can’t assign a value to this property.
- The second property, `name`, isn’t given an initial value, which means you must provide one when creating an instance of this `struct`.
- The third property, `isChecked`, is assigned an initial value of `false` using the `var` keyword. This means that the value of `isChecked` can be changed — either when creating the instance, or at a later time.

How `ChecklistItem`’s properties are defined means that you have a couple of options when creating `ChecklistItem` instances. You can provide a value for the `name` property:

```
ChecklistItem(name: "Sweep the floor")
```

This creates a new `ChecklistItem` instance whose `name` value is “Sweep the floor” and whose `isChecked` value is the default value, `false`.

You can also provide values for both the `name` and `isChecked` properties:

```
ChecklistItem(name: "Clean the bathroom", isChecked: true)
```

This creates a new `ChecklistItem` instance whose `name` value is “Clean the bathroom” and whose `isChecked` value is `true`.

Going back to `RowView` and its property, remember that this line defines it:

```
@State var checklistItem: ChecklistItem
```

The property allows another object to specify which checklist item the row should represent. By not giving the property an initial value, the checklist item has to be specified when the row generates. You’ll see this in action in the next part, where we finally make use of this new view.

Updating ChecklistView to use RowView

Our goal was to make each checklist row responsible to drawing itself. Now that we’ve defined the view that lets rows do just that, let’s update `ChecklistView`.

► Open `ChecklistView.swift` and in the `body` property of `ChecklistView`, change the lines that define each row in the list from this:

```
HStack {  
    Text(checklistItem.name)  
    Spacer()  
    Text(checklistItem.isChecked ? "✅" : "◻")  
}
```

To this:

```
RowView(checklistItem: checklistItem)
```

► Run the app. It will appear to run as before, which means that we’ve successfully moved the responsibility of drawing individual rows from `ChecklistItem` to `RowView`.

Let's see what happens if you tap on a row.

- Tap on any item in the list. You'll see that it no longer checks or unchecks items.

Don't worry; we'll give individual rows the ability to respond to taps shortly.

In the meantime, since the code in ChecklistView that responds to taps on the list no longer works, let's remove it.

- Update the body property in ChecklistView by removing the code for handling taps on the list. The result should look like this:

```
var body: some View {
    NavigationView {
        List {
            ForEach(checklist.items) { checklistItem in
                RowView(checklistItem: checklistItem)
            }
            .onDelete(perform: checklist.deleteListItem)
            .onMove(perform: checklist.moveListItem)
        }
        .navigationBarItems(
            leading: Button(action:
                self.newChecklistItemViewIsVisible = true)) {
                Image(systemName: "plus")
            },
            trailing: EditButton()
        )
        .navigationBarTitle("Checklist")
        .onAppear() {
            self.checklist.printChecklistContents()
        }
    }
    .sheet(isPresented: $newChecklistItemViewIsVisible) {
        NewChecklistItemView(checklist: self.checklist)
    }
}
```

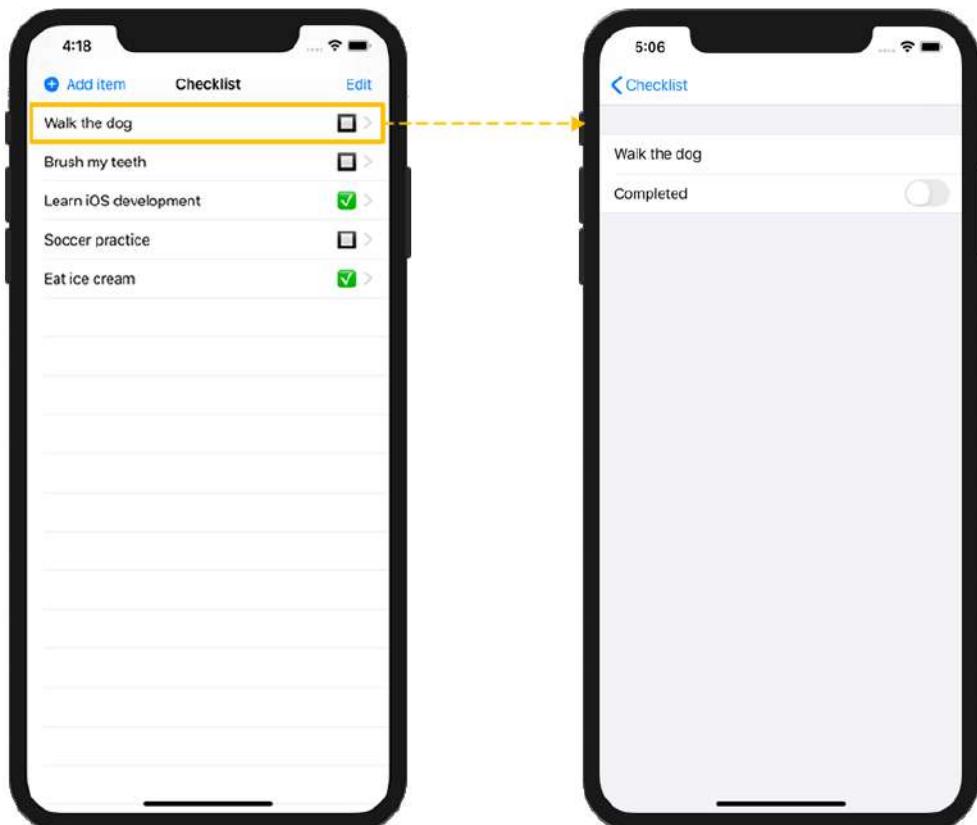
- Run the app to confirm that the changes you made didn't create any errors.

Just as we made each row responsible for drawing itself by moving the row-drawing code to RowView, we'll also make each row responsible for responding to user taps by moving the tap-response code to the same place.



Making rows respond to taps

Instead of checking or unchecking the corresponding item, tapping a row should take the user to a screen where they can edit both the item's name and checked status:



Tapping on a checklist item will take the user to an edit screen

You already have experience navigating between screens in a SwiftUI app. You used the `NavigationLink` view to provide the user a link that, when tapped, takes them to another screen. We'll use the same kind of view to take the user to an “Edit item” screen when they tap a checklist item.

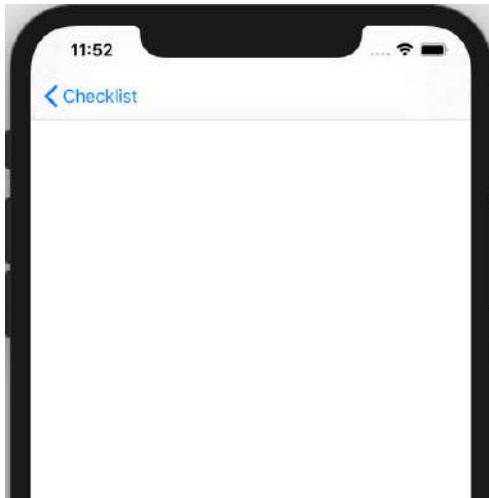
- Open **RowView.swift** and update the `body` property of `RowView` to the following:

```
var body: some View {
    NavigationLink(destination: EditChecklistItemView()) {
        HStack {
            Text(checklistItem.name)
            Spacer()
            Text(checklistItem.isChecked ? "✓" : "□")
        }
    }
}
```

You just took the `HStack` that defined a checklist row and put it inside a `NavigationLink`. This makes the entire row respond to taps from the user, and it will respond by taking the user to the view specified in the `destination:` parameter: A new instance of the `EditChecklistItemView` view.

Reminder: You created `EditChecklistItemView` and its file, `EditChecklistItemView.swift`, back in Chapter 11. Right now, `EditChecklistItemView` defines a mostly empty screen that says, “Hello World.”

- Run the app. Tap on any item in the list. You’ll see the following:



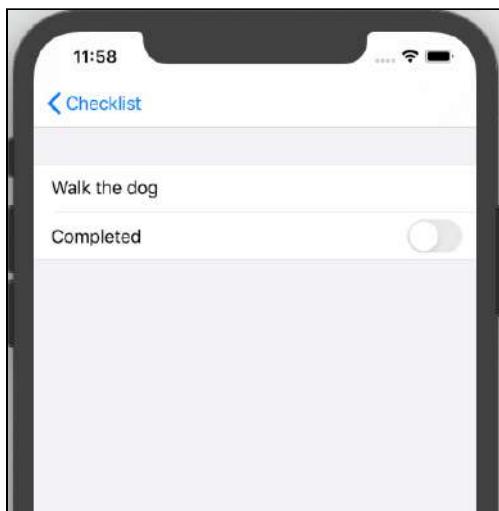
The initial “Edit checklist item” screen

Now that tapping on a checklist item takes you to `EditChecklistItemView`, it’s time to define that screen.



Defining EditChecklistItemView

Remember, when the user taps on a checklist item, we want them to see an “Edit” screen that looks like this:



The initial “Edit checklist item” screen

This screen should have the following:

- A `TextField` view containing the name of the selected checklist item. The user should be able to change the name of the checklist item by changing the text in this view. You used this control when creating the **Add new item** sheet in the previous chapter.
- A control that displays the current checked status of the checklist item. The user should be able to change the checked status of the item by toggling this control. We’ll use a `Toggle` view to create this control.

Just as we did with `NewChecklistItemView`, we’ll put these into a `Form` view that will organize and display them in a way that is most suitable for gathering user input.

► Open `EditChecklistItemView.swift` and change all the code below the 'Import SwiftUI' with the following:

```
struct EditChecklistItemView: View {  
    @State var checklistItem: ChecklistItem  
  
    var body: some View {  
        Form {
```

```
        TextField("Name", text: $checklistItem.name)
        Toggle("Completed", isOn: $checklistItem.isChecked)
    }
}

struct EditChecklistItemView_Previews: PreviewProvider {
    static var previews: some View {
        EditChecklistItemView(checklistItem: ChecklistItem(name: "Sample item"))
    }
}
```

You might be tempted to run the app right now to see how the `EditChecklistItemView` screen looks, but you won't be able to just yet. This new code has caused an error to pop up in `RowView`.

- Open `RowView.swift`. Look at `RowView`'s body property, and you'll see a familiar error: **Missing argument for parameter 'checklistItem' in call...**



The “Missing argument” error in `RowView`

Before you read on, ask yourself: How did you fix this error the last time you saw it?

The reason that the `NavigationLink` line now has an error is because of a key change you made in `EditChecklistItem`. You gave it a property that doesn't have an initial value: `checklistItem`. It's there so that the `NavigationLink` can do more than just bring up the “Edit item” screen. It can also tell the “Edit item” screen which item it's editing.

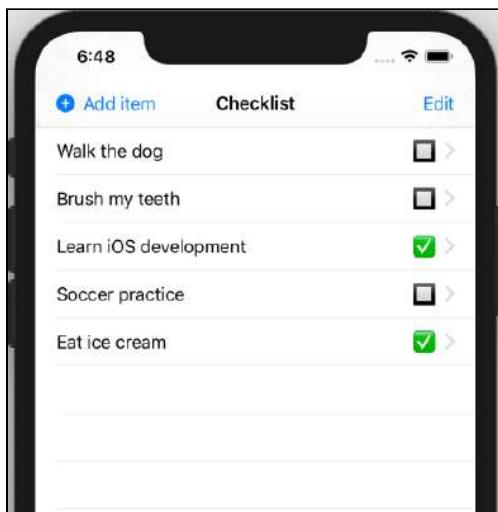
Since `EditChecklistItem`'s `checklistItem` property doesn't have an initial value, we need to provide that value when creating the `EditChecklistItemView` view. Let's do that.

- In **RowView.swift**, update the body property of RowView to the following:

```
var body: some View {
    NavigationLink(destination:
        EditChecklistItemView(checklistItem: checklistItem)) {
        HStack {
            Text(checklistItem.name)
            Spacer()
            Text(checklistItem.isChecked ? "✓" : "□")
        }
    }
}
```

With this change, the error should vanish. Let's see the “Edit item” screen in action now!

- Run the app:

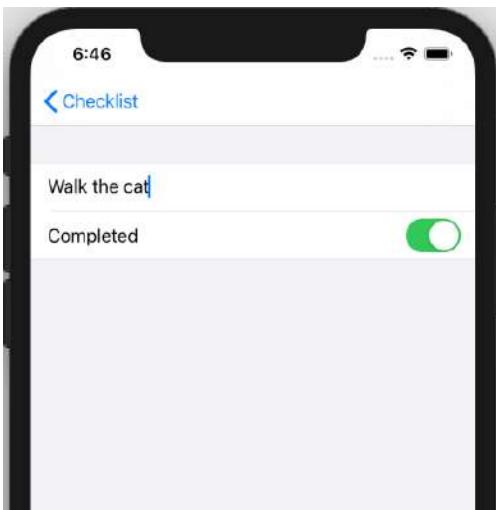


The checklist before attending to edit the 'Walk the dog' item

Let's try editing the **Walk the dog** item.

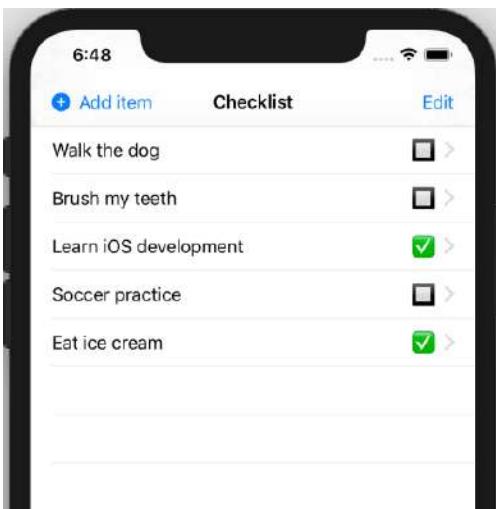
- Tap the **Walk the dog** row. The “Edit item” screen will appear, containing the item's current name and checked status.

- Change “Walk the dog” to “Walk the cat” and moved the **Completed** toggle from the “off” to the “on” position:



Editing a checklist item

- Now that you’ve made those edits, tap the < **Checklist** button in the Navigation Bar to return to the checklist. Here’s what you’ll see:



The checklist after attending to edit the 'Walk the dog' item

Your changes vanished! The first checklist item’s name is still “Walk the dog” instead of “Walk the cat,” and it remains unchecked instead of checked.

What happened?

Retracing our steps so far

As you progress as a developer, you’re going to have more of these experiences where you’re coding away, and everything seems fine when suddenly, you run into an unexpected problem. Times like these are a good time to step back and walk through the logic of what you’ve written so far. Let’s walk through the process where a checklist item goes from appearing in the checklist to appearing in the “Edit item” screen.

- Open **ChecklistView.swift** and look at its body property. Here’s the part of body that draws all the items in the checklist:

```
ForEach(checklist.items) { checklistItem in
    RowView(checklistItem: checklistItem)
}
```

The `ForEach` view goes through `checklist.items`, the array containing all the items in the checklist. For each item in that array, it creates a new `RowView` instance and, in doing so, sets that `RowView` instance’s `checklistItem` property to the current checklist item.

Each checklist item is an instance of `ChecklistItem`, which is a struct. That means that when you set a `RowView` instance’s `checklistItem` property, you’re giving the `RowView` instance its own copy of the checklist item.

- Open **RowView.swift** and look at its body property. Here’s the line in body that determines what happens when the user taps the row:

```
NavigationLink(destination: EditChecklistItemView(checklistItem:
    checklistItem)) {
```

The `NavigationLink`, when tapped, takes the user to the view specified in its `destination:` parameter. In this case, the destination is a new `EditChecklistItemView` view. In creating the new `EditChecklistItemView`, we set its `checklistItem` property to the checklist item used by `RowView`.

Once again, the checklist item that we’re passing to `EditChecklistItemView` is a struct, which means that we’re giving the `EditChecklistItemView` instance its own copy of the checklist item, which in turn is a copy of the checklist item from `ChecklistView`.



Here's what you should take from all this retracing: When you're editing a checklist item in `EditChecklistItemView`, you're editing a copy of a copy of an item in the checklist. That's why your changes to the "Walk the dog" item don't appear in the checklist after you dismiss the "Edit item" window.

What we need is a way to pass a connection to the actual checklist item from `ChecklistView` to `RowView` to `EditChecklistItemView` instead of a mere copy. That way, any changes made in `EditChecklistItemView` will be made in the checklist.

@Binding properties

Luckily for us, there *is* a way to pass a connection to a checklist item rather than a copy. Let's make use of it by starting with `EditChecklistItemView`.

Updating `EditChecklistItemView`

- Open `EditChecklistItemView.swift`. Change the line that defines the `checklistItem` property from this:

```
@State var checklistItem: ChecklistItem
```

To this:

```
@Binding var checklistItem: ChecklistItem
```

You've just changed `checklistItem` from a `@State` property to a `@Binding` property. As a `@State` property, `checklistItem` was a property that belonged to `EditChecklistItemView`. When a `RowView` instance passes a checklist item to an `EditChecklistItemView` instance via the `checklistItem` item property, it makes a copy of `RowView`'s checklist item. Any changes made to the checklist item in `EditChecklistItemView` aren't reflected in the matching checklist item in `RowView`, which is what we want.

As a `@Binding` property, `checklistItem` is a connection to another object's property. Now, when a `RowView` instance passes a checklist item to an `EditChecklistItemView` via the `checklistItem` item property, any changes made to the checklist item in `EditChecklistItemView` will be reflected in the matching checklist item in `RowView`.



This change will cause an error in the preview code, whose code is trying to pass put a checklist item into a property that now expects a binding to a checklist item:



The error message that appears in the preview section

- Update the preview code in `EditChecklistItemView` to the following:

```
struct EditChecklistItemView_Previews: PreviewProvider {
    static var previews: some View {
        EditChecklistItemView(checklistItem: .constant(ChecklistItem(name: "Sample item")))
    }
}
```

Wrapping `ChecklistItem(name: "Sample item")` inside the `.constant` function creates a binding to a checklist item, which is the kind of value that the `checklistItem` property expects.

This completes all the changes we need to make to `EditChecklistItemView`. It's time to edit the blueprint for objects that pass checklist items to `EditChecklistItemView: RowView`.

Updating RowView

- Open `RowView.swift`. Change the line that defines the `checklistItem` property from:

```
@State var checklistItem: ChecklistItem
```

To:

```
@Binding var checklistItem: ChecklistItem
```

This should give you a sense of *déjà vu*, and with good reason. You made the exact same changes in `EditChecklistItemView`! The connection to a checklist item that `EditChecklistItemView` receives from `RowView` is, in fact, a connection that `RowView` will receive from `ChecklistItemView`.

Since `RowView` will not be passing a checklist item to `EditChecklistItemView`, but a **binding** to a checklist item, we need to specify that.

- Change the `NavLink` line in `RowView`'s `body` property from:

```
NavLink(destination: EditChecklistItemView(checklistItem:  
checklistItem)) {
```

To:

```
NavLink(destination: EditChecklistItemView(checklistItem:  
$checklistItem)) {
```

The change is so subtle that you might have missed it. Instead of setting `EditChecklistItemView`'s `checklistItem` property to `checklistItem`, you're now setting it to `$checklistItem`. The `$` makes the difference: `checklistItem` is a checklist item, and `$checklistItem` is a **binding** to a checklist item.

Just as with `EditChecklistItemView`, changing `RowView`'s `checklistItem` property into a `@Binding` created an error in the preview code. Once again, it's a matter of changing its code so that it passes a binding to a checklist item and not just a checklist item to `RowView`.

- Update the preview code in `RowView` to the following:

```
struct RowView_Previews: PreviewProvider {  
    static var previews: some View {  
        RowView(checklistItem: .constant(ChecklistItem(name: "Sample  
item")))  
    }  
}
```

We're done making the necessary changes to `RowView`. But, there's one more object blueprint to edit: `ChecklistView`.

Updating ChecklistView

Just as `RowView` passes a binding to its checklist item to `EditChecklistItemView`, we want `ChecklistView` to pass bindings to checklist items to `RowView`. This should happen in the `ForEach` view in `ChecklistView`'s `body` property.

- Open `ChecklistView.swift` and look at the `ForEach` view in the `body` property:

```
ForEach(checklist.items) { checklistItem in  
    RowView(checklistItem: checklistItem)  
}
```

Since the `checklistItem` property of `RowView` now holds bindings to checklist items instead of checklist items, the current code causes Xcode to display an error message:

```
ForEach(checklist.items) { checklistItem in
    RowView(checklistItem: checklistItem)
}
```

The error message that appears in ChecklistView

This should easily be fixed by changing the value we put into `RowView`'s `checklistItem` property from a checklist item into a binding to a checklist item by prefacing it with a `$` character.

- Change the `ForEach` view in the `body` property to the following:

```
ForEach(checklist.items) { checklistItem in
    RowView(checklistItem: $checklistItem)
}
```

That won't work either:

```
ForEach(checklist.items) { checklistItem in
    RowView(checklistItem: $checklistItem)
}
```

The resulting error message in ChecklistView

The error message, **Use of unresolved identifier '\$checklistItem'**, is Xcode's way of saying: "I have no idea what you mean by '\$checklistItem.' The problem is that you can only create a binding to a `@State` or `@Binding` variable, and the `checklistItem` inside `ForEach`'s braces is neither.

The perils of new platforms, again

In the previous chapter, we worked around a bug that caused strange behavior in the navigation bar buttons. You've just run into another rough edge that comes with working with a brand new platform like SwiftUI. There **is** a workaround, but it requires learning about another Swift feature.

Introducing extensions

Sometimes a struct or class gives you *almost* all the functionality you need. If it's one that you wrote or have the source code for, you can add that missing functionality by writing more properties and methods. But what do you do when you didn't write the struct or class, and you don't have the source code?

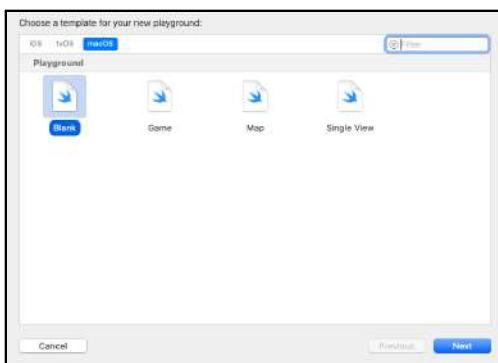


That's when you use **extensions**. They're a way for you to say: "Here's some extra code that I'd like to add to the struct or class."

Making a simple extension

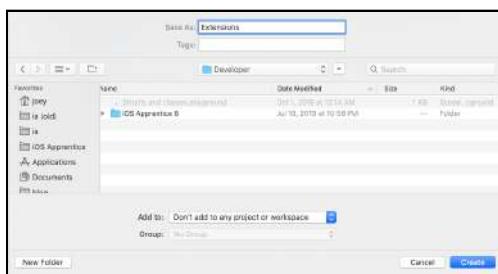
The best way to understand extensions is to see them in action, and the simplest way to do that is to start another Xcode playground session!

- In Xcode's **File** menu, select **New ➤** and then **Playground....** The **Choose a template for your new playground** window will appear. Select **macOS** and **Blank**, then click **Next**.



Options for creating a new playground

- The **Save as:** window will appear. Enter a name for the playground. I used **Extensions**. In the **Add to:** menu, select **Don't add to any project or workspace**. Once you've done that, click the **Create** button:



Choosing a place to save the playground

- Replace the code in the playground with the following:

```
print(true.asYesOrNo)
print(false.asYesOrNo)
```

Soon after you enter the code, you'll see the following error messages:

```
print(true.asYesOrNo)  
print(false.asYesOrNo)
```

Value of type 'Bool' has no member 'asYesOrNo'
Value of type 'Bool' has no member 'asYesOrNo'

The 'Bool' types doesn't have an 'asYesOrNo' property...yet

That's because `true` and `false` are both instances of the `Bool` type, which doesn't have a property called `asYesOrNo`. `Bool` is a struct, which means that we can add a property to it using an extension.

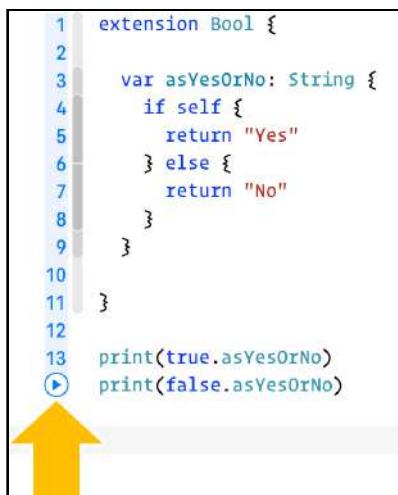
The property we'll add will be called `asYesOrNo`, and it will return the string "Yes" if the `Bool`'s value is `true` and the string "No" if the `Bool`'s value is `false`.

- Change the contents of the playground to the following:

```
extension Bool {  
  
    var asYesOrNo: String {  
        if self {  
            return "Yes"  
        } else {  
            return "No"  
        }  
    }  
  
}  
  
print(true.asYesOrNo)  
print(false.asYesOrNo)
```

Let's test the extension.

- Move the cursor over the number for the last line of code in the playground and click the "Play" button that appears in the margin:



```
1 extension Bool {
2
3     var asYesOrNo: String {
4         if self {
5             return "Yes"
6         } else {
7             return "No"
8         }
9     }
10
11 }
12
13 print(true.asYesOrNo)
14 print(false.asYesOrNo)
```

Testing the extension in the playground

The debug console will show the output of both the print statements: “Yes” for `true.asYesOrNo` and “No” for `false.asYesOrNo`.

That’s the power of extensions — they let you add functionality to objects, even if you don’t have access to their source code.

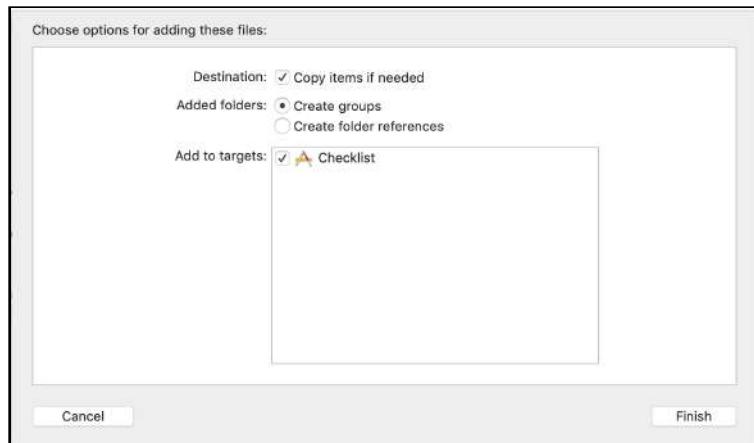
Adding extensions to Checklist

Let’s get back to the issue that we currently have with *Checklist*.

We need a way for `ChecklistView` to go through each item in the checklist and give `RowView` a binding to each item. SwiftUI doesn’t have a built-in way to do this, but we’ve written some extensions that make up for this shortcoming.

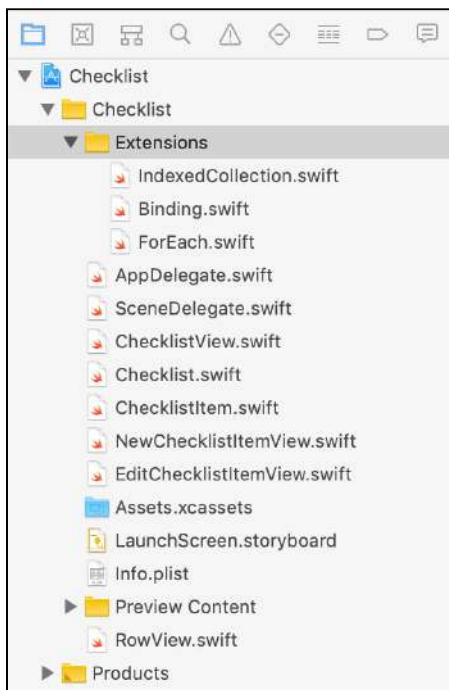
- Open the **Resources** folder that comes with this book, and then open the **Checklist** subfolder. Inside that folder, you’ll find a folder named **Extensions**. Drag this folder onto the yellow **Checklist** folder in the Xcode project.

- When the **Choose options for adding these files:** window appears, make sure that the **Copy items if needed** checkbox is checked, the **Create groups** option is selected and that the **Checklist** item in the **Add to targets** menu is checked:



Choose options for adding these files

The project should look similar to this in Xcode's Project Navigator:



The extensions folder in Xcode

Updating EditChecklistItemView

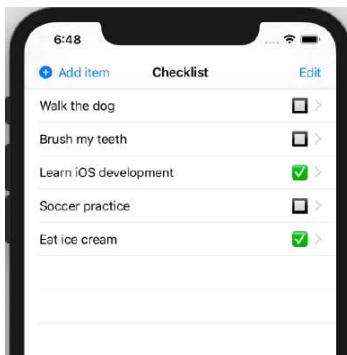
Now that the project has the necessary extensions, let's make use of them!

- Open **ChecklistView.swift**. Change the **ForEach** view in the **body** property to:

```
ForEach(checklist.items) { index in
    RowView(checklistItem: self.$checklist.items[index])
}
```

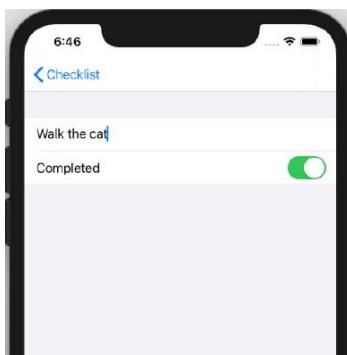
With the help of the extensions, this code goes through `checklist.items` and passes a binding to each item to `RowView`.

- Run the app. It should display the default list of items:



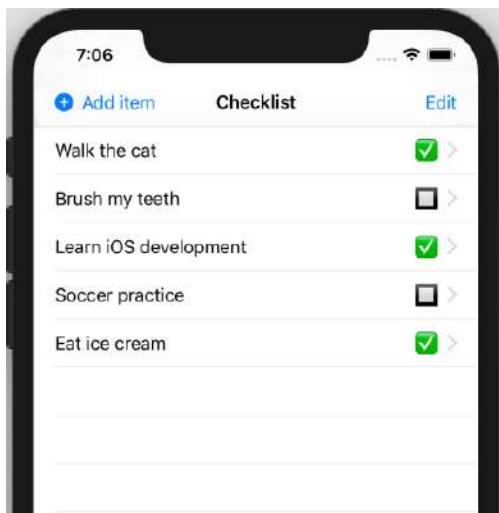
The checklist before editing the 'Walk the dog' item

- Select a checklist item to edit by tapping on one of them. In this example, I tapped on the first item, “Walk the dog” and edited it by changing its name to “Walk the cat” and changing its status to completed:



Editing a checklist item

- Tap on the < Checklist button in the upper left-hand corner of the screen to return to the checklist. You'll see that this time, your edits remain!



The checklist after editing the 'Walk the dog' item

Congratulations — Checklist is now CRUD!

Key points

In this chapter, you:

- Created a new view, allowing rows to draw themselves and respond to taps independently.
- Learned more about initializing `structs` and their properties.
- Updated Checklist's user interface to support both checking items and editing their names.
- Defined the “Edit item” screen.
- Learned about how `@Bindings` can be used to shared properties among screens.
- Learned about extensions and how to use them to extend the functionality of objects.

- Used extensions to get around a rough edge in SwiftUI.
- Brought the app to the point where it can list checklist items, create a new checklist item, edit an existing checklist item and delete checklist items. You have a full CRUD app now!

In the next chapter, we'll add a much-needed capability to checklist: The ability to remember list items between sessions.

Chapter 14: Saving and Loading

Joey deVilla

Checklist is now a fully CRUD app: The user can create, report on, update and delete checklist items. It would be a fully-functional basic checklist app if not for two issues. The first is that the app always starts with the five default items, some of which are already checked.

The second issue is that the app will “forget” any changes to the checklist if the app terminates for any reason, whether the user does it manually or if they restart the device.

It’s time to make some changes so that the app behaves in the following ways:

- When the user launches *Checklist* for the first time, they’ll start with an empty checklist instead of the five default checklist items that the app’s had since the beginning.
- The app will remember the state of its checklist after it terminates. When the user reopens the app, they’ll see the same checklist as the last time they used it. The contents of the checklist should *persist* over time.



In this chapter, you'll cover the following topics:

- **Data persistence:** In most cases, apps need to remember something from the last time you used them.
- **The Documents folder:** Each app has its own place where it can store data. You'll learn how to find this place and use it to store checklist items.
- **Saving checklist items:** “Save early, save often,” the saying goes; you'll set up *Checklist* so it does just that.
- **Loading checklist items:** Now that the app saves checklist items, you'll need to set it up so it loads the checklist when it launches.

Data persistence

Modern smartphone operating systems are technological wonders. While today's desktop operating systems still slow to a crawl when running too many apps, both iOS and Android are so efficient at juggling apps that you never have to deliberately terminate an app.

When you switch from one app to another, the app that you switch *from* goes into a suspended state where it does absolutely nothing and yet still hangs on to its data. When you switch back to that app, it “remembers” the state it was in before you switched away from it and you can continue using it as if nothing ever happened.

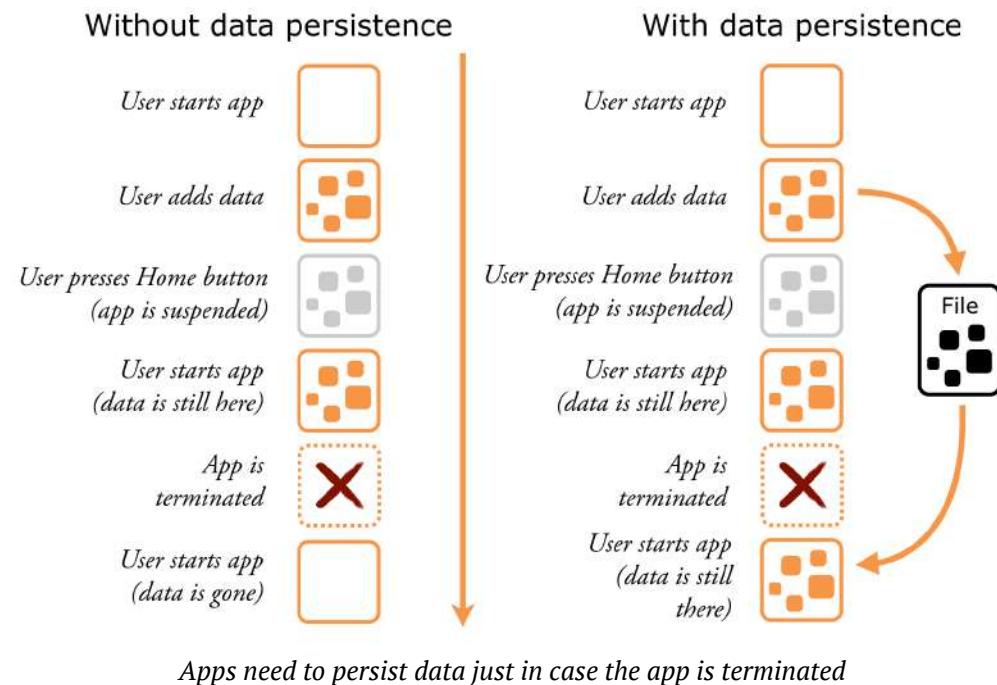
However, it's not a perfect world, and sometimes an app will terminate. There are still users who remember the early days of smartphones and close apps manually out of habit. Apps and operating systems can also crash, requiring a restart. And sometimes a device will run out of power before you can recharge it.

Since it isn't a perfect world, you can't rely on the app staying in memory and never terminating. Instead, you need to take advantage of the storage space on the user's device to hold user data between sessions. It's not just for cat pictures and videos!

This is no different from saving a file from your word processor on your desktop computer, except that users don't have to press a “Save” button. Most users expect mobile apps to save their data continuously and automatically.



Saving data between app launches is called **data persistence**. You'll add this feature to *Checklist* in this chapter.



If you were working on a traditional desktop app, implementing data persistence might take a fair bit of code. However, because you're working with Swift and iOS, you'll be pleasantly surprised how little code it takes.

It's time to start working on that data persistence functionality! The first step is to figure out *where* you'll store the data.

The Documents directory

Unlike desktop apps, which mostly have unfettered access to the computer's hard drive, each iOS app goes into a **sandbox** when installed. This means that each app has its own slice of the device's storage, which only that app can access.

This is a security measure designed to prevent malicious software from doing any serious damage. If an app can change only its own files, it can't modify or mess with any other part of the system.

Think of the sandbox as your app's very own hard drive. Within that hard drive is the **Documents** directory, which is the designated place for your app to store data.

Note: If you're unfamiliar with the term “directory”, it's just the more computer science-y term for “folder.”

The **Documents** folder has a couple of benefits:

- **Automatic backup:** When the user syncs their device with their computer or iCloud, the system also backs up the **Documents** directories for their apps.
- **Persistence between updates:** When you release a new version of your app, the update doesn't touch the app's **Documents** directory. Any saved data in this directory persists between updates.

With its security features and benefits, the **Documents** folder is the perfect place to store your app's data files.

Now that you know about the **Documents** folder, it's time to find it so that you can put it to use.

Finding the Documents directory to save checklist data

- Add the following methods to **Checklist.swift** after the `moveListItem(whichElement:destination:)` method:

```
func documentsDirectory() -> URL {
    let paths = FileManager.default.urls(for: .documentDirectory,
                                         in: .userDomainMask)
    return paths[0]
}

func dataFilePath() -> URL {
    return
    documentsDirectory().appendingPathComponent("Checklist.plist")
}
```

The first method, `documentsDirectory()`, returns the location of the app's **Documents** directory. It does this by using the built-in `FileManager.default` object, which is the preferred way to access the file system in an app's sandbox.

`FileManager.default` has a method called `urls(for:in:)`, which lets you specify a kind of directory to look for — in this case, the **Documents** directory — and returns an array containing one or more URLs where you may find them. Even though each app has just one **Documents** directory, `urls(for:in:)` returns an array of results. The path for the **Documents** directory is in that array's first and only element, which is what `documentsDirectory()` returns.

The second method, `dataFilePath()`, uses the result of `documentsDirectory()` to construct the full path to the file that will store the checklist items. This file is named **Checklist.plist** and it will live inside the **Documents** folder.

Notice that both methods return a URL object. You may think of a URL as a “web address”, but it’s really just a path for a given directory or file, which can be either online or on the local system. iOS uses URLs to refer to files in its file system. When a URL begins with `http://` or `https://`, it refers to a directory or file on the web. When it refers to a local file, a URL will begin with `file://`.

Note: Double check to make sure your code says `.documentDirectory` and not `.documentationDirectory`. Xcode’s autocomplete can easily trip you up here!

Now that you have these methods, go ahead and put them to use.

- Still in **Checklist.swift**, add the following method to the start of the **Methods** section, immediately after the “Methods” comment:

```
init() {
    print("Documents directory is: \(documentsDirectory())")
    print("Data file path is: \(dataFilePath())")
}
```

`init()` is a method that you haven’t seen before. Its name comes from the term “initializer.” It’s a special method built into structs and classes that are automatically called when a new instance is created.

You use Initializers to set up or *initialize* an object at the moment when it’s created, and you can also use them to perform tasks at that moment.

Right now, you’re using the `init()` method to print the paths of the **Documents** directory and the file where you’ll save the checklist to Xcode’s debug console. Later on, you’ll use it to restore the saved checklist when the app starts up.



- Run the app and look at Xcode’s debug console. It will display the file paths of the **Documents** directory and where the app will eventually save **Checklist.plist**.

The full names of the file paths will differ from device to device. When I run the app on the Simulator on my computer, the Xcode debug console displays this:

```
Documents directory is:  
file:///Users/joey/Library/Developer/CoreSimulator/Devices/C74A344A-0  
9BB-4E7E-8A24-882C58C421A1/data/Containers/Data/Application/43286A9D-  
86D5-4DAC-A546-E892EA375F41/Documents/  
Data file path is:  
file:///Users/joey/Library/Developer/CoreSimulator/Devices/C74A344A-0  
9BB-4E7E-8A24-882C58C421A1/data/Containers/Data/Application/43286A9D-  
86D5-4DAC-A546-E892EA375F41/Documents/Checklists.plist
```

Console output showing Documents folder and data file locations

If you run the app on your iPhone, your path will look slightly different. Here’s what mine says:

```
Documents directory is: file:///var/mobile/Containers/Data/  
Application/5F4CB154-1CAD-4F54-8673-4ADCCE98D78/Documents/  
Data file path is: file:///var/mobile/Containers/Data/  
Application/5F4CB154-1CAD-4F54-8673-4ADCCE98D78/Documents/  
Checklist.plist
```

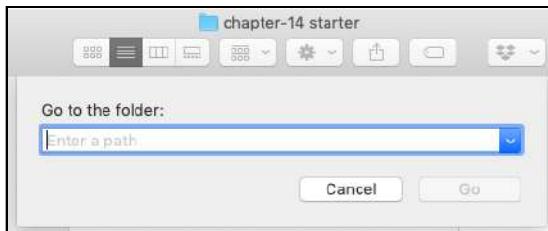
As you’ll notice, the sandbox’s directory name is a set of random characters that are determined when you install the app. Anything inside that directory, such as the **Documents** directory, is part of the app’s sandbox.

Browsing the **Documents** directory

For the rest of this chapter, run the app on the simulator instead of a device. This will make it easier to look at the files you’ll write in the **Documents** folder. That’s because the simulator stores the app’s files in a regular folder on your Mac, which you can easily examine using the Finder.

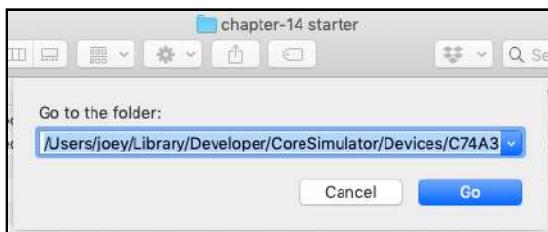
- Run the app in the simulator. When the path for the **Documents** directory appears in Xcode’s debug console, copy it. Don’t include the **file://** bit — the path starts with **/Users/yourname/....**
- Open a new Finder window by clicking on the Desktop and typing **Command+N** or by clicking the Finder icon in your dock, if you have one. Then press **Command+Shift+G** or select **Go ▶ Go to Folder...** from the menu.

You'll see a dialog box that says **Go to the folder:**



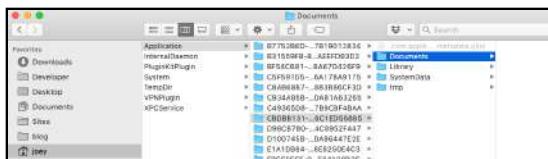
The 'Go to the folder:' dialog box

- Paste the full path of the **Documents** folder into the text field in the dialog box and click the **Go** button:



The Finder window shows you the contents of that folder.

Keep this window open; you'll want to be able to verify that **Checklist.plist**, the file containing the checklist data, is actually created when the time comes.



The app's directory structure in the Simulator

Note: If you want to navigate to the simulator's app directory by traversing your folder structure, you should know that the **Library** folder, which is in your home folder, is normally hidden. If you can't see the **Library** folder, hold down the **Alt/Option** key and click on Finder's **Go** menu (or hold down the **Alt/Option** key while the **Go** menu is open). This should reveal a shortcut to the **Library** folder on the **Go** menu, if it wasn't visible before.

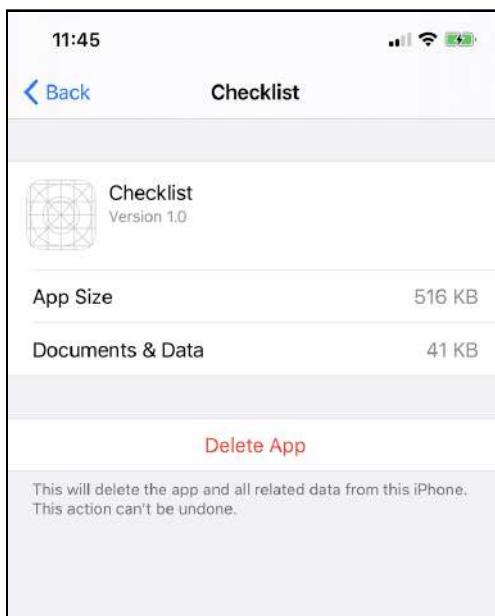
You can see several folders inside the app's sandbox folder:

- The **Documents** folder, where the app will put its data files. It's currently empty.
- The **Library** folder has cache files and preferences files. The operating system manages the contents of this folder.
- The **SystemData** folder is for the operating system to use to store any system-level information relevant to the app.
- The **tmp** folder is for temporary files. Sometimes, apps need to create files for temporary use. You don't want these to clutter up your Documents folder, so tmp is a good place to put them. iOS will clear out this folder from time to time.

You can also get an overview of the **Documents** folder of other apps on your device.

- On your device, open **Settings** and then select **General** ▶ **iPhone Storage**. Scroll down to the list of installed apps and tap the name of an app.

You'll see the size of its Documents folder, but not the actual content:



The Documents folder on the device

Now that you have a good understanding of where you'll save your app's information, it's time to move on to implementing your save functionality.

Saving checklist items

For your next step, you'll write the code that will save the list of to-do items. These items will save to a file named **Checklist.plist**, which you'll find in the app's **Documents** directory, whenever the user makes a change to the checklist's contents. Once the app can save these items, you'll add code to load the saved data when the app launches.

.plist files

You probably looked at the name of the checklist data file, **Checklist.plist** and wondered, "What's a **.plist** file?"

You've already seen a file named **Info.plist** back when you were working on *Bullseye*. All apps, *Checklist* included, have such a file, which you can see if you look at its files in Xcode's Project navigator. **Info.plist** contains information about the app for iOS to use, such as what name to display under the app's icon on the home screen.

"**.plist**" is short for **Property List**. It's an XML file format that stores app settings and their corresponding values. In iOS, .plist files are often used for storing app data, as they're simple to read for both apps and their human programmers.

The Codable protocol, encoding, and decoding

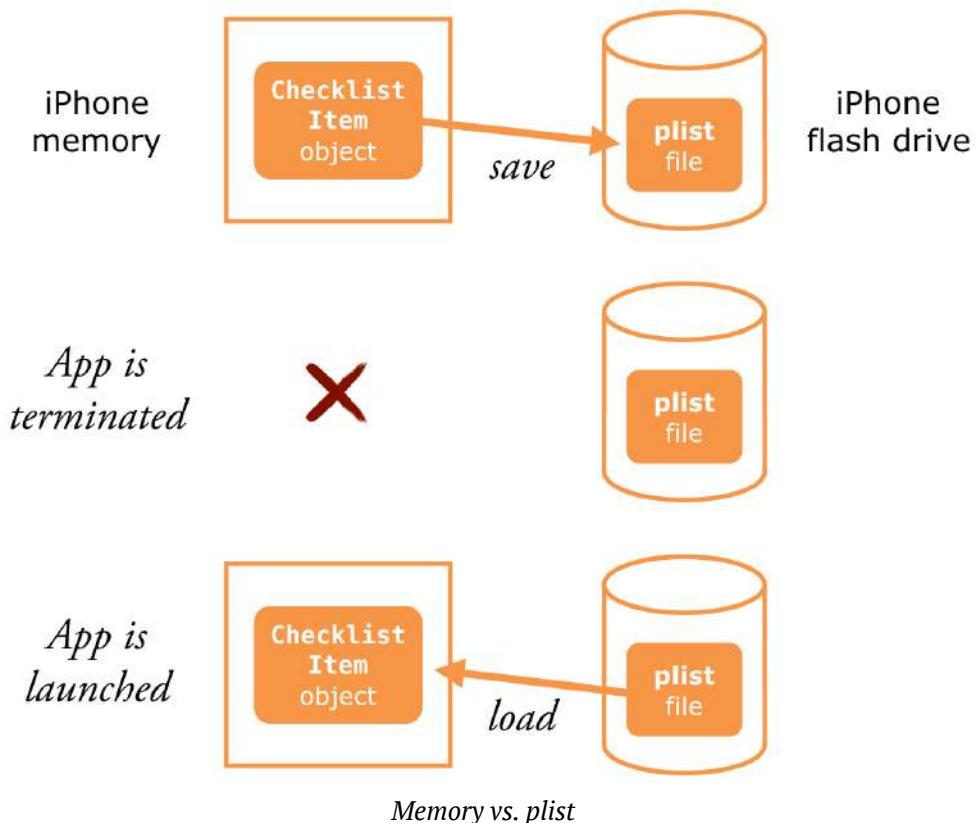
In the past few chapters, you've had so much new information thrown at you that you might have forgotten what a **protocol** is — at least in the Swift sense. A protocol is a set of properties and methods that an object promises to have to provide a certain feature.

For the app to save its checklist items, you'll use the **Codable** protocol, which gives objects the ability to save their data to and load their data from the file system.

The beauty of **Codable** is that it insulates you from having to know much about the format of the files it writes. In this case, **Codable** will save the checklist item data in a .plist file. You won't have to work with the file directly. All you care about is that the data is stored as a file in the app's **Documents** folder, and **Codable** will do most of the work.

The name **Codable** captures the two kinds of tasks it will perform, namely:

- **Encoding**, which is converting an object's data from its form in system RAM into a form that you can write to "disk"... or, in this case, the device's flash drive. Think of encoding as saving a file in a word processor.
- **Decoding**, which is reading data stored on "disk" and converting it back into a form that an app's object can use. Think of decoding like loading a file in a word processor.



The process of converting objects to files and back again is known as **serialization**. It's a big topic in software engineering.

Programmers use all sorts of metaphors for serialization, and many of them revolve around food preservation techniques. Some programmers think of serialization like taking a living object and freezing it, preserving it and suspending it in time. You store that frozen object in a file on the device's flash drive, where it will spend some

time in cryostasis. Later, you can read that file into memory and defrost the object, bringing it back to life again.

Saving data to a file

Now that you have a method that determines *where* the app will write **Checklist.plist**, it's time to write a method to save that file.

► Add the following method to **Checklist.swift**:

```
func saveListItems() {  
    // 1  
    let encoder = PropertyListEncoder()  
    // 2  
    do {  
        // 3  
        let data = try encoder.encode(items)  
        // 4  
        try data.write(to: dataFilePath(),  
                      options: Data.WritingOptions.atomic)  
        // 5  
    } catch {  
        // 6  
        print("Error encoding item array: \  
              \(error.localizedDescription)")  
    }  
}
```

This method takes the contents of the `items` array, converts it to a block of binary data and then writes this data to the **Checklist.plist** file in the app's **Documents** directory.

In order to understand this code, go through the commented lines step-by-step:

1. First, the method creates an instance of `PropertyListEncoder`, a type of object that Apple operating systems use to encode the data stored in an app's objects into a property list.
2. The `do` keyword, which you haven't encountered before, sets up the first of two blocks of code, which are Swift's way of catching errors that might come up when the program is running.

```
do {
```

Code that attempts to do something
that has a chance of failing or resulting in an error

```
} catch {
```

Code that will be executed if the code
in the do block fails or results in an error

```
}
```

'do' and 'catch blocks, illustrated'

The do block contains code that might fail or result in an error — or, as you say in programming, *throw* an error. Under normal circumstances, this would cause the app to come to a crashing halt. The do block changes all that: It lets you mark lines of code that might fail with the `try` keyword, and if any of those lines throw an error, the code in the catch block takes over.

3. Here, you call the encoder’s `encode()` method to encode the `items` array. The method could fail. It throws an error if it’s unable to encode the data for some reason: Perhaps it’s not in the expected format, or it’s corrupted, or the device’s flash drive is unavailable.

The `try` keyword indicates that the call to `encode` can fail and if that happens, that it will throw an error. The `try` keyword is mandatory when calling methods that throw errors; in fact, if you remove the `try` keyword that comes before `encoder.encode(items)`, Xcode will display an error message.

If the call to `encode()` fails, execution will immediately jump to the catch block instead of proceeding to the next line.

4. If the call to `encode()` succeeds, `data` now contains the contents of the `items` array in encoded form. This line attempts to write this encoded data to a file using the file path returned by a call to `dataFilePath()`. The `write()` method, like many file operations, can fail for many reasons and throw an error. Once again, you have to make use of a `try` statement, so the catch block can handle the case where `write()` fails.
5. This is the start of the catch block, which contains the code to execute if any line of code in the do block threw an error.
6. This is the code that executes if code in the do block throws an error. If you were planning to sell this app in the App Store, you might do all kinds of things with



this code to deal with cases where encoding the data or writing it to the device's file system fails. In this case, you'll simply print out an error message to Xcode's console.

You might notice that the `print()` statement references an `error` variable. Where did that come from?

When you create a pair of `do — catch` code blocks, you can explicitly check for specific types of errors. This chapter won't get into that. All you need to know is that if you have a `catch` block, Swift will automatically create a local variable named `error`. It will contain the error thrown by the code within the `do` block. You can refer to that `error` variable within the `catch` block, which is handy for printing out a descriptive error message that indicates the error's source.

You'll notice that Xcode is showing one of its cryptic error messages: "Referencing instance method 'encode' on 'Array' requires that 'ChecklistItem' conform to 'Encodable'". This is because any object encoded or decoded by a `PropertyListEncoder` — or for that matter, any of the other encoders and decoders compatible with the `Codable` protocol — must support the `Codable` protocol.

A closer look at the `Codable` protocol

Swift arrays — as well as most other standard Swift objects and data types — already conform to the `Codable` protocol. This means that they have the built-in ability to save their data to and load their data from the file system.

The `items` property of `Checklist` is an array, so it conforms to `Codable`. However, the objects contained within the `items` array must also support `Codable` in order for the array to be serialized. The question becomes: Is your `ChecklistItem` class `Codable` compliant? Apparently not...

Note: Sometimes when working with code dealing with `Codable` support, you'll see error messages or references to `Encodable` or `Decodable` protocols. So, it might be good to know that `Codable` is actually a protocol which combines these two other protocols, `Encodable` and `Decodable` — one for each side of the serialization process.

► Switch to `ChecklistItem.swift` and modify the `struct` line as follows:

```
struct ChecklistItem: Identifiable, Codable {
```

In the above code, you’re telling Swift that `ChecklistItem` is not just a kind of `Identifiable`, but also a kind of `Codable`. This tells the compiler that `ChecklistItem` conforms to the `Codable` protocol. That’s all you need to do!

“Now, hold on,” you might say. “In *Bullseye*, I had to write additional code to support the `ViewModifier` protocol. How come I don’t have to do that here?”

In case you’ve forgotten, you used the `ViewModifier` protocol to style different parts of *Bullseye*’s user interface in Chapter 7, “The New Look.” To make use of it, you had to add a `body` property to objects that adopted it.

That’s because protocols can have default implementations, which means that objects that adopt them don’t need any additional code. It’s often useful for a protocol to have a default implementation that provides functionality that makes things easier or covers a lot of standard scenarios.

In this case, all of the properties of `ChecklistItem` — `id`, `name` and `isChecked` — are standard Swift types that conform to the `Codable` protocol. As a result, Swift already knows how to encode and decode them. So, you can simply piggyback on existing functionality without having to write any code of your own to implement encoding or decoding in `ChecklistItem`. Handy, eh?

Putting `saveListItems()` to use

Now that you have the `saveListItems()` method, you need to be able to call it from the places in the code where the user can modify the list of items.

Challenge: Before continuing, ask yourself: Where in the source code would you call `saveListItems()`?

You should call `saveListItems()` when any of the following happens:

- The user adds a new item to the checklist.
- The user changes an existing item in the checklist, either by changing its name or its checked status.
- The user deletes an item from the checklist.
- The user moves an item to a different location within the checklist.

The code for deleting and moving checklist items is in `Checklist`, so you’ll add calls to `saveListItems()` to handle those cases first.



- Open **Checklist.swift** and add calls to `saveListItems()` to the end of `deleteListItem(whichElement:)` and `moveListItem(whichElement:)`. They should end up looking like this:

```
func deleteListItem(whichElement: IndexSet) {
    items.remove(atOffsets: whichElement)
    printChecklistContents()
    saveListItems()
}

func moveListItem(whichElement: IndexSet, destination: Int) {
    items.move(fromOffsets: whichElement, toOffset: destination)
    printChecklistContents()
    saveListItems()
}
```

The code for creating a new checklist item is in `NewChecklistItemView`, so another call to `saveListItems()` should go there.

- Switch to **NewChecklistItemView.swift**. In body, add a call to `saveListItems()` in the Button's `action:` parameter so that the lines defining the button look like this:

```
Button(action: {
    var newItem = ChecklistItem(name: self.newItemName)
    self.checklist.items.append(newItem)
    self.checklist.printChecklistContents()
    self.checklist.saveListItems()
    self.presentationMode.wrappedValue.dismiss()
}) {
```

There's one more situation where you need to call `saveListItems()`: When the user makes a change to a checklist item. This happens in `EditChecklistItemView`.

- Open **EditChecklistItemView.swift** and look at the code that defines the view:

```
struct EditChecklistItemView: View {
    @Binding var checklistItem: ChecklistItem

    var body: some View {
        Form {
            TextField("Name", text: $checklistItem.name)
            Toggle("Completed", isOn: $checklistItem.isChecked)
        }
    }
}
```

There's a problem here: Unlike `NewChecklistItemView`, `EditChecklistItemView` doesn't have a property that contains a reference to the checklist. Without access to the checklist object, there's no way to call its `saveListItems()`.

This is a good time to step back and take a look at what happens the user edits a checklist item:

- The user taps on a checklist item.
- The app responds by displaying the “Edit checklist item” screen.
- The user has the option of changing the item’s name, checked status or both. The user can also opt to *not* change anything.
- The user returns to the checklist screen by pressing the **Checklist** button located in the upper-left corner of the “Edit checklist item” screen.
- The app responds by displaying the checklist screen.

That last event in the list — “The app responds by displaying the checklist screen” — always happens after the user closes the “Edit checklist item” screen, whether they have made any changes to the item or not. The checklist screen has access to the checklist object, which means that the call to `saveListItems()` should be made when the “Edit checklist item” screen closes and the checklist screen appears.

Take a look at the checklist screen’s code.

► Switch to `ChecklistView.swift`. Look near the end of `ChecklistView`’s body and you’ll see `onAppear()`:

```
.onAppear() {
    self.checklist.printChecklistContents()
}
```

Add a call to `saveListItems()` to `onAppear()`, which causes the app to save the checklist any time the checklist displays:

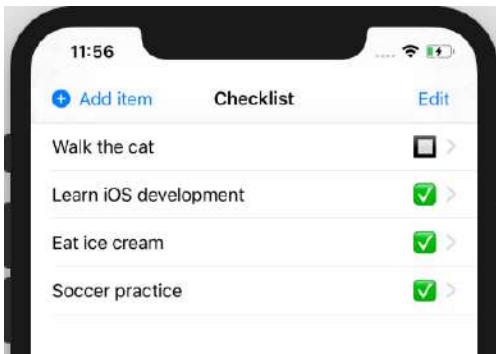
```
.onAppear() {
    self.checklist.printChecklistContents()
    self.checklist.saveListItems()
}
```

This takes care of all the cases where the checklist or one of its items changes. Next, you’ll confirm that our calls to `saveListItems()` works.



Verifying the saved file

- Run the app now and do something that results in a save. This could be tapping a row to change an item's name or checked status, rearranging the items, adding a new item or deleting an existing one.



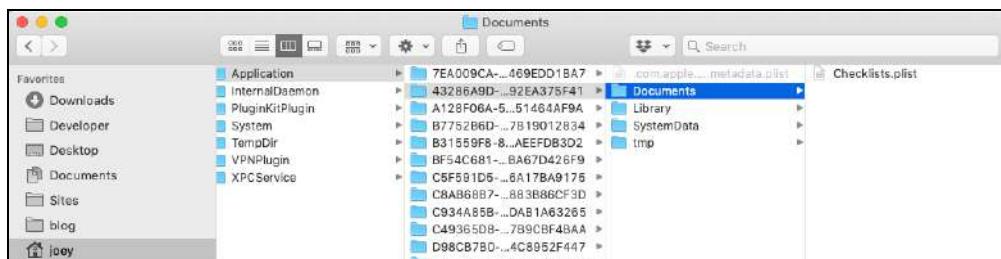
The updated checklist

Remember to run the app in the simulator. You'll need access to the simulator's file system, which is easy to view on your Mac.

In my case, I made the following changes to the checklist items:

- Edited the name of the “Walk the dog” item, changing it to “Walk the cat.”
- Deleted the “Brush my teeth” item.
- Checked the “Soccer practice” item.
- Rearranged the items so that “Eat ice cream” comes before “Soccer practice.”

- Go to the Finder window that has the app’s **Documents** directory open:



The Documents directory now contains a Checklist.plist file

There's now a **Checklist.plist** file in the Documents folder, which contains the items from the list.

You can look inside this file if you want. What you see depends on which app you use to open it. If you use a general-purpose text editor, the file's contents won't make much sense. Here's what it looks like in *Visual Studio Code* on my computer:



```
bplist00RidNameYisChecked_ $FA9D1B56-7BB9-4233-83AC-8761C9F7A19E\Wa
lk the cat#
$_3C397BFB-575E-4B84-912A-379F5CB1DD98_Learn iOS development  0
$_5279F8E8-F25D-46FB-85DD-B5DD7FF0B69F\Eat ice cream
@_23595DAF-F0BC-468B-AED2-771534ADB308_Soccer practice  0
00000000000000000000000000000000
00000000000000000000000000000000
```

The plist file, as seen in Visual Studio Code

Even though it's XML, the .plist file is stored in a binary format, which is why it looks so garbled in *Visual Studio Code*.

Some text editors, especially those designed specifically for macOS, support this file format and can read it as if it were text. *TextWrangler* is a good option, and it's a free download from the Mac App Store. Here's what **Checklist.plist** looks like when you view it in one of these editors:

```
<?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>id</key>
        <string>FA9D1B56-7BB9-4233-83AC-8761C9F7A19E</string>
        <key>isChecked</key>
        <false/>
        <key>name</key>
        <string>Walk the cat</string>
    </dict>
    <dict>
        <key>id</key>
        <string>3C397BFB-575E-4B84-912A-379F5CB1DD98</string>
        <key>isChecked</key>
        <true/>
        <key>name</key>
        <string>Learn iOS development</string>
    </dict>
    <dict>
        <key>id</key>
        <string>5279F8E8-F25D-46FB-85DD-B5DD7FF0B69F</string>
        <key>isChecked</key>
        <true/>
        <key>name</key>
        <string>Eat ice cream</string>
    </dict>
</array>
```



```
</dict>
<dict>
    <key>id</key>
    <string>23395DAA-F08C-468B-AED2-771534ADB308</string>
    <key>isChecked</key>
    <true/>
    <key>name</key>
    <string>Soccer practice</string>
</dict>
</array>
</plist>
```

Checklist.plist is considerably readable in this form. You can even see how it corresponds to the items in the app's checklist.

Naturally, you can also open the plist file with Xcode, which displays the contents of plist files in an even more user-friendly way.

- Right-click the **Checklist.plist** file and choose **Open With ▶ Xcode**.

You'll see the following window appear:

The screenshot shows the Xcode interface with the Checklist.plist file open. The table view displays the following data:

Key	Type	Value
Root	Array	(4 items)
Item 0	Dictionary	(3 items)
Item 1	Dictionary	(3 items)
Item 2	Dictionary	(3 items)
Item 3	Dictionary	(3 items)

The plist file with the items closed, as seen in Xcode

To see the contents of the checklist items, expand each item by clicking on its disclosure triangle. You'll see each item's ID value, name and checked status:

The screenshot shows the Xcode interface with the Checklist.plist file open. The table view displays the following expanded data:

Key	Type	Value
Root	Array	(4 items)
Item 0	Dictionary	(3 items)
id	String	FA9D1B56-7BB9-4233-83AC-B761C9F7A19E
name	String	Walk the cat
isChecked	Boolean	NO
Item 1	Dictionary	(3 items)
id	String	9C3978FB-575E-4B84-912A-379F5CB1DD98
name	String	Learn iOS development
isChecked	Boolean	YES
Item 2	Dictionary	(3 items)
id	String	E279F8E8-F25D-40FB-85DD-B5DD7FF0B69F
name	String	Eat ice cream
isChecked	Boolean	YES
Item 3	Dictionary	(3 items)
id	String	23395DAA-F08C-468B-AED2-771534ADB308
name	String	Soccer practice
isChecked	Boolean	YES

The plist file with the items closed, as seen in Xcode

You'll also see that the contents of the plist file correspond to the current state of the checklist in the app. This confirms that the checklist data is saving properly.

It's now time to take care of loading the data.

Loading the file

Saving is all well and good, but it's only half of what the app needs. It also needs to load the data from the **Checklist.plist** file.

Fortunately, loading saved data is very straightforward. You're going to do the same thing you just did for encoding the items array — but in *reverse*.

Reading data from a file

- Open **Checklist.swift** and add the following new method, just after `saveListItems()`:

```
func loadListItems() {  
    // 1  
    let path = dataFilePath()  
    // 2  
    if let data = try? Data(contentsOf: path) {  
        // 3  
        let decoder = PropertyListDecoder()  
        do {  
            // 4  
            items = try decoder.decode([ChecklistItem].self,  
                                         from: data)  
            // 5  
        } catch {  
            print("Error decoding item array: \  
                  (error.localizedDescription)")  
        }  
    }  
}
```

As you did with `saveListItems()`, go through the commented lines in `loadListItems()` step-by-step:

1. First, you store the results of `dataFilePath()` — the path to the **Checklist.plist** file — in a temporary constant named `path`.
2. The method tries to load the contents of **Checklist.plist** into a new `Data` object. The `try?` command attempts to create the `Data` object, but returns `nil` — Swift's

way of saying “no result” — if it fails. That’s why you put it in an `if let` statement.

Why would it fail? If there is no `Checklist.plist` file, then there are obviously no `ChecklistItem` objects to load. This happens when the app starts up for the very first time. In that case, you’ll skip the rest of this method.

Notice that this is another way to use the `try` statement. Instead of enclosing the `try` statement within a `do` block, as you did previously, you have a `try?` statement that indicates that the `try` could fail. If it does, it will return `nil`. Whether you use the `do` block approach or this one is completely up to you.

3. When the app does find a `Checklist.plist` file, the method creates an instance of `PropertyListDecoder`.
4. The method loads the saved data back into `items` using the decoder’s `decode` method. The only item of interest here is the first parameter passed to `decode`. The decoder needs to know what type of data the result of the `decode` operation will be. You let it know that it will be an array of `ChecklistItem` objects.

This populates the array with exact copies of the `ChecklistItem` objects that you froze into the `Checklist.plist` file.

5. This is the start of the `catch` block, which contains the code that executes if any line of code in the `do` block throws an error.

As with `saveListItems()`, if this were an app that would go into the App Store, this code might do all sorts of things to deal with cases where decoding the data or reading it from the device’s file system fails. Once again, you’ll simply print out an error message to Xcode’s console.

Putting `loadListItems()` to use

You now have the `loadListItems()` method, which restores the app’s data from `Checklist.plist`.

Challenge: Before continuing, ask yourself: Where in the source code would you call the `saveListItems()` method?

There’s only one time when you need to load the saved checklist data: when the app launches, or more specifically, at the moment when the `Checklist` instance is created. This is where `Checklist`’s `init()` method — its initializer — comes in



handy. It's called at that very moment, making it the perfect place to put the call to `loadListItems()`.

► Open **Checklist.swift** and add a call to `loadListItems()` at the end of its `init()` method. `init()` should look like this:

```
init() {
    print("Documents directory is: \(documentsDirectory())")
    print("Data file path is: \(dataFilePath())")
    loadListItems()
}
```

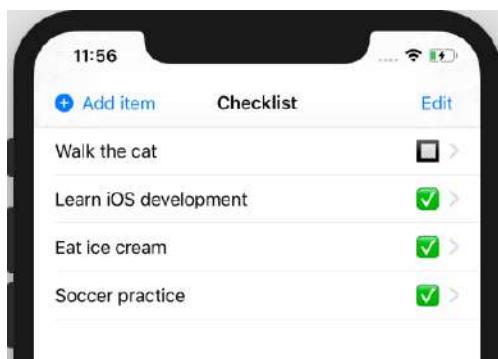
It's time to test `loadListItems()` by seeing if the app "remembers" changes to the checklist after the user closes and reopens it.

► Run the app and make some changes to the checklist.

In my case, I made the same changes that I made when testing `saveListItems()`:

- Edited the name of the "Walk the dog" item, changing it to "Walk the cat."
- Deleted the "Brush my teeth" item.
- Checked the "Soccer practice" item.
- Rearranged the items so that "Eat ice cream" comes before "Soccer practice."

After these changes, my checklist looked like this:

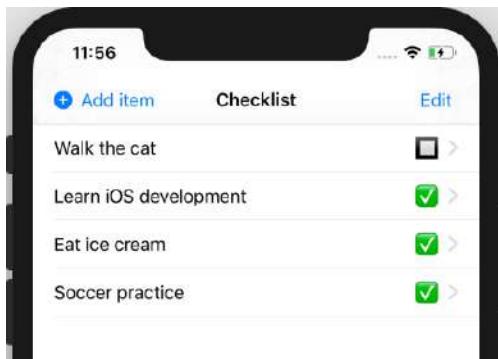


The updated checklist

► Close the app. You can do this by clicking the **Stop** button in Xcode or by terminating the app in the simulator.

► Restart the app. Instead of the five default checklist items, you should see the checklist as it was when you quit the app.

In my case, the newly-launched app showed this checklist:



The updated checklist

Before you make the final change to the app, take a closer look at the `init()` method where you placed `loadListItems()`.

A closer look at initializers

Methods named `init` are special in Swift. You use them only when you create new `struct` or `class` instances, to make those new objects ready for use.

Think of it like buying new clothes. The clothes are in your possession (the memory for the object is allocated) but they're still in the bag. You need to put the new clothes on (initialization) before you're ready to go out and party.

When you write the following to create a new object:

```
let checklist = Checklist()
```

Swift first allocates a chunk of memory big enough to hold the new object and then calls `Checklist`'s `init()` method with no parameters.

Every object blueprint has a built-in `init()` method with no parameters. If you don't write your own object blueprint, Swift simply uses the built-in one, which just creates a new instance.

It's pretty common for objects to have more than one `init` method. Which one the app uses depends on the circumstances.

Consider ChecklistItem. Here's its code:

```
struct ChecklistItem: Identifiable, Codable {  
    let id = UUID()  
    var name: String  
    var isChecked: Bool = false  
}
```

It has two properties that can be set: name and isChecked. name doesn't have a value assigned to it, which means that it must be assigned a value when the ChecklistItem instance is created. isChecked can also be assigned a value when the instance is created, but it has a default value of false. This means that setting its value at instance creation is optional.

As a result, ChecklistItem has two init() methods:

- ChecklistItem(name:), which you call when you want to create a ChecklistItem instance and specify just its name. Its isChecked property will take the default value of false.
- ChecklistItem(name:isChecked), which you call when you want to create a ChecklistItem instance and specify both its name and its checked status.

Now, consider the case of Checklist, where you defined your own init() method:

```
init() {  
    print("Documents directory is: \(documentsDirectory())")  
    print("Data file path is: \(dataFilePath())")  
    loadListItems()  
}
```

By writing this method, you're **overriding** the built-in init(), which means that you're replacing the default initializer with one of your own. You did this because you wanted to do more than simply creating a Checklist instance. You also wanted to perform some tasks as the instance was being created.

Note that unlike other methods, init does not have the func keyword.

Sometimes, you'll see it written as override init or required init?. That's necessary when you're adding the init method to an object that's a subclass of some other object. Much more about that later.

You use the version with the question mark when init? can potentially fail and return a nil value instead of a real object. Decoding an object can fail if not enough information is present in the plist file.



Inside the `init` method, you first need to make sure that all your instance variables and constants have a value. Recall that in Swift, all variables must always have a value, except for optionals.

When you declare an instance variable, you can give it an initial value (or initialize it), like so:

```
var checked = false
```

It's also possible to write just the variable name and its type (or declare the variable), but not give the variable a value yet:

```
var checked: Bool
```

In the latter case, you have to give this variable a value in your `init` method:

```
init() {
    checked = false
}
```

You must use one of these approaches. If you don't give the variable a value at all, Swift considers this an error. The only exception is optionals, which don't need to have a value. If they don't, they are `nil`. You'll learn more about optionals later in this book.

Swift's rules for initializers can be a bit complicated but fortunately, the compiler will remind you if you forget to provide an `init` method.

Removing the default checklist items

Since *Checklist* now remembers its checklist items between sessions, it no longer needs its default checklist items.

You want the app to behave this way:

- If the app has been launched before, it should contain the same checklist items from the previous session when it launches again.
- If the user has just installed the app and has never used it, the app should display an empty checklist at launch.

This change in behavior is easy to accomplish.

- Switch to **Checklist.swift** and change the definition of the `items` property from this:

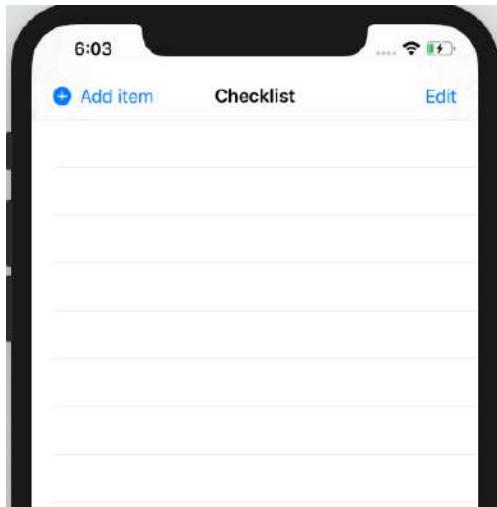
```
@Published var items = [  
    ChecklistItem(name: "Walk the dog", isChecked: false),  
    ChecklistItem(name: "Brush my teeth", isChecked: false),  
    ChecklistItem(name: "Learn iOS development", isChecked: true),  
    ChecklistItem(name: "Soccer practice", isChecked: false),  
    ChecklistItem(name: "Eat ice cream", isChecked: true),  
]
```

To this:

```
@Published var items: [ChecklistItem] = []
```

Now, test the effect of this change. You'll need to run the app as if it were freshly installed and never used, which requires getting rid of the existing **Checklist.plist** file.

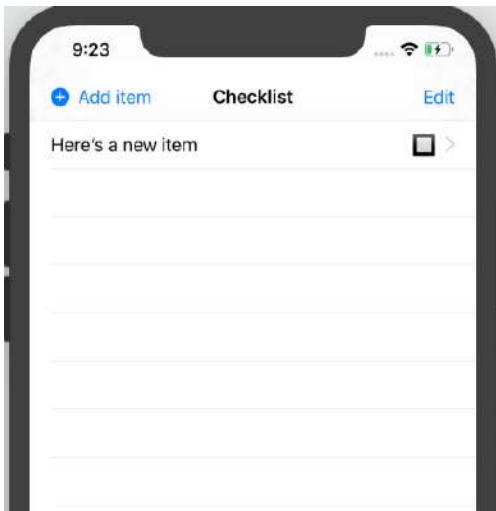
- Go to the Finder window displaying **Checklist.plist** file. Delete the file.
► Run the app. You should now see an empty checklist, ready for you to fill it:



An empty checklist

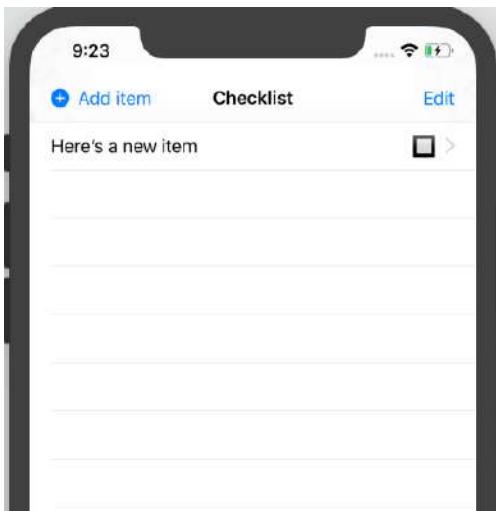
Next, make sure that everything in the app works properly.

- Tap the **Add item** button, enter a name for the new item, and tap the **Add new item** button. You'll return to the checklist, which will display the newly-created item:



The checklist with a newly-created item

- Close the app. Once again, you can do this by clicking the **Stop** button in Xcode or by terminating the app in the simulator.
- Run the app again. You'll see that the checklist is the same as when you closed the app:



The saved checklist reappears when you restart the app

With this final change, *Checklist* is complete! You have a fully-functional checklist app that remembers its contents between sessions.

This is a good time to go back and repeat those parts you're still a bit fuzzy about. Don't rush through these chapters — there are no prizes for finishing first. Rather than going fast, take your time to truly understand what you're doing.

As always, feel free to change the app and experiment. Here at iOS Apprentice Academy, we not only allow breaking things — we encourage it! You can find the project files for the app up to this point under **14 – Saving and Loading** in the **Source Code** folder.

Next steps

iOS should start to make sense by now. You've written an entire app from scratch! Already, you've touched on several advanced topics, and hopefully you were able to follow along. Kudos for sticking with it until the end!

It's okay if you're still a bit fuzzy on the details. Sleep on it for a bit and keep tinkering with the code. Programming requires its own way of thinking, and you won't learn that overnight. Don't be afraid to create this app again from the start — it will make more sense the second time around!

The first two sections of this book focused mainly on the SwiftUI framework, which is the newest way of building iOS apps. The next sections of this book will introduce you to UIKit, the framework that iOS developers have been using since the beginning of the iPhone, and which many will continue to use for some time.

The next section will also take a step back and cover more details about the Swift language. Pay particular attention, as it's helpful not just for understanding the code in your upcoming projects...it'll also give you a better understanding of the code in the projects you've already completed.

Section 3: Getting Started with UIKit

In this section, you'll learn about UIKit, which is an alternative way to build the UI of your app. UIKit has been around since the first iOS and is currently powering all of the existing iOS apps in the App Store. UIKit is the foundation on which most of SwiftUI is built upon.

You're about to create Bullseye again but this time using UIKit so you can see the differences between using SwiftUI and UIKit. We feel like SwiftUI is the future of iOS development but we truly think a programmer learning iOS you should have a good grasp of both.

You'll start by creating a basic view to understand how UIKit works, how it places UI elements on the screen and how to interact with them. You'll also read about the most common design pattern used when building apps using UIKit. You'll then go on to create Bullseye!

This section contains the following chapters:

15. UIKit & The One-Button App: You've built two apps using SwiftUI, yay!. In this chapter, you will start building Bullseye using UIKit, Apple's existing way of building UI for iOS apps.

16. Sliders & Labels: Congratulations, you have a UIButton on the screen. It's time to start adding the UISlider which will be fundamental to the game.

17. Outlets: You'll be well on your way of noticing the differences between building an app using SwiftUI and now UIKit. In this chapter you will deal with random numbers, adding rounds to the game and calculating the points scored.



- 18. Polish:** In this chapter, we will add some UIKit polish to the app and show an alert to the user.
- 19. The New Look:** Bullseye is looking great! The gameplay is now complete but it's time to make it look pretty. In this chapter, we will add some graphics and create an about screen to display the rules of the game.
- 20. TableView:** TableViews are fundamental in the UIKit toolbox. In this chapter, you will learn about data sources, delegates, and general TableView best practices. Be sure to take this knowledge in your future iOS career.
- 21. The Data Model:** A TableView is no good without real data. It's time to create the data model that will hold the high score data.
- 22. Navigation Controllers:** UINavigationControllerControllers are super important in an iOS app. In this chapter, you will learn about adding a NavigationController, which will help present the high scores view. You will also add the functionality to delete rows from the TableView.
- 23. Edit High Score Screen:** Now that you have the navigation flow sorted, it's time to implement the edit high score functionality. In this chapter, we will add a static TableViewCell and read the contents from the UITextField.
- 24. Delegates & Protocols:** You now have an edit high score screen but how do we get this data back to the high score screen? In this chapter, you will learn about Delegates and how best to use them.
- 25. The Final App:** Phew! You have successfully created Bullseye using UIKit. In this final chapter of this section you will learn about supporting different device sizes and add some beautiful animations.

Chapter 15: UIKit and The One-Button App

Eli Ganim

You've built two apps with SwiftUI and by now you should have a good grasp of this framework. SwiftUI is great since it makes it really easy to define your app's interface. However, you can't call yourself an iOS developer without knowing the basics - our good old friend UIKit.

UIKit is an alternative way to build the UI of your app. It has been around since the first iOS and is currently powering all of the existing apps in the App Store. Actually, it's the foundation on which SwiftUI is built.

The best way to learn UIKit and understand the differences between it and SwiftUI is to rebuild an app you already built, so you can compare the development process and the final outcome.

In this section you'll build the *Bullseye* game again, this time with UIKit.

This chapter covers the following:

- **The Bullseye game:** A reminder of the game you already built in Section 1.
- **The one-button app:** Creating a simple one-button app in which the button can take an action based on a tap on the button.
- **The anatomy of an app:** A brief explanation as to the inner-workings of an app.



The Bullseye game

As a reminder, this is what the *Bullseye* game will look like when you’re finished:



The finished Bullseye game

As you probably remember, the objective of the game is to put the bullseye, which is on a slider that goes from 1 to 100, as close to a randomly chosen target value as you can. In the screenshot above, the aim is to put the bullseye at 84. Because you can’t see the current value of the slider, you’ll have to “eyeball” it.

When you’re confident of your estimate, you press the “Hit Me!” button and a pop-up will tell you what your score is. The closer to the target value you are, the more points you score. After you dismiss the alert pop-up, a new round begins with a new random target. The game repeats until the player presses the “Start Over” button, which resets the score to 0.

Making a programming to-do list

Just like in Section 1, you'll follow this to-do list to get the job done:

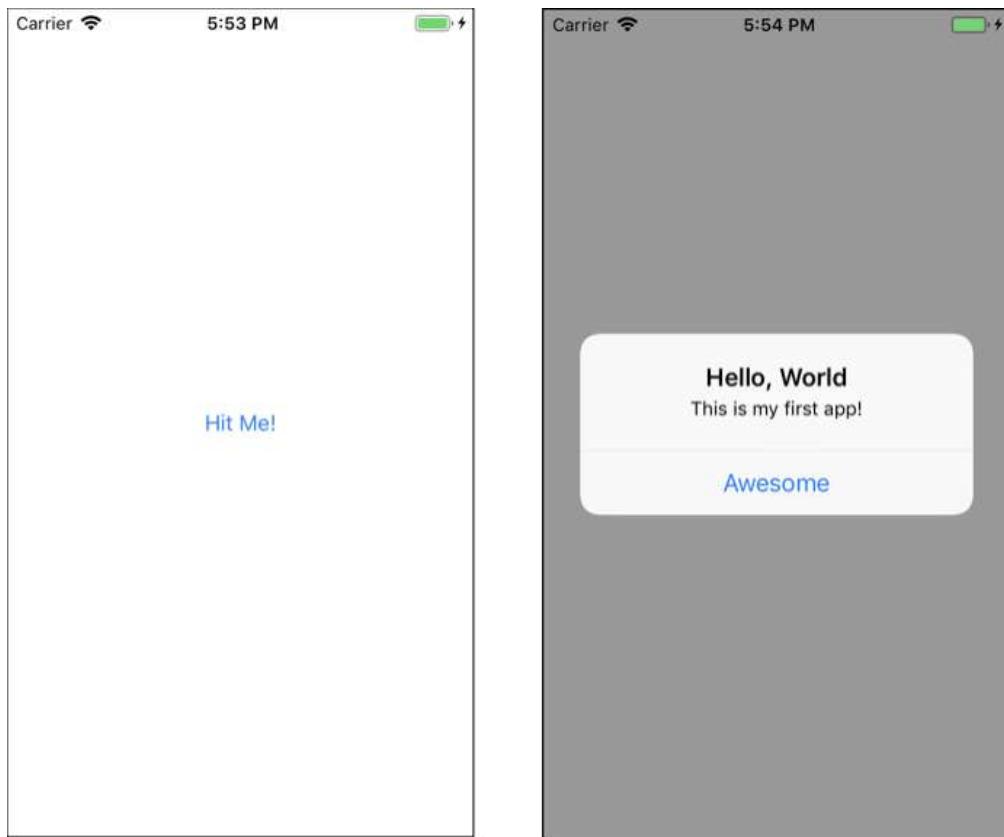
- Put a button on the screen and label it “Hit Me!”
- When the player presses the Hit Me! button, the app has to show an alert pop-up to inform the player how well he or she did. Somehow, you have to calculate the score and put that into this alert.
- Put text on the screen, such as the “Score:” and “Round:” labels. Some of this text changes over time; for example, the score, which increases when the player scores points.
- Put a slider on the screen with a range between the values 1 and 100.
- Read the value of the slider after the user presses the Hit Me! button.
- Generate a random number at the start of each round and display it on the screen. This is the target value.
- Compare the value of the slider to that random number and calculate a score based on how far off the player is. You show this score in the alert pop-up.
- Put the Start Over button on the screen. Make it reset the score and put the player back to the first round.
- Put the app in landscape orientation.
- Make it look pretty.

The one-button app

Start at the top of the list and make an extremely simple first version of the game that just displays a single button. When you press the button, the app pops up an alert message. That's all you are going to do for now. Once you have this working, you can build the rest of the game on this foundation.



The app will look like this:



The app contains a single button (left) that shows an alert when pressed (right)

Creating a new project

- Launch Xcode.
- Choose **Create a new Xcode project**.
- Select **Single View Application** and press **Next**.
- Fill out these options as follows:
 - Product Name: **Bullseye**.
 - Team: **None** or your team.
 - Organization Name: Your name or the name of your company.

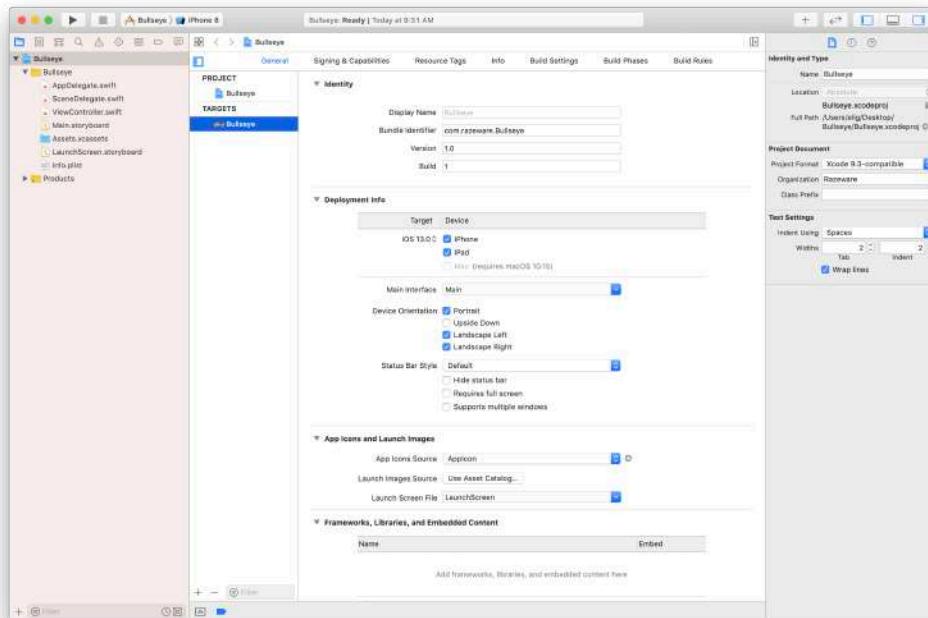
- Organization Identifier: Your own identifier in reverse domain notation.
- Language: **Swift**
- Make sure the three options at the bottom — Use Core Data, Include Unit Tests, and Include UI Tests — are *not* selected. You won't use those in this project.
- Most importantly: Uncheck the **Use SwiftUI** checkbox! This is how you tell Xcode you're going to create a UIKit based app.

► Press **Next**.

You can ignore the “Create Git repository on My Mac” checkbox for now. You’ll learn about the Git version control system later on.

► Press **Create** to finish.

When it is done, the screen should look something like this:

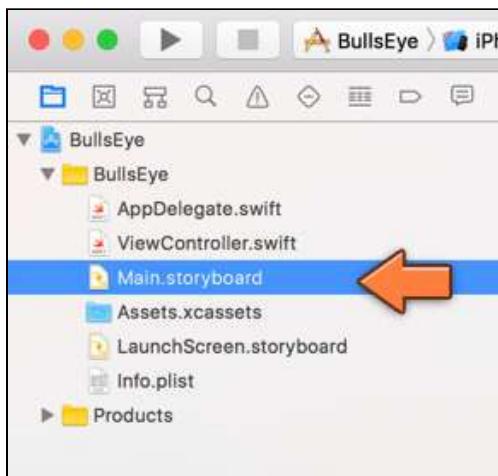


The main Xcode window at the start of your project

This is very similar to what you saw when you created your first SwiftUI app, but there are some minor differences which you'll learn about later on.

Adding a button

- In the **Project navigator**, find the item named **Main.storyboard** and click it once to select it:



The Project navigator lists the files in the project

Like a superhero changing his or her clothes in a phone booth, the main editing pane now transforms into the **Interface Builder**. This tool lets you drag-and-drop user interface components such as buttons to create the UI of your app.

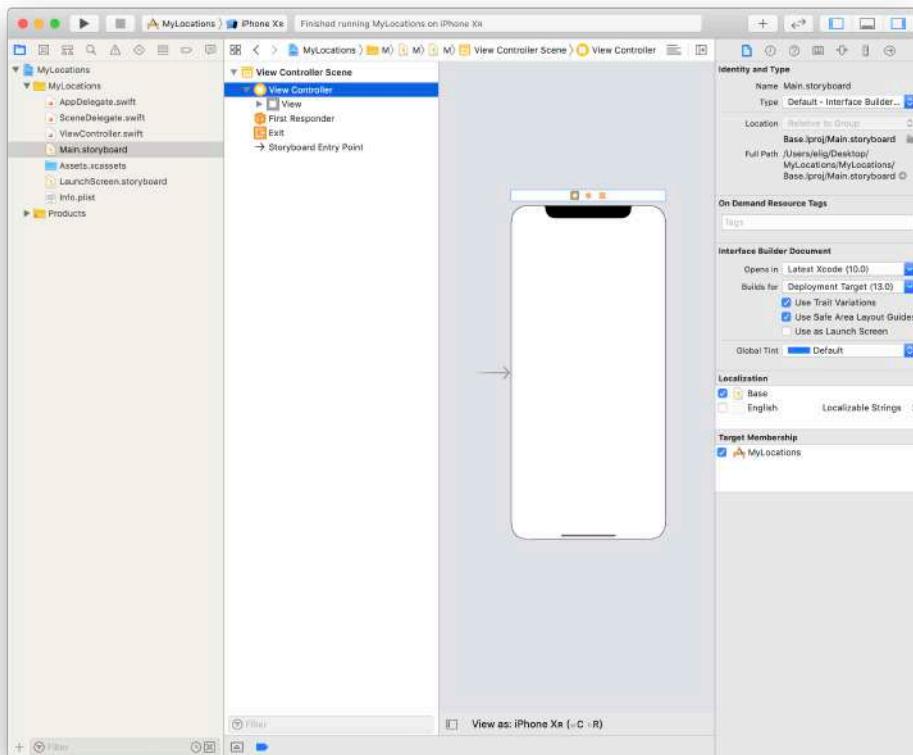
- If it's not already blue, click the **Hide or Show the Inspectors** button in Xcode's toolbar.



Click this button to show the Utilities pane

These toolbar buttons in the top-right corner change the appearance of Xcode. This one in particular opens a new pane on the right side of the Xcode window.

Your Xcode should now look something like this:



Editing Main.storyboard in Interface Builder

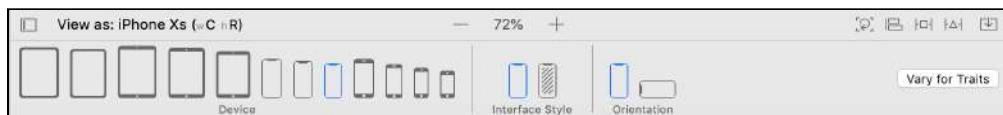
This is the *storyboard* for your app. The storyboard contains the designs for all of your app’s screens and shows the navigation flow in your app from one screen to another.

Currently, the storyboard contains just a single screen or *scene*, represented by a rectangle in the middle of the Interface Builder canvas.

Note: If you don’t see the rectangle labeled “View Controller” but only an empty canvas, then use your mouse or trackpad to scroll the storyboard around a bit. Trust me, it’s in there somewhere! Also make sure your Xcode window is large enough. Interface Builder takes up a lot of space...

The scene currently is probably set to the size of an iPhone XR. To keep things simple, you will first design the app for the iPhone 8, which has a smaller screen. Later, you'll also make the app fit on the larger iPhone models.

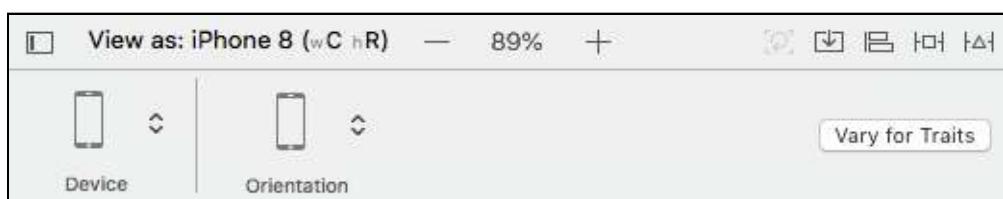
- At the bottom of the Interface Builder window, click **View as: iPhone XR** to open up the following panel:



Choosing the device type

Select the **iPhone 8**, thus resizing the preview UI you see in Interface Builder to be set to that of an iPhone 8. You'll notice that the scene's rectangle now becomes a bit smaller.

Do note that depending on the size of your Xcode window, the above panel might also look something like this:



Choosing the device type - compact view

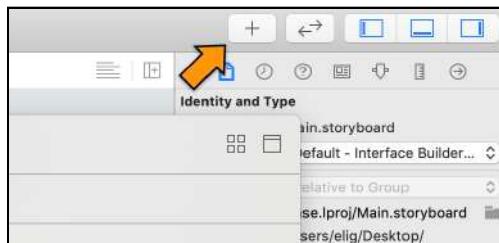
If you get this screen, just select the **iPhone 8** from the list of choices you get when you click on **Device**.

- In the Xcode toolbar, make sure it says **Bullseye > iPhone 8** (next to the Stop button). If it doesn't, then click it and pick iPhone 8 from the list.

Now, when you run the app, it will run on the iPhone 8 Simulator (try it out!).

Back to the storyboard.

- The first button on the top right toolbar shows the **Library** panel when you click it:

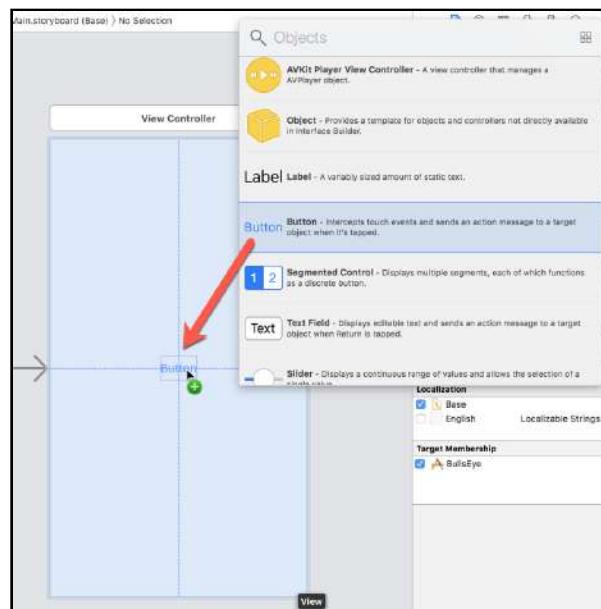


The Object Library

Scroll through the items in the Object Library list until you see **Button**.

Alternatively, you can type the word "button" in to the search/filter box at the top.

- Click on **Button** and drag it onto the working area, on top of the scene's rectangle.



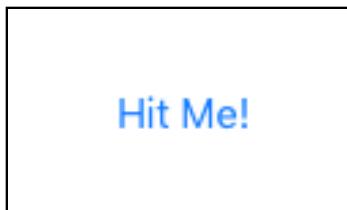
Dragging the button on top of the scene

That's how easy it is to add most new UI items — just drag and drop. You'll do a lot of this, so take some time to get familiar with the process.

Drag and drop a few other controls, such as labels, sliders, and switches, just to get the hang of it. Once you are done, delete everything except for the first button you added.

This should give you some idea of the UI controls that are available in iOS. Notice that the Interface Builder helps you to lay out your controls by snapping them to the edges of the view and to other objects. It's a very handy tool!

- Double-click the button to edit its title. Call it Hit Me!



The button with the new title

It's possible that your button might have a border around it:



The button with a bounds rectangle

This border is not part of the button, it's just there to show you how large the button is. You can turn these borders on or off using the **Editor** ▶ **Canvas** ▶ **Show Bounds Rectangles** menu option.

When you're done playing with Interface Builder, press the Run button from Xcode's toolbar. The app should now appear in the simulator, complete with your "Hit Me!" button. However, when you tap the button, it doesn't do anything yet. For that, you'll have to write some Swift code!

Using the source code editor

A button that doesn't do anything when tapped is of no use to anyone. So let's make it show an alert pop-up. In the finished game, the alert will display the player's score; for now, you will limit yourself to a simple text message (the traditional "Hello, World!").

- In the **Project navigator**, click on **ViewController.swift**.

The Interface Builder will disappear and the editor area now contains a bunch of brightly colored text. This is the Swift source code for your app.

- Add the following lines directly **above** the very last } bracket in the file:

```
@IBAction func showAlert() {  
}
```

The source code for **ViewController.swift** should now look like this:

```
import UIKit  
  
class ViewController: UIViewController {  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        // Do any additional setup after loading the view.  
    }  
  
    @IBAction func showAlert() {  
    }  
}
```

View controllers

You've edited the **Main.storyboard** file to build the user interface of the app. It's only a button on a white background, but a user interface nonetheless. You also added source code to **ViewController.swift**.

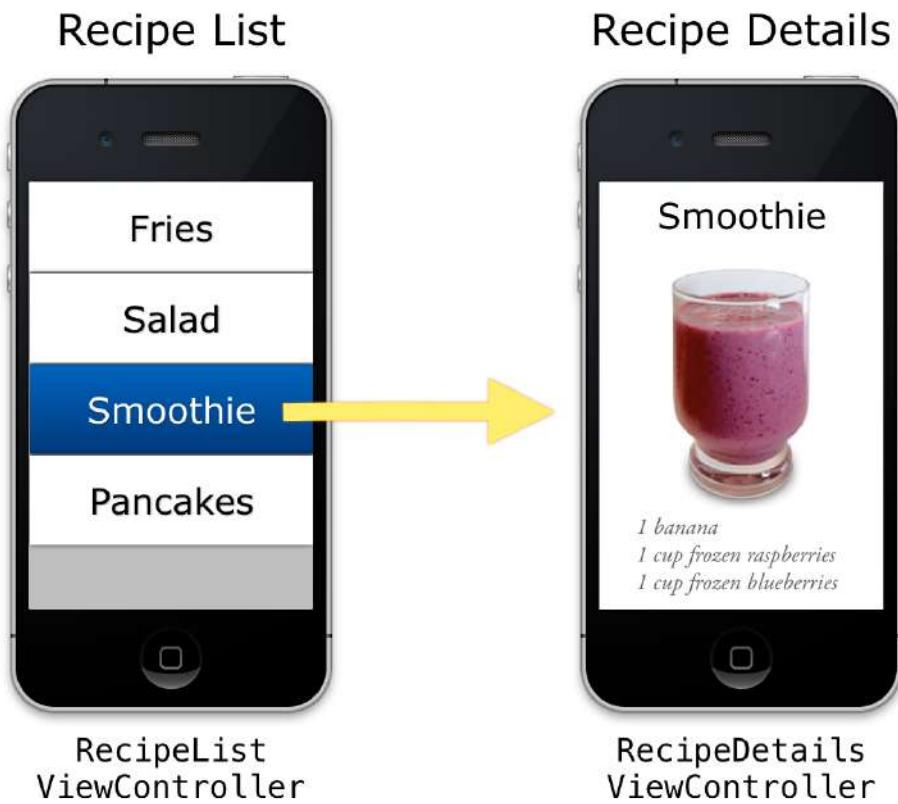
These two files — the storyboard and the Swift file — together form the design and implementation of a *view controller*. A lot of the work in building iOS apps is making view controllers. The job of a view controller, generally, is to manage a single screen in your app.

Take a simple cookbook app, for example. When you launch the cookbook app, its main screen lists the available recipes.

Tapping a recipe opens a new screen that shows the recipe in detail with an appetizing photo and cooking instructions.

Each of these screens is managed by a view controller.





The view controllers in a simple cookbook app

What these two screens do is very different. One is a list of several items; the other presents a detail view of a single item.

That's why you need two view controllers: One that knows how to deal with lists and another that can handle images and cooking instructions. One of the design principles of iOS is that each screen in your app gets its own view controller.

Currently, *Bullseye* has only one screen (the white one with the button) and thus only needs one view controller. That view controller is simply named “ViewController,” and the storyboard and Swift file work together to implement it.

If you are curious, you can check the connection between the screen and the code for it by switching to the Identity inspector on the right sidebar of Xcode in the storyboard view. The Class value shows the current class associated with the storyboard scene.

Simply put, the Main.storyboard file contains the design of the view controller's user interface, while ViewController.swift contains its functionality — the logic that makes the user interface work, written in the Swift language.

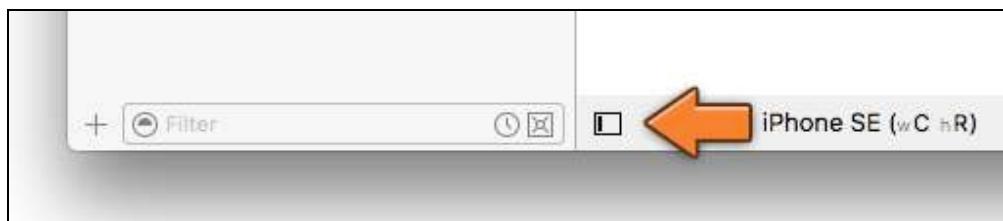
Because you used the Single View Application template, Xcode automatically created the view controller for you. Later, you will add a second screen to the game and you will create your own view controller for that.

Making connections

The two lines of source code you just added to ViewController.swift lets Interface Builder know that the controller has a “showAlert” action, which presumably will show an alert pop-up. You will now connect the button on the storyboard to that action in your source code.

- Click **Main.storyboard** to go back into Interface Builder.

In Interface Builder, there should be a second pane on the left, next to the navigator area, called the **Document Outline**, that lists all the items in your storyboard. If you do not see that pane, click the small toggle button in the bottom-left corner of the Interface Builder canvas to reveal it.

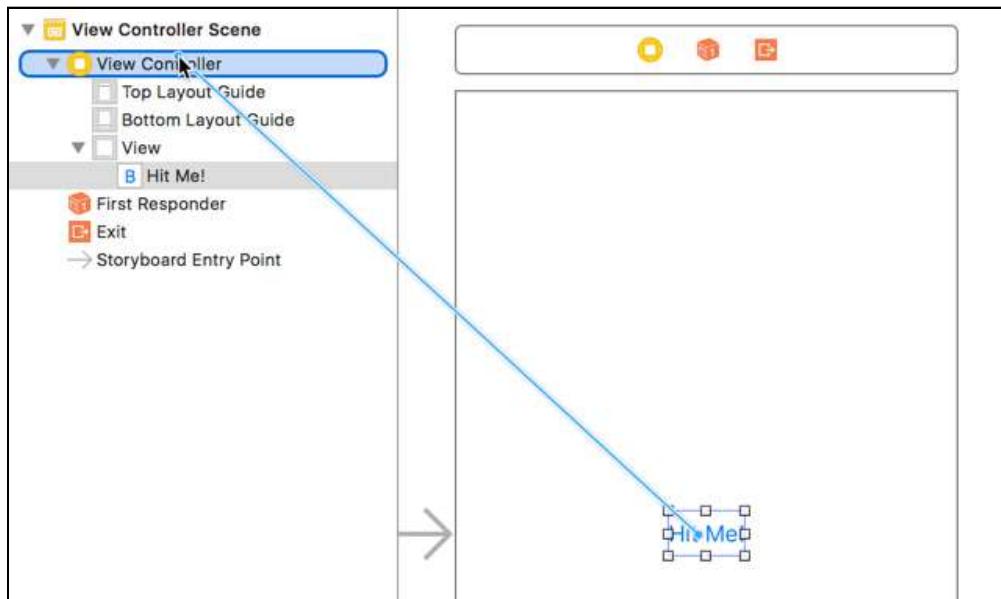


The button that shows the Document Outline pane

- Click the **Hit Me** button once to select it.

With the Hit Me button selected, hold down the **Control** key, click on the button and drag up to the **View Controller** item in the Document Outline. You should see a blue line going from the button up to View Controller.

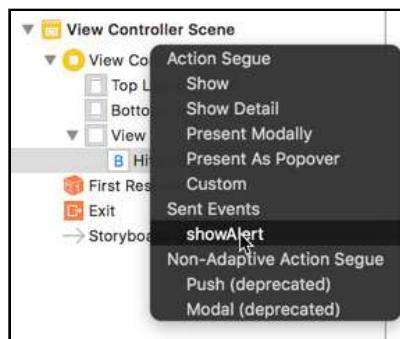
Please note that, instead of holding down Control, you can also right-click and drag, but don't let go of the mouse button before you start dragging.



Ctrl-drag from the button to View Controller

Once you're on View Controller, let go of the mouse button and a small menu will appear. It contains several sections: "Action Segue," "Sent Events," and "Non-Adaptive Action Segue," with one or more options below each. You're interested in the **showAlert** option under Sent Events.

The Sent Events section shows all possible actions in your source code that can be hooked up to your storyboard — **showAlert** is the name of the action that you added earlier in the source code of **ViewController.swift**.



The pop-up menu with the showAlert action

- Click on **showAlert** to select it. This instructs Interface Builder to make a connection between the button and the line `@IBAction func showAlert()`.

From now on, whenever the button is tapped the `showAlert` action will be performed. That is how you make buttons and other controls do things: You define an action in the view controller's Swift file and then you make the connection in Interface Builder. You can see that the connection was made by going to the **Connections inspector** in the Utilities pane on the right side of the Xcode window. You should have the button selected when you do this.

- Click the small arrow-shaped button at the top of the pane to switch to the Connections inspector:



The inspector shows the connections from the button to any other objects

In the Sent Events section, the “Touch Up Inside” event is now connected to the `showAlert` action. You should also see the connection in the Swift file.

- Select **ViewController.swift** to edit it.

Notice how, to the left of the line with `@IBAction func showAlert()`, there is a solid circle? Click on that circle to reveal what this action is connected to.



The screenshot shows a portion of the ViewController.swift file in Xcode. Line 39 contains the code `@IBAction func showAlert()`. To the left of this line, there is a small solid black circle. An orange arrow points from the left margin towards this circle. A callout bubble originates from the circle, pointing towards the storyboard icon and the button labeled "Hit Me!" in the storyboard preview area.

```
30
31 class ViewController: UIViewController {
32
33     override func viewDidLoad() {
34         super.viewDidLoad()
35         // Do any additional setup after loading the
36     }
37
38
39     @IBAction func showAlert() {
40
41 }
```

A solid circle means the action is connected to something

Acting on the button

You now have a screen with a button. The button is hooked up to an action named `showAlert` that will be performed when the user taps the button. Currently, however, the action is empty and nothing will happen (try it out by running the app again, if you like). You need to give the app more instructions.

- In `ViewController.swift`, modify `showAlert` to look like the following:

```
@IBAction func showAlert() {
    let alert = UIAlertController(title: "Hello, World",
                                 message: "This is my first
app!",
                                 preferredStyle: .alert)

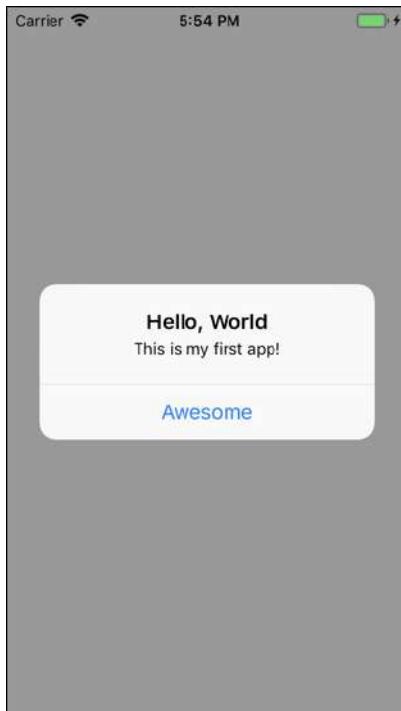
    let action = UIAlertAction(title: "Awesome", style: .default,
                           handler: nil)

    alert.addAction(action)

} present(alert, animated: true, completion: nil)
```

The code in `showAlert` creates an alert with a title “Hello, World,” a message that states, “This is my first app!” and a single button labeled “Awesome.”.

- Click the **Run** button from Xcode's toolbar. If you didn't make any typos, your app should launch in iOS Simulator and you should see the alert box when you tap the button.



The alert pop-up in action

Congratulations, you've just written your first UIKit app!

You can strike off the first two items from the to-do list already: Putting a button on the screen and showing an alert when the user taps the button.

The anatomy of an app

It might be good at this point to get some sense of what goes on behind the scenes of an app.

An app is essentially made up of **objects** that can send messages to each other. Many of the objects in your app are provided by iOS; for example, the button is a `UIButton` object and the alert pop-up is a `UIAlertController` object. Some objects you will have to program yourself, such as the view controller.

These objects communicate by passing messages to each other. For example, when the user taps the Hit Me button in the app, that `UIButton` object sends a message to your view controller. In turn, the view controller may message more objects.

On iOS, apps are *event-driven*, which means that the objects listen for certain events to occur and then process them.

As strange as it may sound, an app spends most of its time doing... absolutely nothing. It just sits there waiting for something to happen. When the user taps the screen, the app springs to action for a few milliseconds, and then it goes back to sleep again until the next event arrives.

Your part in this scheme is that you write the source code for the actions that will be performed when your objects receive the messages for such events.

In the app, the button's Touch Up Inside event is connected to the view controller's `showAlert` action. So when the button recognizes it has been tapped, it sends the `showAlert` message to your view controller.

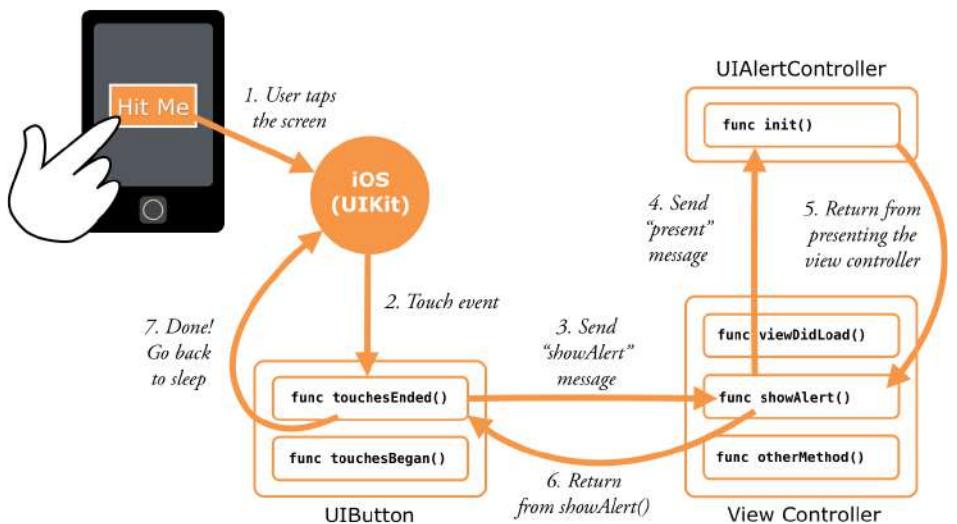
Inside `showAlert`, the view controller sends another message, `addAction`, to the `UIAlertController` object. And to show the alert, the view controller sends the `present` message.

Your whole app will be made up of objects that communicate in this fashion.

Maybe you have used PHP or Ruby scripts on your web site. This event-based model is different from how a PHP script works. The PHP script will run from top-to-bottom, executing the statements one-by-one until it reaches the end and then it exits.

Apps, on the other hand, don't exit until the user terminates them (or they crash!). They spend most of their time waiting for input events, then handle those events and go back to sleep.

Input from the user, mostly in the form of touches and taps, is the most important source of events for your app, but there are other types of events as well. For example, the operating system will notify your app when the user receives an incoming phone call, when it has to redraw the screen, when a timer has counted down, etc.



The general flow of events in an app

Everything your app does is triggered by some event.

You can find the project files for the app up to this point under **15-The One-Button App** in the Source Code folder.

16

Chapter 16: Slider & Labels

Eli Ganim

Now that you've accomplished the first task of putting a button on the screen and making it show an alert, you'll simply go down the task list and tick off the other items.

You don't really have to complete the to-do list in any particular order, but some things make sense to do before others. For example, you cannot read the position of the slider if you don't have a slider yet.

Now add the rest of the controls — the slider and the text labels — and turn this app into a real game!

When you're done, the app will look like this:



The game screen with standard UIKit controls



Similarly to what you did in the SwiftUI version of Bullseye, you'll start off with a barebone version of the app and add graphics later.

In this chapter, you'll cover the following:

- **Portrait vs. landscape:** Switch your app to landscape mode.
- **Objects, data and methods:** A quick primer on the basics of object-oriented programming.
- **Adding the other controls:** Add the rest of the controls necessary to complete the user interface of your app.

Portrait vs. landscape

Just like in SwiftUI, your app can be displayed in landscape or portrait orientation. Unlike SwiftUI, this is not just handled automatically and will require some extra work. You'll take care of that now.

Converting the app to landscape

To switch the app from portrait to landscape, you have to do two things:

1. Make the view in **Main.storyboard** landscape instead of portrait.
2. Change the **Supported Device Orientations** setting of the app.

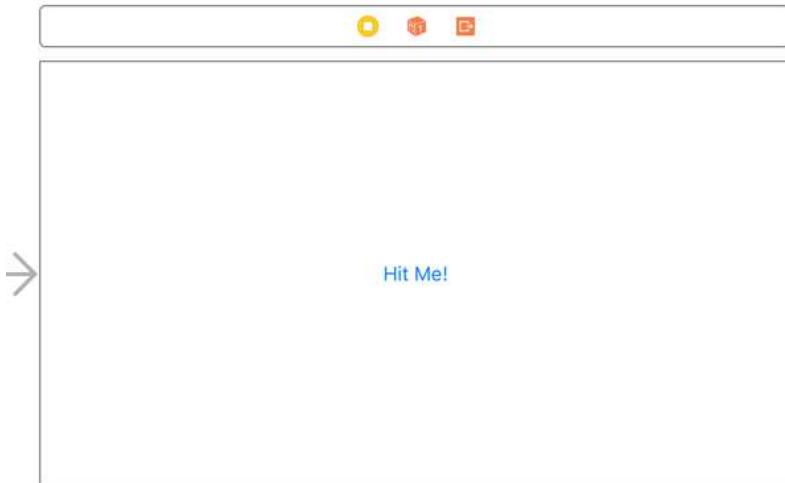
► Open **Main.storyboard**. In Interface Builder, in the **View as: iPhone 8** panel, change **Orientation** to landscape:



Changing the orientation in Interface Builder

This changes the dimensions of the view controller. It also puts the button off-center.

- Move the button back to the center of the view because an untidy user interface just won't do in this day and age.



The view in landscape orientation

That takes care of the view layout.

- Run the app on iPhone 8 Simulator. Note that the screen does not show up as landscape yet, and the button is no longer in the center.

- Choose **Hardware** ▶ **Rotate Left** or **Rotate Right** from Simulator's menu bar at the top of the screen, or hold ⌘ and press the left or right Arrow keys on your keyboard. This will flip the simulator around.

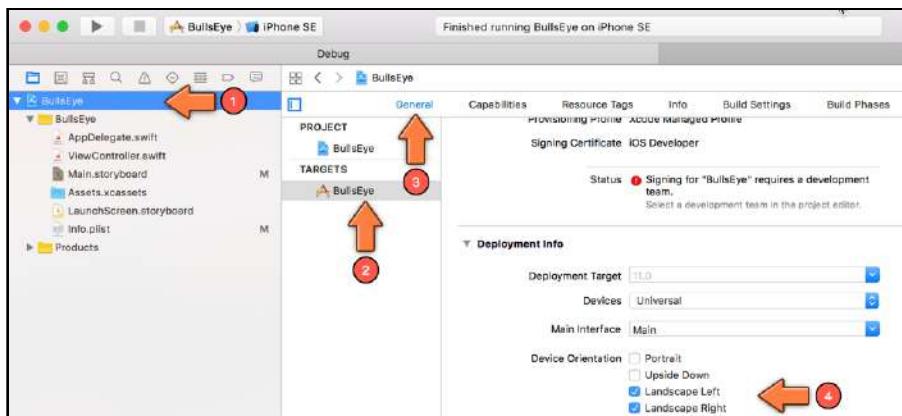
Now, everything will look as it should.

Notice that in landscape orientation the app no longer shows the iPhone's status bar. This gives apps more room for their user interfaces.

To finalize the orientation switch, you should block the Portrait and Upside Down device orientations, just like you did when you built *Bullseye* in SwiftUI.

- Click the blue **Bullseye** project icon at the top of the **Project navigator**. The editor pane of the Xcode window now reveals a bunch of settings for the project.

- Make sure that the **General** tab is selected:



The settings for the project

In the **Deployment Info** section, there is an option for **Device Orientation**.

- Check only the **Landscape Left** and **Landscape Right** options and leave the Portrait and Upside Down options unchecked.

Run the app again and it properly launches in the landscape orientation right from the start.

Understanding objects, data and methods

Time for some programming theory. No, you can't escape it.

Swift is a so-called “object-oriented” programming language, which means that most of the stuff you do involves objects of some kind. I already mentioned a few times that an app consists of objects that send messages to each other.

When you write an iOS app, you'll be using objects that are provided for you by the system, such as the `UIButton` object from `UIKit`, and you'll be making objects of your own, such as view controllers.

Objects

So what exactly *is* an object? Think of an object as a building block of your program. Programmers like to group related functionality into objects. *This* object takes care of parsing a file, *that* object knows how to draw an image on the screen, and *that* object over there can perform a difficult calculation.

Each object takes care of a specific part of the program. In a full-blown app, you will have many different types of objects (tens or even hundreds).

Even your small starter app already contains several different objects. The one you have spent the most time with so far is `ViewController`. The Hit Me! button is also an object, as is the alert pop-up. And the text values that you put on the alert — “Hello, World” and “This is my first app!” — are also objects.

The project also has an object named `AppDelegate` — you’re going to ignore that for the moment, but feel free to look at its source if you’re curious. These object thingies are everywhere!

Data and methods

An object can have both *data* and *functionality*:

- An example of data is the Hit Me! button that you added to the view controller earlier. When you dragged the button into the storyboard, it actually became part of the view controller’s data. Data *contains* something. In this case, the view controller contains the button.
- An example of functionality is the `showAlert` action that you added to respond to taps on the button. Functionality *does* something.

The button itself also has data and functionality. Examples of button data are the text and color of its label, its position on the screen, its width and height and so on. The button also has functionality: It can recognize that the user tapped on it and it will trigger an action in response.

The thing that provides functionality to an object is commonly called a *method*. Other programming languages may call this a “procedure” or “subroutine” or “function.” You will also see the term function used in Swift; a method is simply a function that belongs to an object.

Your `showAlert` action is an example of a method. You can tell it’s a method because the line says `func` (short for “function”) and the name is followed by parentheses:

```
@IBAction func showAlert() {  
    ↑  
    ↑
```

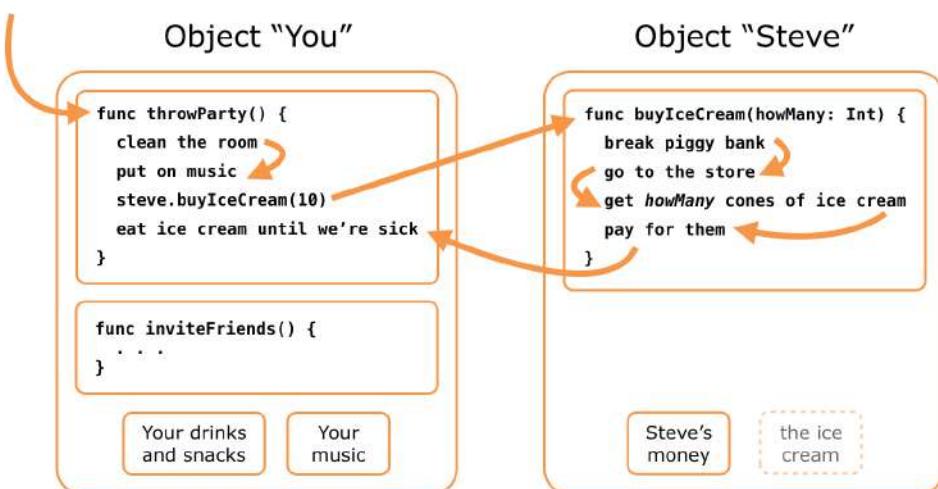
All method definitions start with the word func and have parentheses

If you look through the rest of **ViewController.swift**, you'll see another method, `viewDidLoad()`. It currently doesn't do much; the Xcode template placed it there for your convenience. It's a method that's often used by view controllers, so it's likely that you will need to add some code to it at some point.

Note: These additional methods added by an Xcode template are known as "boilerplate code." If you don't need to add functionality to these boilerplate methods, feel free to remove them — it'll make your code cleaner and more compact.

There's a caveat though; sometimes, the boilerplate code is needed in order not to get a compiler error. You will see this later on when we start using more complex view controllers. So if you remove the boilerplate code and get a compiler error, restore the code and try removing the code selectively until you figure out what is needed and what is not.

The concept of methods may still feel a little weird, so here's an example:



You (or at least an object named "You") want to throw a party, but you forgot to buy ice cream. Fortunately, you have invited the object named Steve who happens to live next door to a convenience store. It won't be much of a party without ice cream so, at some point during your party preparations, you send object Steve a message asking him to bring some ice cream.

The computer now switches to object Steve and executes the commands from his `buyIceCream()` method, one by one, from top to bottom.

When the `buyIceCream()` method is done, the computer returns to your `throwParty()` method and continues with that, so you and your friends can eat the ice cream that Steve brought back with him.

The Steve object also has data. Before he goes to the store, he has money. At the store, he exchanges this money data for other, much more important, data: ice cream! After making that transaction, he brings the ice cream data over to the party (if he eats it all along the way, your program has a bug).

Messages

“Sending a message” sounds more involved than it really is.

It’s a good way to think conceptually of how objects communicate, but there really aren’t any pigeons or mailmen involved. The computer simply jumps from the `throwParty()` method to the `buyIceCream()` method and back again.

Often the terms “calling a method” or “invoking a method” are used instead. That means the exact same thing as sending a message: The computer jumps to the method you’re calling and returns to where it left off when that method is done.

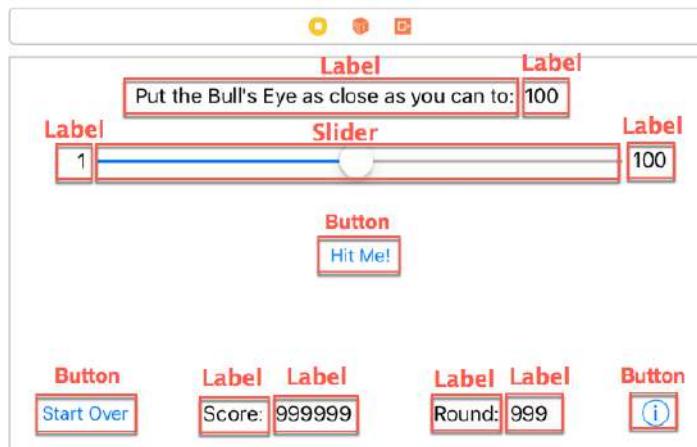
The important thing to remember is that objects have methods (the steps involved in buying ice cream) and data (the actual ice cream and the money to buy it with).

Objects can look at each other’s data (to some extent anyway, just like Steve may not approve if you peek inside his wallet) and can ask other objects to perform their methods.

That’s how you get your app to do things. But not all data from an object can be inspected by other objects and/or code — this is an area known as access control and you’ll learn about this later.

Adding the other controls

Your app already has a button, but you still need to add the rest of the UI controls, also known as “views.” Here is the screen again, this time annotated with the different types of views:



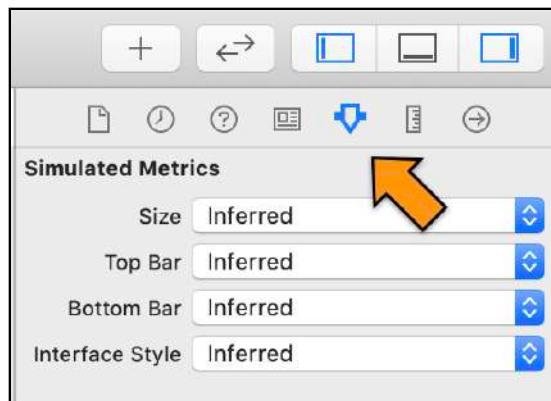
The different views in the game screen

As you can see, you'll add placeholder values in some of the labels (for example, “999999”). That makes it easier to see how the labels will fit on the screen when they're actually used. The score label could potentially hold a large value, so you'd better make sure the label has room for it.

► Try to re-create the above screen on your own by dragging the various controls from the Object Library onto your scene. You'll need a few new Buttons, Labels and a Slider. You can see in the screenshot above how big the items should (roughly) be. It's OK if you're a few points off.

Note: It might seem a little annoying to use the Library panel since it goes away as soon as you drag an item from it. You then have to tap the icon on the toolbar to show the Library panel again to select another item. If you are placing multiple components, just hold down the Alt/Option key (⌥) as you drag an item from the Library panel — the Library panel will remain open, allowing you to select another item.

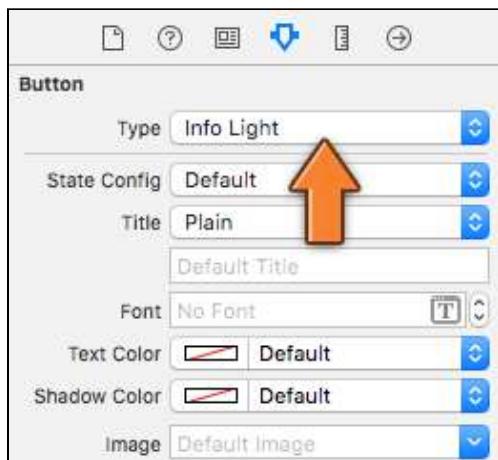
To tweak the settings of these views, you use the **Attributes inspector**. You can find this inspector in the right-hand pane of the Xcode window:



The Attributes inspector

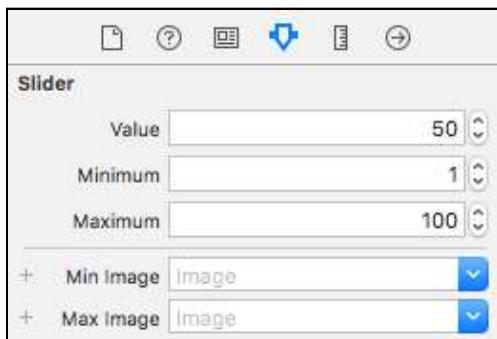
The inspector area shows various aspects of the item that is currently selected. The Attributes inspector, for example, lets you change the background color of a label or the size of the text on a button. You've already seen the Connections inspector that showed the button's actions. As you become more proficient with Interface Builder, you'll be using all of these inspector panes to configure your views.

- Hint: The ⓘ button is actually a regular button, but its **Type** is set to **Info Light** in the Attributes inspector:



The button type lets you change the look of the button

- Also use the Attributes inspector to configure the **slider**. Its minimum value should be 1, its maximum 100, and its current value 50.



The slider attributes

When you're done, you should have 12 user interface elements in your scene: one slider, three buttons and a whole bunch of labels. Excellent!

- Run the app and play with it for a minute. The controls don't really do much yet (except for the button that should still pop up the alert), but you can at least drag the slider around.

You can now tick a few more items off the to-do list, all without much programming! That is going to change really soon, because you will have to write Swift code to actually make the controls do something.

The slider

The next item on your to-do list is: “Read the value of the slider after the user presses the Hit Me! button.”

If, in your messing around in Interface Builder, you did not accidentally disconnect the button from the `showAlert` action, you can modify the app to show the slider’s value in the alert pop-up. (If you did disconnect the button, then you should hook it up again first. You know how, right?)

Remember how you added an action to the view controller in order to recognize when the user tapped the button? You can do the same thing for the slider. This new action will be performed whenever the user drags the slider.

The steps for adding this action are largely the same as before.

- First, go to **ViewController.swift** and add the following at the bottom, just before the final closing curly bracket:

```
@IBAction func sliderMoved(_ slider: UISlider) {  
    print("The value of the slider is now: \(slider.value)")  
}
```

- Second, go to the storyboard and Control-drag from the slider to View controller in the Document Outline. Let go of the mouse button and select **sliderMoved:** from the pop-up. Done!

Just to refresh your memory, the Document Outline sits on the left-hand side of the Interface Builder canvas. It shows the View hierarchy of the storyboard. Here, you can see that the View controller contains a view (succinctly named View), which, in turn, contains the sub-views you've added: the buttons and labels.



The Document Outline shows the view hierarchy of the storyboard

Remember, if the Document Outline is not visible, click the little icon at the bottom of the Xcode window to reveal it:

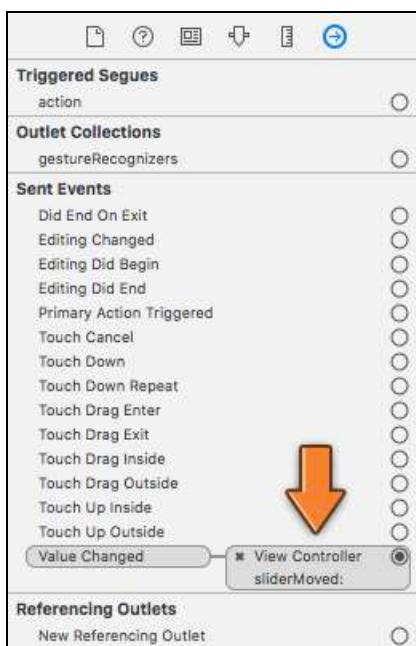


This button shows or hides the Document Outline

When you connect the slider, make sure to Control-drag to View controller (the yellow circle icon), not View controller Scene at the very top. If you don't see the yellow circle icon, then click the arrow in front of View controller scene (called the "disclosure triangle") to expand it.

If all went well, the `sliderMoved:` action is now hooked up to the slider's Value Changed event. This means the `sliderMoved()` method will be called every time the user drags the slider to the left or right, changing its value.

You can verify that the connection was made by selecting the slider and looking at the **Connections inspector**:

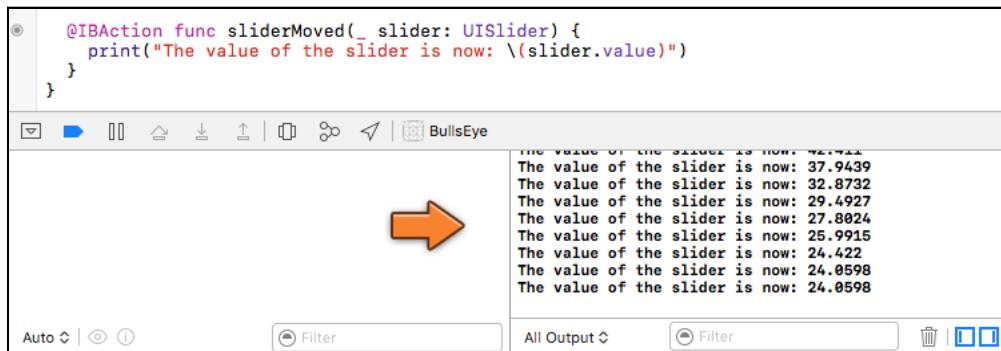


The slider is now hooked up to the view controller

Note: Did you notice that the `sliderMoved:` action has a colon in its name but `showAlert` does not? That's because the `sliderMoved()` method takes a single parameter, `slider`, while `showAlert()` does not have any parameters. If an action method has a parameter, Interface Builder adds a `:` to the name. You'll learn more about parameters and how to use them soon.

- Run the app and drag the slider.

As soon as you start dragging, the Xcode window should open a new pane at the bottom, the **Debug area**, showing a list of messages:



The screenshot shows the Xcode interface with a code editor on the left containing a Swift function definition:

```
⑥     @IBAction func sliderMoved(_ slider: UISlider) {  
        print("The value of the slider is now: \(slider.value)")  
    }  
}
```

To the right of the code editor is the Debug area, which is split into two panes. The left pane contains a toolbar with various icons. An orange arrow points from the text "Printing messages in the Debug area" to the right pane of the Debug area. The right pane displays a list of messages printed by the `print` statements in the code:

```
The value of the slider is now: 42.411  
The value of the slider is now: 37.9439  
The value of the slider is now: 32.8732  
The value of the slider is now: 29.4927  
The value of the slider is now: 27.8024  
The value of the slider is now: 25.9915  
The value of the slider is now: 24.422  
The value of the slider is now: 24.0598  
The value of the slider is now: 24.0598
```

At the bottom of the Debug area are two filter buttons labeled "Filter".

Printing messages in the Debug area

Note: If, for some reason, the Debug area does not show up, you can always show (or hide) the Debug area by using the appropriate toolbar button on the top right corner of the Xcode window. You will notice from the above screenshot that the Debug area is split into two panes. You can control which of the panes is shown/hidden by using the two blue square icons shown above in the bottom-right corner.



Show Debug area

If you swipe the slider all the way to the left, you should see the value go down to 1. All the way to the right, the value should stop at 100.

The `print()` function is a great way to show you what is going on in the app. Its entire purpose is to write a text message to the **Console** — the right-hand pane in the Debug area. Here, you used `print()` to verify that you properly hooked up the action to the slider and that you can read the slider value as the slider is moved.

Developers often use `print()` to make sure their apps are doing the right thing before they add more functionality. Printing a message to the Console is quick and easy.

Note: You may see a bunch of other messages in the Console, too. This is debug output from UIKit and iOS Simulator. You can safely ignore these messages.

Creating a variable and functions

Printing information with `print()` to the Console is very useful during the development process, but it's absolutely useless to users because they can't see the Console when the app is running on a device.

You're going to improve this by showing the value of the slider in the alert pop-up. So how do you get the slider's value?

When you read the slider's value in `sliderMoved()`, that piece of data disappears when the action method ends. It would be handy if you could remember this value until the user taps the Hit Me! button.

► Open `ViewController.swift` and add the following at the top, directly below the line that says `class ViewController:`:

```
var currentValue: Int = 0
```

► Change the contents of the `sliderMoved()` method in `ViewController.swift` to the following:

```
@IBAction func sliderMoved(_ slider: UISlider) {
    currentValue = lroundf(slider.value)
}
```

You removed the `print()` statement and replaced it with this line:

```
currentValue = lroundf(slider.value)
```

► Now change the `showAlert()` method to the following:

```
@IBAction func showAlert() {
    let message = "The value of the slider is: \(currentValue)"

    let alert = UIAlertController(title: "Hello, World",
                                 message: message, // changed
                                 preferredStyle: .alert)

    let action = UIAlertAction(title: "OK", // changed
                             style: .default,
                             handler: nil)
```

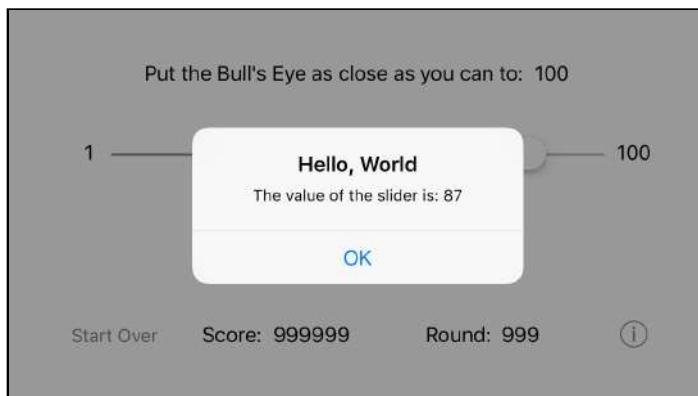
```
    alert.addAction(action)  
    present(alert, animated: true, completion: nil)  
}
```

As before, you create and show a `UIAlertController`, except this time its message says: “The value of the slider is: X,” where X is replaced by the contents of the `currentValue` variable (a whole number between 1 and 100).

Suppose `currentValue` is 34, which means the slider is about one-third to the left. The new code above will convert the string "The value of the slider is: \ (currentValue)" into "The value of the slider is: 34" and put that into a new object named `message`.

The old `print()` did something similar, except that it printed the result to the Console. Here, however, you do not wish to print the result but show it in the alert pop-up. That is why you tell the `UIAlertController` that it should now use this new string as the message to display.

► Run the app, drag the slider and press the button. Now, the alert should show the actual value of the slider.



The alert shows the value of the slider

Cool. You have used a variable, `currentValue`, to remember a particular piece of data, the rounded-off position of the slider, so that it can be used elsewhere in the app — in this case in the alert’s message text.

If you tap the button again without moving the slider, the alert will still show the same value. The variable keeps its value until you put a new one into it.

Your first bug

There is a small problem with the app, though. Maybe you've noticed it already. Here is how to reproduce the problem:

- Press the Stop button in Xcode to completely terminate the app, then press Run again. Without moving the slider, immediately press the Hit Me! button.

The alert now says: "The value of the slider is: 0". But the slider is obviously at the center, so you would expect the value to be 50. You've discovered a bug!

Exercise: Think of a reason why the value would be 0 in this particular situation (start the app, don't move the slider, press the button).

Answer: The clue here is that this only happens when you don't move the slider. Of course, without moving the slider the `sliderMoved()` message is never sent and you never put the slider's value into the `currentValue` variable.

The default value for the `currentValue` variable is 0, and that is what you are seeing here.

- To fix this bug, change the declaration of `currentValue` to:

```
var currentValue: Int = 50
```

Now the starting value of `currentValue` is 50, which should be the same value as the slider's initial position.

- Run the app again and verify that the bug is fixed.

You can find the project files for the app up to this point under **16-Slider and Labels** in the Source Code folder.



Chapter 17: Outlets

Eli Ganim

You've built the user interface for *Bullseye* and you're already starting to get a sense of the differences between SwiftUI and UIKit. This chapter takes care of a few items from the to-do list and covers the following:

- **Improve the slider:** Set the initial slider value (in code) to be whatever value was set in the storyboard instead of assuming an initial value.
- **Generate the random number:** Generate the random number to be used as the target by the game.
- **Add rounds to the game:** Add the ability to start a new round of the game.
- **Display the target value:** Display the generated target number on screen.
- **Calculating the points scored:** Determine how many points to award to the player, based on how closely they positioned the slider to the target value.



Improving the slider

You completed storing the value of the slider into a variable and showing it via an alert. That's great, but you can still improve on it a little.

What if you decide to set the initial value of the slider in the storyboard to something other than 50 — say, 1 or 100? Then `currentValue` would be wrong again because the app always assumes it will be 50 at the start. You'd have to remember to also fix the code to give `currentValue` a new initial value.

Take it from me — that kind of thing is hard to remember, especially when the project becomes bigger and you have dozens of view controllers to worry about, or when you haven't looked at the code for weeks.

Getting the initial slider value

To fix this issue once and for all, you're going to do some work inside the `viewDidLoad()` method in **ViewController.swift**. That method currently looks like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view.
}
```

The `viewDidLoad()` message is sent by UIKit immediately after the view controller loads its user interface from the storyboard file. At this point, the view controller isn't visible yet, so this is a good place to set instance variables to their proper initial values.

► Change `viewDidLoad()` to the following:

```
override func viewDidLoad() {
    super.viewDidLoad()
    currentValue = lroundf(slider.value)
}
```

The idea is that you take whatever value is set on the slider in the storyboard (whether it is 50, 1, 100 or anything else) and use that as the initial value of `currentValue`.

Recall that you need to round off the number, because `currentValue` is an `Int` and integers cannot take decimal (or fractional) numbers.

Unfortunately, Xcode immediately complains about these changes even before you try to run the app.

```
33 class ViewController: UIViewController {  
34     var currentValue: Int = 50  
35  
36     override func viewDidLoad() {  
37         super.viewDidLoad()  
38         currentValue = lroundf(slider.value) ① Use of unresolved identifier 'slider'  
39     }  
}
```

Xcode error message about missing identifier

Note: Xcode tries to be helpful and it analyzes the program for mistakes as you're typing. Sometimes, you may see temporary warnings and error messages that will go away when you complete the changes that you're making.

Don't be too intimidated by these messages; they are only short-lived while the code is in a state of flux.

The above happens because `viewDidLoad()` does not know of anything named `slider`.

Then why did this work earlier, in `sliderMoved()`? Let's take a look at that method again:

```
@IBAction func sliderMoved(_ slider: UISlider) {  
    currentValue = lroundf(slider.value)  
}
```

Here, you do the exact same thing: You round off `slider.value` and put it into `currentValue`. So why does it work here but not in `viewDidLoad()`?

The difference is that, in the code above, `slider` is a *parameter* of the `sliderMoved()` method. Parameters are the things inside the parentheses following a method's name. In this case, there's a single parameter named `slider`, which refers to the `UISlider` object that sent this action message.

Action methods can have a parameter that refers to the UI control that triggered the method. This is convenient when you wish to refer to that object in the method, just as you did here (the object in question being the `UISlider`).

When the user moves the slider, the `UISlider` object basically says, "Hey, View controller! I'm a slider object and I just got moved. By the way, here's my phone number so you can get in touch with me."



The `slider` parameter contains this “phone number,” but it is only valid for the duration of this particular method.

In other words, `slider` is *local*; you cannot use it anywhere else.

Locals

When you first learned about variables, it was mentioned that each variable has a certain lifetime, known as its *scope*. The scope of a variable depends on where in your program you defined that variable.

There are three possible scope levels in Swift:

1. **Global scope:** These objects exist for the duration of the app and are accessible from anywhere.
2. **Instance scope:** This is for variables such as `currentValue`. These objects are alive for as long as the object that owns them stays alive.
3. **Local scope:** Objects with a local scope, such as the `slider` parameter of `sliderMoved()`, only exist for the duration of that method. As soon as the execution of the program leaves this method, the local objects are no longer accessible.

Let’s look at the top part of `showAlert()`:

```
@IBAction func showAlert() {
    let message = "The value of the slider is: \(currentValue)"

    let alert = UIAlertController(title: "Hello, World",
                                 message: message,
                                 preferredStyle: .alert)

    let action = UIAlertAction(title: "OK", style: .default,
                             handler: nil)
    ...
}
```

Because the `message`, `alert`, and `action` objects are created inside the method, they have local scope. They only come into existence when the `showAlert()` action is performed and they cease to exist when the action is done.

As soon as the `showAlert()` method completes, i.e., when there are no more statements for it to execute, the computer destroys the `message`, `alert`, and `action` objects and their storage space is cleared out.

The `currentValue` variable, however, lives on forever... or at least for as long as the `ViewController` does, which is until the user terminates the app. This type of variable is named an *instance variable*, because its scope is the same as the scope of the object instance it belongs to.

In other words, you use instance variables if you want to keep a certain value around, from one action event to the next.

Setting up outlets

So, with this newly gained knowledge of variables and their scopes, how do you fix the error that you encountered?

The solution is to store a reference to the slider as a new instance variable, just like you did for `currentValue`. Except that this time, the data type of the variable is not `Int`, but `UISlider`. And you're not using a regular instance variable but a special one called an *outlet*.

► Add the following line to `ViewController.swift`:

```
@IBOutlet weak var slider: UISlider!
```

It doesn't really matter where this line goes, just as long as it is somewhere inside the brackets for `class ViewController`. It's common to put outlets with the other instance variables — at the top of the class implementation.

This line tells Interface Builder that you now have a variable named `slider` that can be connected to a `UISlider` object. Just as Interface Builder likes to call methods *actions*, it calls these variables *outlets*. Interface Builder doesn't see any of your other variables, only the ones marked with `@IBOutlet`.

Don't worry about `weak` or the exclamation point for now. Why these are necessary will be explained later on. For now, just remember that a variable for an outlet needs to be declared as `@IBOutlet weak var` and has an exclamation point at the end. (Sometimes you'll see a question mark instead; all this hocus pocus will be explained in due time.)

Once you add the `slider` variable, you'll notice that the Xcode error goes away. Does that mean that you can run your app now? Try it and see what happens.

The app crashes on start with an error similar to the following:



```

29 import UIKit
30
31 class ViewController: UIViewController {
32     @IBOutlet weak var slider: UISlider!
33     var currentValue: Int = 50
34
35     override func viewDidLoad() {
36         super.viewDidLoad()
37         currentValue = lroundf(slider.value) // Thread 1: Fatal error: Unexpectedly found nil while unwrapping an Optional value
38     }
39

```

App crash when outlet is not connected

So, what happened?

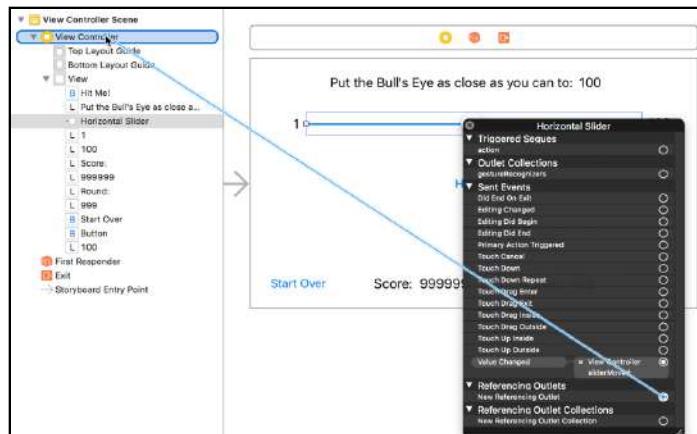
Remember that an outlet has to be *connected* to something in the storyboard. You defined the variable, but you didn't actually set up the connection yet. So, when the app ran and `viewDidLoad()` was called, it tried to find the matching connection in the storyboard and could not — and crashed.

Let's set up the connection in storyboard now.

- Open the storyboard. Hold **Control** and click on the **slider**. Don't drag anywhere, though — a menu should pop up that shows all the connections for this slider. (Instead of Control-clicking, you can also right-click once.)

This pop-up menu works exactly the same as the Connections inspector. It just an alternative approach.

- Click on the open circle next to **New Referencing Outlet** and drag to **View controller**:



Connecting the slider to the outlet

- In the pop-up that appears, select **slider**.

This is the outlet that you just added. You have successfully connected the slider object from the storyboard to the view controller's `slider` outlet.

Now that you have done all this set up work, you can refer to the slider object from anywhere inside the view controller using the `slider` variable.

With these changes in place, it no longer matters what you choose for the initial value of the slider in Interface Builder. When the app starts, `currentValue` will always correspond to that setting.

- Run the app and immediately press the Hit Me! button. It correctly says: "The value of the slider is: 50." Stop the app, go into Interface Builder and change the initial value of the slider to something else — say, 25. Run the app again and press the button. The alert should read 25, now.

Note: When you change the slider value — or the value in any Interface Builder field — remember to tab out of field when you make a change. If you make the change but your cursor remains in the field, the change might not take effect. This is something which can trip you up often.

Put the slider's starting position back to 50 when you're done playing.

Exercise: Give `currentValue` an initial value of 0 again. Its initial value is no longer important — it will be overwritten in `viewDidLoad()` anyway — but Swift demands that all variables always have some value and 0 is as good as any.

Generating the random number

You already read in section 1 how to generate random numbers, so you'll jump straight to the code:

- Add a new variable at the top of `ViewController.swift`, with the other variables:

```
var targetValue = 0
```

You might wonder why we didn't specify the type of the `targetValue` variable, similar to what we'd done earlier for `currentValue`. This is because Swift is able to *infer* the type of variables if it has enough information to work with. Here, for example, you initialize `targetValue` with 0 and, since 0 is an integer value, the compiler knows that `targetValue` will be of type `Int`.

It should be clear why you made `targetValue` an instance variable: You want to calculate the random number in one place – like in `viewDidLoad()` – and then remember it until the user taps the button in `showAlert()` when you have to check this value against the user selection.

Next, you need to generate the random number. A good place to do this is when the game starts.

- Add the following line to `viewDidLoad()` in `ViewController.swift`:

```
targetValue = Int.random(in: 1...100)
```

The complete `viewDidLoad()` should now look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    currentValue = lroundf(slider.value)
    targetValue = Int.random(in: 1...100)
}
```

Displaying the random number

- Change `showAlert()` to the following:

```
@IBAction func showAlert() {
    let message = "The value of the slider is: \(currentValue)" +
                  "\nThe target value is: \(targetValue)"

    let alert = . .
}
```

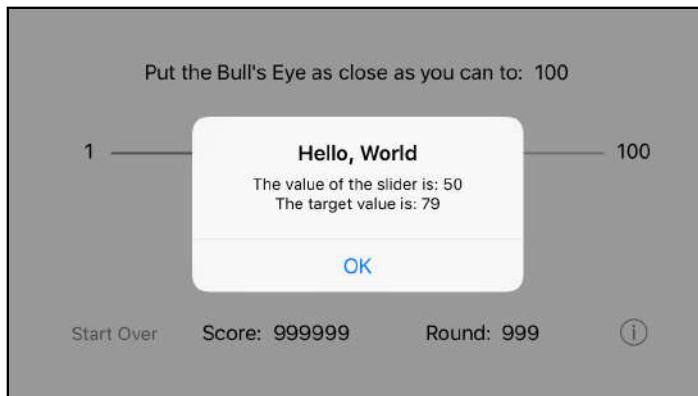
Tip: Whenever you see `...` in a source code listing, this is a shorthand for: This part didn't change. Don't go replacing the existing code with actual ellipsis!

You've simply added the random number, which is now stored in `targetValue`, to the message string. This should look familiar to you by now: The `\(targetValue)` placeholder is replaced by the actual random number.

The `\n` character sequence is new. It means that you want to insert a special “new line” character at that point, which will break up the text into two lines so the

message is a little easier to read. The `+` is also new but is simply used here to combine two strings. We could just as easily have written it as a single long string, but it might not have looked as good to the reader.

► Run the app and try it out!



The alert shows the target value on a new line

Note: Earlier, you used the `+` operator to add two numbers together (just like how it works in math) but, here, you're also using `+` to glue different bits of text into one big string.

Swift allows the use of the same operator for different tasks, depending on the data types involved. If you have two integers, `+` adds them up. But with two strings, `+` concatenates, or combines, them into a longer string.

Programming languages often use the same symbols for different purposes, depending on the context. After all, there are only so many symbols to go around!

Adding rounds to the game

If you press the Hit Me! button a few times, you'll notice that the random number never changes. I'm afraid the game won't be much fun that way. This happens because you generate the random number in `viewDidLoad()` and never again afterwards.

The `viewDidLoad()` method is only called once when the view controller is created during app startup. The item on the to-do list actually said: “Generate a random number *at the start of each round*”. Let’s talk about what a round means in terms of this game.

When the game starts, the player has a score of 0 and the round number is 1. You set the slider halfway (to value 50) and calculate a random number. Then you wait for the player to press the Hit Me! button. As soon as they do, the round ends.

You calculate the points for this round and add them to the total score. Then you increment the round number and start the next round. You reset the slider to the halfway position again and calculate a new random number. Rinse, repeat.

Starting a new round

Whenever you find yourself thinking something along the lines of, “At this point in the app we have to do such and such,” then it makes sense to create a new method for it. This method will nicely capture that functionality in a self-contained unit of its own.

- With that in mind, add the following new method to `ViewController.swift`:

```
func startNewRound() {  
    targetValue = Int.random(in: 1...100)  
    currentValue = 50  
    slider.value = Float(currentValue)  
}
```

It doesn’t really matter where you put the code, as long as it is inside the `ViewController` implementation (within the class curly brackets), so that the compiler knows it belongs to the `ViewController` object.

It’s not very different from what you did before, except that you moved the logic for setting up a new round into its own method, `startNewRound()`. The advantage of doing this is that you can execute this logic from more than one place in your code.

Using the new method

First, you’ll call this new method from `viewDidLoad()` to set up everything for the very first round. Recall that `viewDidLoad()` happens just once when the app starts up, so this is a great place to begin the first round.



- Change `viewDidLoad()` to:

```
override func viewDidLoad() {
    super.viewDidLoad()
    startNewRound() // Replace previous code with this
}
```

Note that you've removed some of the existing statements from `viewDidLoad()` and replaced them with just the call to `startNewRound()`.

You will also call `startNewRound()` after the player pressed the Hit Me! button, from within `showAlert()`.

- Make the following change to `showAlert()`:

```
@IBAction func showAlert() {
    ...
    startNewRound()
}
```

The call to `startNewRound()` goes at the very end, right after `present(alert, ...)`.

Until now, the methods from the view controller have been invoked for you by UIKit when something happened: `viewDidLoad()` is performed when the app loads, `showAlert()` is performed when the player taps the button, `sliderMoved()` when the player drags the slider, and so on. This is the event-driven model we talked about earlier.

It is also possible to call methods directly, which is what you're doing here. You are sending a message from one method in the object to another method in that same object.

In this case, the view controller sends the `startNewRound()` message to itself in order to set up the new round. Program execution will then switch to that method and execute its statements one-by-one. When there are no more statements in the method, it returns to the calling method and continues with that — either `viewDidLoad()`, if this is the first time, or `showAlert()` for every round after.

Calling methods in different ways

Sometimes, you may see method calls written like this:

```
self.startNewRound()
```

That does the exact same thing as `startNewRound()` without `self.` in front. Recall you read that the view controller sends the message to itself. Well, that's exactly what `self` means.

To call a method on an object, you'd normally write:

```
receiver.methodName(parameters)
```

The `receiver` is the object you're sending the message to. If you're sending the message to yourself, then the receiver is `self`. But because sending messages to `self` is very common, you can also leave this special keyword out for many cases.

This isn't exactly the first time you've called methods. `addAction()` is a method on `UIAlertController` and `present()` is a method that all view controllers have, including yours.

When you write Swift apps, a lot of what you do is calling methods on objects, because that is how the objects in your app communicate.

The advantages of using methods

It's very helpful to put the "new round" logic into its own method. If you didn't, the code for `viewDidLoad()` and `showAlert()` would look like this:

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

    targetValue = Int.random(in: 1...100)
    currentValue = 50
    slider.value = Float(currentValue)
}

@IBAction func showAlert() {
    .

    targetValue = Int.random(in: 1...100)
    currentValue = 50
    slider.value = Float(currentValue)
}
```

Can you see what is going on here? The same functionality is duplicated in two places. Sure, it is only three lines of code but, often, the code you duplicate could be much larger.

And what if you decide to make a change to this logic (as you will shortly)? Then you will have to make the same change in two places.

You might be able to remember to do so if you recently wrote this code and it is still fresh in memory, but, if you have to make that change a few weeks down the road, chances are that you'll only update it in one place and forget about the other.

Code duplication is a big source of bugs. So, if you need to do the same thing in two different places, consider making a new method for it instead of duplicating code.

Naming methods

The name of the method also helps to make it clear as to what it is supposed to be doing. Can you tell at a glance what the following does?

```
targetValue = Int.random(in: 1...100)
currentValue = 50
slider.value = Float(currentValue)
```

You probably have to reason your way through it: “It is calculating a new random number and then resets the position of the slider, so I guess it must be the start of a new round.”

Some programmers will use a comment to document what is going on (and you can do that too), but, in my opinion, the following is much clearer than the above block of code with an explanatory comment:

```
startNewRound()
```

This line practically spells out for you what it will do. And if you want to know the specifics of what goes on in a new round, you can always look up the `startNewRound()` method implementation.

Well-written source code speaks for itself!

- Run the app and verify that it calculates a new random number between 1 and 100 after each tap on the button.

You should also have noticed that, after each round, the slider resets to the halfway position. That happens because `startNewRound()` sets `currentValue` to 50 and then tells the slider to go to that position. That is the opposite of what you did before (you used to read the slider's position and put it into `currentValue`), but it would work better in the game if you start from the same position in each round.

Exercise: Just for fun, modify the code so that the slider does not reset to the halfway position at the start of a new round.

Type conversion

By the way, you may have been wondering what `Float(...)` does in this line:

```
slider.value = Float(currentValue)
```

Swift is a *strongly typed* language, meaning that it is really picky about the shapes that you can put into the boxes. For example, if a variable is an `Int`, you cannot put a `Float`, or a non-whole number, into it, and vice versa.

The value of a `UISlider` happens to be a `Float` — you've seen this when you printed out the value of the slider — but `currentValue` is an `Int`. So the following won't work:

```
slider.value = currentValue
```

The compiler considers this an error. Some programming languages are happy to convert the `Int` into a `Float` for you, but Swift wants you to be explicit about such conversions.

When you say `Float(currentValue)`, the compiler takes the integer number that's stored in `currentValue` and converts it into a new `Float` value that it can pass on to the `UISlider`.

Because Swift is stricter about this sort of thing than most other programming languages, it is often a source of confusion for newcomers to the language. Unfortunately, Swift's error messages aren't always very clear about what part of the code is wrong or why.

Just remember, if you get an error message saying, “Cannot assign value of type ‘something’ to type ‘something else’,” then you’re probably trying to mix incompatible data types. The solution is to explicitly convert one type to the other — if conversion is allowed, of course — as you’ve done here.

Displaying the target value

Great, you figured out how to calculate the random number and how to store it in an instance variable, `targetValue`, so that you can access it later.

Now, you are going to show that target number on the screen. Without it, the player won’t know what to aim for and that would make the game impossible to win.

Setting up the storyboard

When you set up the storyboard, you added a label for the target value (top-right corner). The trick is to put the value from the `targetValue` variable into this label. To do that, you need to accomplish two things:

1. Create an outlet for the label so you can send it messages.
2. Give the label new text to display.

This will be very similar to what you did with the slider. Recall that you added an `@IBOutlet` variable so you could reference the slider anywhere from within the view controller. Using this outlet variable you could ask the slider for its value, through `slider.value`. You’ll do the same thing for the label.

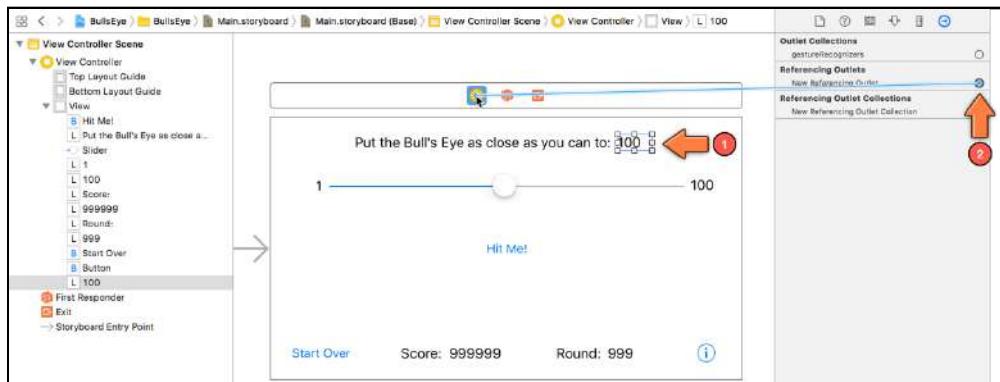
► In `ViewController.swift`, add the following line below the other outlet declaration:

```
@IBOutlet weak var targetLabel: UILabel!
```

► In `Main.storyboard`, click to select the correct label — the one at the very top that says “100.”

► Go to the **Connections inspector** and drag from **New Referencing Outlet** to the yellow circle at the top of your view controller in the central scene.

You could also drag to the **View Controller** in the Document Outline — there are many ways to do the same thing in Interface Builder.



Connecting the target value label to its outlet

- Select **targetLabel** from the pop-up, and the connection is made.

Displaying the target value via code

- Add the following method below `startNewRound()` in **ViewController.swift**:

```
func updateLabels() {  
    targetLabel.text = String(targetValue)  
}
```

You're putting this logic in a separate method because it's something you might use from different places.

The name of the method makes it clear what it does: It updates the contents of the labels. Currently it's just setting the text of a single label, but later on you will add code to update the other labels as well (total score, round number).

The code inside `updateLabels()` should have no surprises for you, although you may wonder why you cannot simply do:

```
targetLabel.text = targetValue
```

The answer again is that you cannot put a value of one data type into a variable of another type — the square peg just won't go in the round hole.

The `targetLabel` outlet references a `UILabel` object. The `UILabel` object has a `text` property, which is a `String` object. So, you can only put `String` values into `text`, but `targetValue` is an `Int`. A direct assignment won't fly because an `Int` and a `String` are two very different types.

So, you have to convert the `Int` into a `String`, and that is what `String(targetValue)` does. It's similar to what you've done before with `Float(...)`.

Just in case you were wondering, you could also convert `targetValue` to a `String` by using string interpolation, like you've done before:

```
targetLabel.text = "\(targetValue)"
```

Which approach you use is a matter of taste. Either approach will work fine.

Notice that `updateLabels()` is a regular method — it is not attached to any UI controls as an action — so it won't do anything until you actually call it. You can tell because it doesn't say `@IBAction` before `func`.

Action methods vs. normal methods

So what is the difference between an action method and a regular method?

Answer: Nothing.

An action method is really just the same as any other method. The only special thing is the `@IBAction` attribute, which allows Interface Builder to see the method so you can connect it to your buttons, sliders and so on.

Other methods, such as `viewDidLoad()`, don't have the `@IBAction` specifier. This is good because all kinds of mayhem would occur if you hooked these up to your buttons.

This is the simple form of an action method:

```
@IBAction func showAlert()
```

You can also ask for a reference to the object that triggered this action, via a parameter:

```
@IBAction func sliderMoved(_ slider: UISlider)
@IBAction func buttonTapped(_ button: UIButton)
```

But the following method cannot be used as an action from Interface Builder:

```
func updateLabels()
```

That's because it is not marked as `@IBAction` and as a result, Interface Builder can't see it. To use `updateLabels()`, you will have to call it yourself.

Calling the method

The logical place to call `updateLabels()` would be after each call to `startNewRound()`, because that is where you calculate the new target value. So, you could always add a call to `updateLabels()` in `viewDidLoad()` and `showAlert()`, but there's another way, too!

What is this other way, you ask? Well, if `updateLabels()` is always (or at least in your current code) called after `startNewRound()`, why not call `updateLabels()` directly from `startNewRound()` itself? That way, instead of having two calls in two separate places, you can have a single call.

► Change `startNewRound()` to:

```
func startNewRound() {
    targetValue = Int.random(in: 1...100)
    currentValue = 50
    slider.value = Float(currentValue)
    updateLabels() // Add this line
}
```

You should be able to type just the first few letters of the method name, like **upd**, and Xcode will show you a list of suggestions matching what you typed.

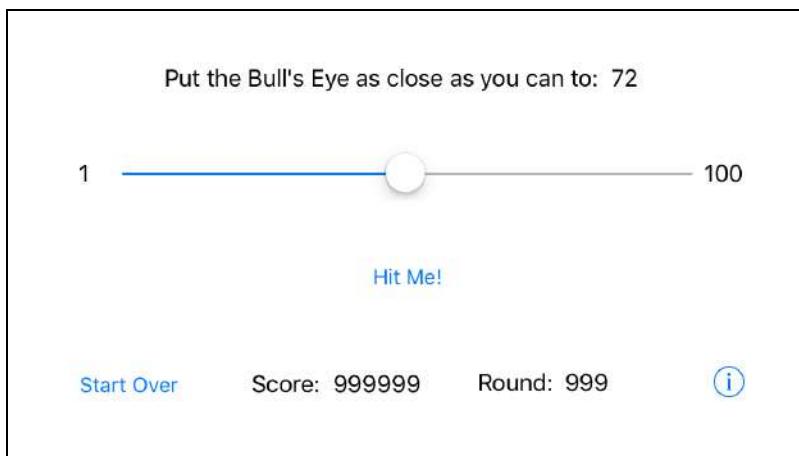
Press **Enter** (or **Tab**) to accept the suggestion (if you are on the right item — or scroll the list to find the right item and then press Enter)



Xcode autocomplete offers suggestions

Also worth noting is that you don't have to start typing the method (or property) name you're looking from the beginning — Xcode uses fuzzy search and typing "dateL" or "label" should help you find "updateLabels" just as easily.

- Run the app and you'll actually see the random value on the screen. That should make it a little easier to aim for.



The label in the top-right corner now shows the random value

Calculating the points scored

Now that you have both the target value (the random number) and a way to read the slider's position, you can calculate how many points the player scored. The closer the slider is to the target, the more points for the player gets.

- Make this change to `showAlert()`:

```
@IBAction func showAlert() {
    let difference = abs(targetValue - currentValue)
    let points = 100 - difference

    let message = "You scored \(points) points"
}
```

Showing the total score

In this game you want to show the player's total score on the screen. After every round, the app should add the newly scored points to the total and then update the score label.

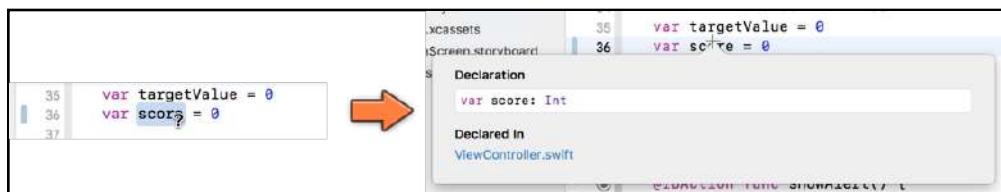
Storing the total score

Because the game needs to keep the total score around for a long time, you will need an instance variable.

- Add a new score instance variable to **ViewController.swift**:

```
class ViewController: UIViewController {  
  
    var currentValue: Int = 0  
    var targetValue = 0  
    var score = 0 // add this line
```

Again, you make use of type inference to not specify a type for score.



Discover the inferred type for a variable

Updating the total score

Now, `showAlert()` can be amended to update this score variable.

- Make the following changes:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    let points = 100 - difference  
  
    score += points // add this line  
  
    let message = "You scored \(points) points"  
    . . .
```

Nothing too shocking here. You just added the following line:

```
score += points
```

This adds the points that the user scored in this round to the total score. You could also have written it like this:

```
score = score + points
```

Displaying the score

To display your current score, you're going to do the same thing that you did for the target label: hook up the score label to an outlet and put the score value into the label's `text` property.

Exercise: See if you can do the above by yourself. You've already done these things before for the target value label, so you should be able to repeat those steps for the score label.

Done? You should have added this line to `ViewController.swift`:

```
@IBOutlet weak var scoreLabel: UILabel!
```

Then, you connect the relevant label on the storyboard (the one that says 999999) to the new `scoreLabel` outlet.

Unsure how to connect the outlet? There are several ways to make connections from user interface objects to the view controller's outlets:

- Control-click on the object to get a context-sensitive pop-up menu. Then, drag from New Referencing Outlet to View controller (you did this with the slider).
- Go to the Connections Inspector for the label. Drag from New Referencing Outlet to View controller (you did this with the target label).
- Control-drag **from** View controller to the label (give this one a try now) — doing it the other way, Control-dragging from the label to View controller, won't work.

Great, that gives you a `scoreLabel` outlet that you can use to display the score. Now, where in the code can you do that? In `updateLabels()`, of course.

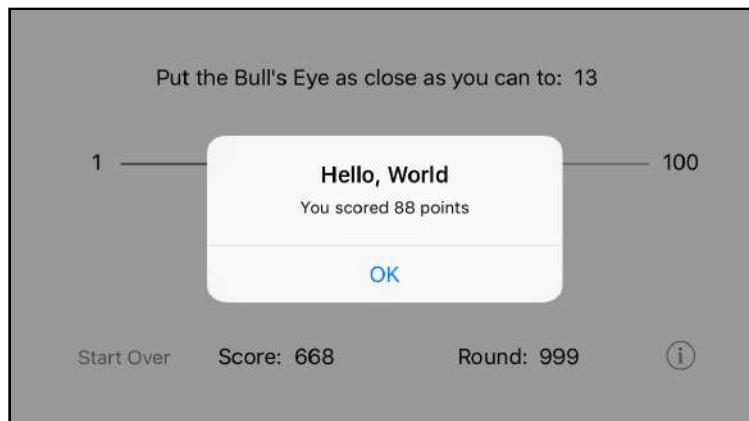


- Back in **ViewController.swift**, change `updateLabels()` to the following:

```
func updateLabels() {  
    targetLabel.text = String(targetValue)  
    scoreLabel.text = String(score) // add this line  
}
```

Nothing new, here. You convert the score — which is an `Int` — into a `String` and then pass that string to the label's `text` property. In response to that, the label will redraw itself with the new score.

- Run the app and verify that the points for this round are added to the total score label whenever you tap the button.



The score label keeps track of the player's total score

One more round...

Speaking of rounds, you also have to increment the round number each time the player starts a new round.

Exercise: Keep track of the current round number (starting at 1) and increment it when a new round starts. Display the current round number in the corresponding label. I may be throwing you into the deep end here, but if you've been able to follow the instructions so far, then you've already seen all the pieces you will need to pull this off. Good luck!

If you guessed that you had to add another instance variable, then you are right. You should add the following line (or something similar) to **ViewController.swift**:

```
var round = 0
```

It's also OK if you included the name of the data type, even though that is not strictly necessary:

```
var round: Int = 0
```

Also, add an outlet for the label:

```
@IBOutlet weak var roundLabel: UILabel!
```

As before, you should connect the label to this outlet in Interface Builder.

Note: Don't forget to make those connections.

Forgetting to make the connections in Interface Builder is an often-made mistake, especially by yours truly.

It happens to me all the time that I make the outlet for a button and write the code to deal with taps on that button but, when I run the app, it doesn't work. Usually, it takes me a few minutes and some head scratching to realize that I forgot to connect the button to the outlet or the action method.

You can tap on the button all you want but, unless that connection exists, your code will not respond.

Finally, `updateLabels()` should be modified like this:

```
func updateLabels() {
    targetLabel.text = String(targetValue)
    scoreLabel.text = String(score)
    roundLabel.text = String(round) // add this line
}
```

Did you also figure out where to increment the `round` variable?

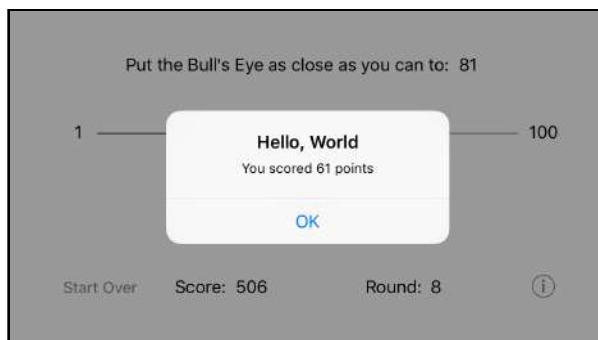
I'd say the `startNewRound()` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

- Change `startNewRound()` to:

```
func startNewRound() {  
    round += 1           // add this line  
    targetValue = ...  
}
```

Note that, when you declared the `round` instance variable, you gave it a default value of 0. Therefore, when the app starts up, `round` is initially 0. When you call `startNewRound()` for the very first time, it adds 1 to this initial value and, as a result, the first round is properly counted as round 1.

Run the app and try it out. The round counter should update whenever you press the Hit Me! button.



The round label counts how many rounds have been played

You're making great progress; well done!

You can find the project files for the app up to this point under **17-Outlets** in the Source Code folder.

Chapter 18: Polish

Eli Ganim

At this point, your game is fully playable. The gameplay rules are all implemented and the logic doesn't seem to have any big flaws, but there's still some room for improvement.

This chapter will cover the following:

- **Tweaks:** Small UI tweaks to make the game look and function better.
- **The alert:** Updating the alert view functionality so that the screen updates *after* the alert goes away.
- **Start over:** Resetting the game to start afresh.



Tweaks

Obviously, the game is not very pretty yet — you will get to work on that soon. In the mean time, there are a few smaller tweaks you can make.

The alert title

Unless you already changed it, the title of the alert still says “Hello, World!” You could give it the name of the game, *Bullseye*, but there’s a better idea. What if you change the title depending on how well the player did?

If the player put the slider right on the target, the alert could say: “Perfect!” If the slider is close to the target but not quite there, it could say, “You almost had it!” If the player is way off, the alert could say: “Not even close...”

And so on. This gives the player a little more feedback on how well they did.

You already did that in Section 1, so without further ado here’s the updated `showAlert()` method:

```
@IBAction func showAlert() {
    let difference = abs(targetValue - currentValue)
    let points = 100 - difference
    score += points

    // add these lines
    let title: String
    if difference == 0 {
        title = "Perfect!"
    } else if difference < 5 {
        title = "You almost had it!"
    } else if difference < 10 {
        title = "Pretty good!"
    } else {
        title = "Not even close..."
    }

    let message = "You scored \(points) points"

    let alert = UIAlertController(title: title, // change this
                                message: message,
                                preferredStyle: .alert)

    let action = UIAlertAction(title: "OK", style: .default,
                             handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
```

```
    startNewRound()
```

You create a new string constant named `title`, which will contain the text that is set for the alert title. Initially, this `title` doesn't have any value. We'll discuss the `title` variable and how it is set up a bit more in detail just a little further on.

To decide which title text to use, you look at the difference between the slider position and the target:

- If it equals 0, then the player was spot-on and you set `title` to “Perfect!”
- If the difference is less than 5, you use the text, “You almost had it!”
- A difference less than 10 is “Pretty good!”
- However, if the difference is 10 or greater, then you consider the player’s attempt “Not even close...”

Can you follow the logic, here? It’s just a bunch of `if` statements that consider the different possibilities and choose a string in response.

When you create the `UIAlertController` object, you now give it this `title` string instead of some fixed text.

Constant initialization

In the above code, did you notice that `title` was declared explicitly as being a `String` constant? And did you ask yourself why type inference wasn’t used there instead? Also, if `title` is a constant, how do we have code which sets its value in multiple places?

The answer to all of these questions lies in how constants (or `let` values, if you prefer) are initialized in Swift.

You could certainly have used type inference to declare the type for `title` by setting the initial declaration to:

```
let title = ""
```

But do you see the issue there? Now you’ve actually set the value for `title` and since it’s a constant, you can’t change the value again. So, the following lines where the `if` condition logic sets a value for `title` would now throw a compiler error since you are trying to set a value to a constant which already has a value. (Go on, try it out for yourself! You know you want to...)



One way to fix this would be to declare `title` as a variable rather than a constant. Like this:

```
var title = ""
```

The above would work great, and the compiler error would go away. But you've got to ask yourself, do you really need a variable there? Or, would a constant do? It's preferable to use constants where possible since they have less risk of unexpected side-effects because the value was accidentally changed in some fashion — for example, because one of your team members changed the code to use a variable that you had originally depended on being unchanged. That is why the code was written the way it was. However, you can go with whichever option you prefer since either approach would work.

But if you do declare `title` as a constant, how is it that your code above assigns multiple values to it?

The secret is in the fact that while there are indeed multiple values being assigned to `title`, only one value would be assigned per each call to `showAlert` since the branches of an `if` condition are mutually exclusive.

So, since `title` starts out without a value (the `let title: String` line only assigns a type, not a value), as long as the code ensures that `title` would always be initialized to a value before the value stored in `title` is accessed, the compiler will not complain.

Again, you can test this by removing the `else` condition in the block of code where a value is assigned to `title`.

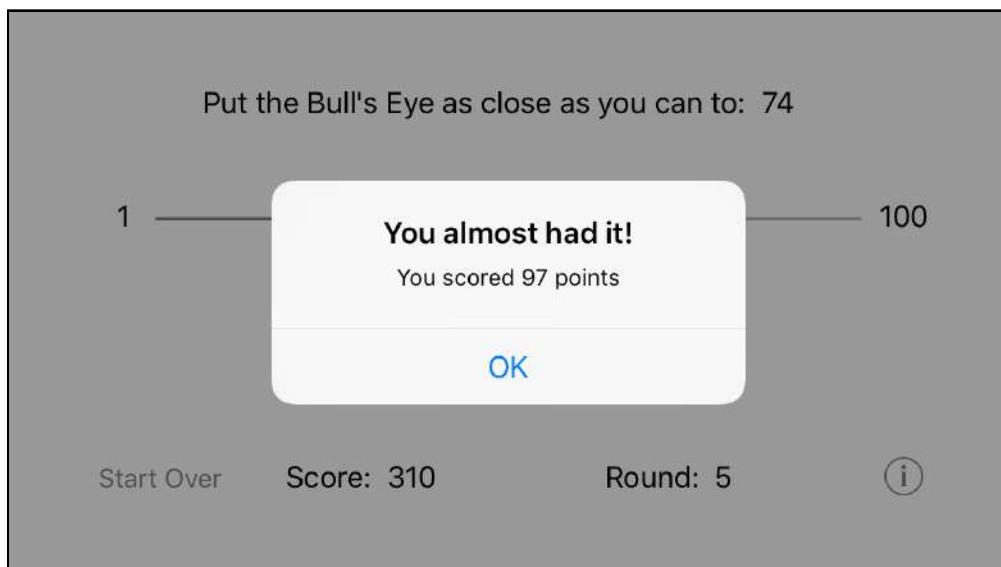
Since an `if` condition is only one branch of a test, you need an `else` branch in order for the tests (and the assignment to `title`) to be exhaustive.

So, if you remove the `else` branch, Xcode will immediately complain with an error like: "Constant 'title' used before being initialized."

```
59  let title: String
60  if difference == 0 {
61      title = "Perfect!"
62  } else if difference < 5 {
63      title = "You almost had it!"
64  } else if difference < 10 {
65      title = "Pretty good!"
66 // } else {
67 //     title = "Not even close..."
68 }
69
70 let message = "You scored \(points) points"
71
72 let alert = UIAlertController(title: title,           ⚡ Constant 'title' used before being initialized
73                             message: message,
74                             preferredStyle: .alert)
```

A constant needs to be initialized exhaustively

Run the app and play the game for a bit. You'll see that the title text changes depending on how well you're doing. That `if` statement sure is handy!



The alert with the new title

Bonus points

Exercise: Give players an additional 100 bonus points when they get a perfect score. This will encourage players to really try to place the bullseye right on the target. Otherwise, there isn't much difference between 100 points for a perfect score and 98 or 95 points if you're close but not quite there.

Now there is an incentive for trying harder — a perfect score is no longer worth just 100 but 200 points! Maybe you can also give the player 50 bonus points for being just one off.

- Here's how you can make these changes:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    var points = 100 - difference // change let to var  
  
    let title: String  
    if difference == 0 {  
        title = "Perfect!"  
        points += 100 // add this line  
    } else if difference < 5 {  
        title = "You almost had it!"  
        if difference == 1 { // add these lines  
            points += 50 // add these lines  
        }  
    } else if difference < 10 {  
        title = "Pretty good!"  
    } else {  
        title = "Not even close..."  
    }  
    score += points // move this line here from  
    the top  
    . . .  
}
```

You should notice a few things:

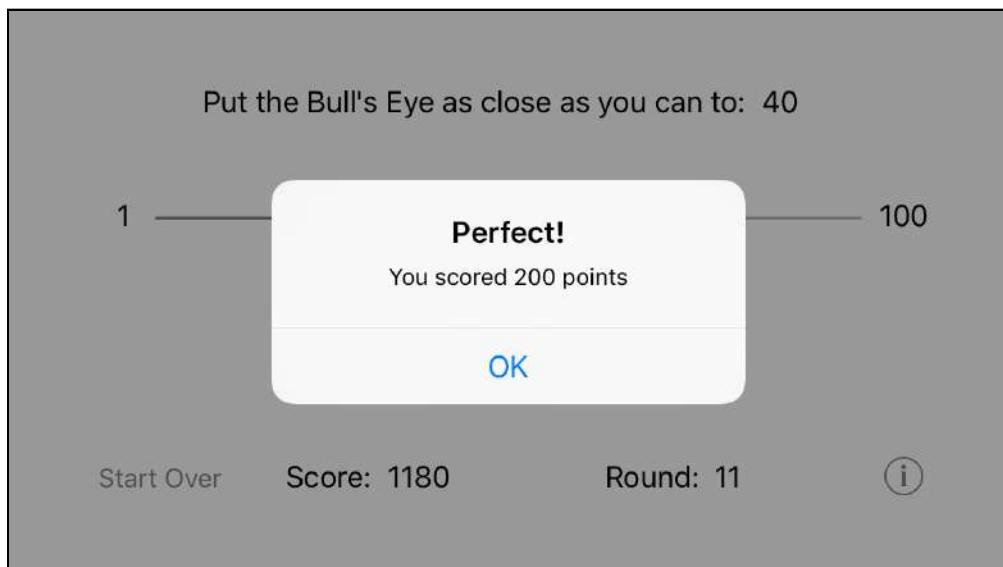
- In the first `if` you'll see a new statement between the curly brackets. When the difference is equal to zero, you now not only set `title` to "Perfect!" but also award an extra 100 points.
- The second `if` has changed, too. There is now an `if` inside another `if`. Nothing wrong with that! You want to handle the case where `difference` is 1 in order to give the player bonus points. That happens inside the new `if` statement.

After all, if the difference is more than 0 but less than 5, it could be 1 (but not necessarily all the time). Therefore, you perform an additional check to see if the difference truly is 1, and if so, add 50 extra points.

- Because these new `if` statements add extra points, `points` can no longer be a constant; it now needs to be a variable. That's why you change it from `let` to `var`.
- Finally, the line `score += points` has moved below the `ifs`. This is necessary because the app updates the `points` variable inside those `if` statements (if the conditions are right) and you want those additional points to count towards the final score.

If your code is slightly different, then that's fine too, as long as it works! There is often more than one way to program something, and if the results are the same, then any approach is equally valid.

► Run the app to see if you can score some bonus points!



Raking in the points...

The alert

One annoying thing about the app is that as soon as you tap the Hit Me! button and the alert pops up, the slider immediately jumps back to its center position, the round number increments, and the target label already gets the new random number.

What happens is that the new round has already begun while you're still watching the results of the last round. That's a little confusing (and annoying).

It would be better to wait on starting the new round until *after* the player has dismissed the alert pop-up. Only then is the current round truly over.

Asynchronous code execution

Maybe you're wondering why this isn't already happening? After all, in `showAlert()` you only call `startNewRound()` after you've shown the alert pop-up:

```
@IBAction func showAlert() {  
    . . .  
    let alert = UIAlertController(. . .)  
    let action = UIAlertAction(. . .)  
    alert.addAction(action)  
  
    // Here you make the alert visible:  
    present(alert, animated: true, completion: nil)  
  
    // Here you start the new round:  
    startNewRound()  
}
```

Contrary to what you might expect, `present(alert:animated:completion:)` doesn't hold up execution of the rest of the method until the alert pop-up is dismissed. That's how alerts on other platforms tend to work, but not on iOS.

Instead, `present(alert:animated:completion:)` puts the alert on the screen and immediately returns control to the next line of code in the method. The rest of the `showAlert()` method is executed right away, and the new round starts before the alert pop-up has even finished animating.

In programmer-speak, alerts work *asynchronously*. We'll talk much more about that in a later chapter, but what it means for you right now is that you don't know in advance when the alert will be done. But you can bet it will be well after `showAlert()` has finished.

Alert event handling

So, if your code execution can't wait in `showAlert()` until the pop-up is dismissed, then how do you wait for it to close?

The answer is simple: events! As you've seen, a lot of the programming for iOS involves waiting for specific events to occur — buttons being tapped, sliders being moved, and so on. This is no different. You have to wait for the “alert dismissed” event somehow. In the mean time, you simply do nothing.

Here's how it works:

For each button on the alert, you have to supply a `UIAlertAction` object. This object tells the alert what the text on the button is — “OK” — and what the button looks like (you're using the default style, here):

```
let action = UIAlertAction(title: "OK", style: .default,  
    handler: nil)
```

The third parameter, `handler`, tells the alert what should happen when the button is pressed. This is the “alert dismissed” event you've been looking for! Currently `handler` is `nil`, which means nothing happens. In case you're wondering, a `nil` in Swift indicates “no value.” You will learn more about `nil` values later on.

You can however, give the `UIAlertAction` some code to execute when the OK button is tapped. When the user finally taps OK, the alert will remove itself from the screen and jump to your code. That's your cue to start a new round. This is also known as the *callback* pattern. There are several ways this pattern manifests on iOS. Often you'll be asked to create a new method to handle the event. But here you'll use a *closure*.

► Change the bottom bit of `showAlert()` to:

```
@IBAction func showAlert() {  
  
    let alert = UIAlertController(...)  
  
    let action = UIAlertAction(title: "OK", style: .default,  
        handler: { _ in  
            self.startNewRound()  
        })  
  
    alert.addAction(action)  
    present(alert, animated: true, completion: nil)  
}
```

Two things have happened here:

1. You removed the call to `startNewRound()` from the bottom of the method. (Don't forget this part!)

2. You placed it inside a block of code that you gave to `UIAlertAction`'s `handler` parameter.

Such a block of code is called a *closure*. You can think of it as a method without a name. This code is not performed right away. Rather, it's performed only when the OK button is tapped. This particular closure tells the app to start a new round (and update the labels) when the alert is dismissed.

- Run the app and see for yourself.

Start over

No, you're not going to throw away the source code and start this project all over! This part is about the game's "Start Over" button. This button is supposed to reset the score and start over from the first round.

One use of the Start Over button is for playing against another person. The first player does ten rounds, then the score is reset and the second player does ten rounds. The player with the highest score wins.

Exercise: Try to implement the Start Over button on your own. You've already seen how you can make the view controller react to button presses, and you should be able to figure out how to change the `score` and `round` variables.

How did you do? If you got stuck, then follow the instructions below.

The new method

First, add a method to `ViewController.swift` that starts a new game. You should put it near `startNewRound()` because the two are conceptually related.

- Add the new method:

```
func startNewGame() {  
    score = 0  
    round = 0  
    startNewRound()  
}
```

This method resets `score` and `round` to zero, and starts a new round as well.



Notice that you set `round` to 0 here, not to 1. You use 0 because incrementing the value of `round` is the first thing that `startNewRound()` does. If you were to set `round` to 1, then `startNewRound()` would add another 1 to it and the first round would actually be labeled round 2.

So, you begin at 0, let `startNewRound()` add one and everything works great.

It's probably easier to figure this out from the code than from my explanation. This should illustrate why we don't program computers in English.

You also need an action method to handle taps on the Start Over button. You could write a new method like the following:

```
@IBAction func startOver() {  
    startNewGame()  
}
```

But you'll notice that this method simply calls the previous method that you added. So, why not cut out the middleman? You can simply change the method you added previously to be an action instead, like this:

```
@IBAction func startNewGame() {  
    score = 0  
    round = 0  
    startNewRound()  
}
```

You could follow either of the above approaches since both are equally valid. Having less code means there's less stuff to maintain (and less of a chance of screwing something up). Sometimes, there could also be legitimate reasons for having a separate action method which calls your own method, but in this particular case, it's better to keep things simple.

Just to keep things consistent, in `viewDidLoad()` you should replace the call to `startNewRound()` with `startNewGame()`.

Because `score` and `round` are already 0 when the app starts, it won't really make any difference to how the app works, but it does make the intention of the source code clearer. If you wonder whether you can call an `IBAction` method directly instead of hooking it up to an action in the storyboard, yes, you certainly can do so.

- Make this change:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    startNewGame()          // this line changed  
}
```

Connect the outlet

Finally, you need to connect the Start Over button to the action method.

- Open the storyboard and Control-drag from the **Start Over** button to View controller. Let go of the mouse button and pick **startNewGame** from the pop-up if you opted to have `startNewGame()` as the action method. Otherwise, pick the name of your action method .

That connects the button's Touch Up Inside event to the action you have just defined.

- Run the app and play a few rounds. Press Start Over and the game puts you back at square one.

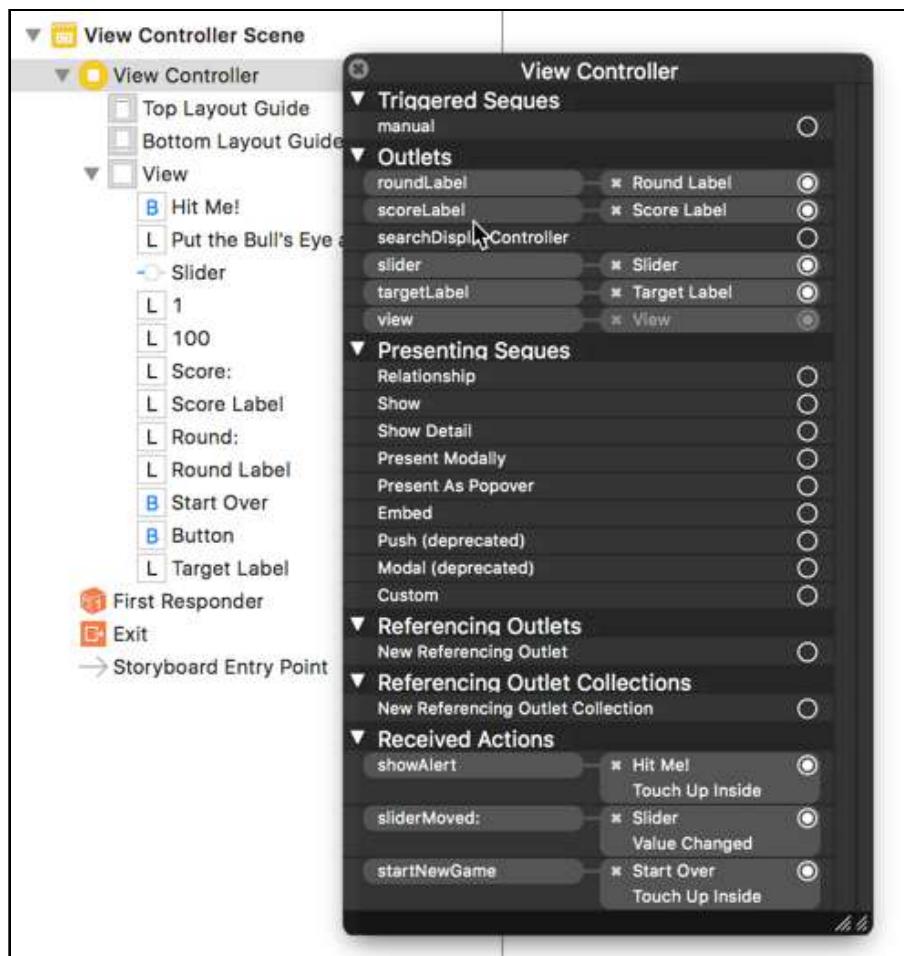
Tip: If you're losing track of what button or label is connected to what method, you can click on View controller in the storyboard to see all the connections that you have made so far.

You can either right-click on View controller to get a pop-up, or simply view the connections in the **Connections inspector**.

This shows all the connections for the view controller.

Now your game is pretty polished and your task list has gotten really short.

You can find the project files for the current version of the app under **18-Polish** in the Source Code folder.



All the connections from View Controller to the other objects

19

Chapter 19: The New Look

By Eli Ganim

Bullseye is looking good! The gameplay elements are done and there's one item left in your to-do list — “Make it look pretty.”

You have to admit the game still doesn't look great. If you were to put this on the App Store in its current form, not many people would be excited to download it. Fortunately, iOS makes it easy for you to create good-looking apps, so let's give *Bullseye* a makeover and add some visual flair.

This chapter covers the following:

- **Landscape orientation revisited:** Project changes to make landscape orientation support work better.
- **Spice up the graphics:** Replace the app UI with custom graphics to give it a more polished look.
- **The about screen:** Add an about screen to the app and make it look spiffy.



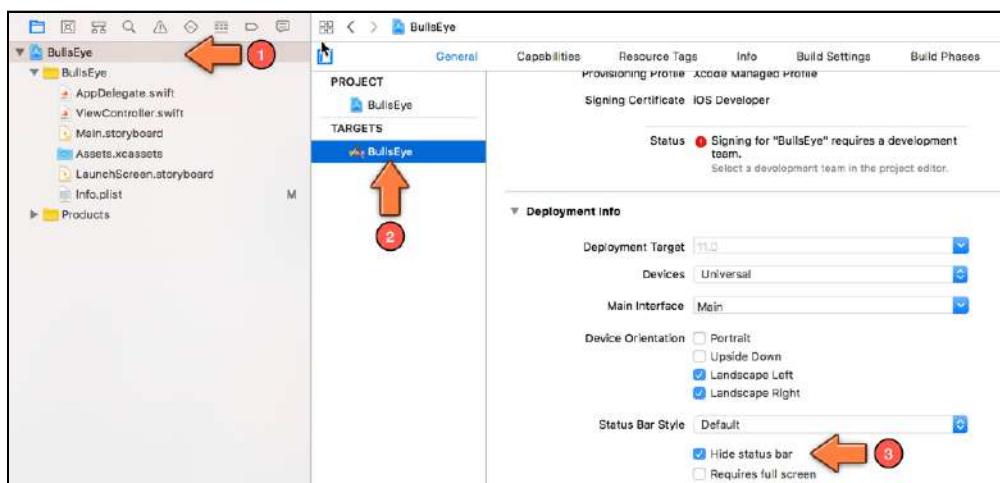
Landscape orientation revisited

Just like you did in the SwiftUI version of Bullseye - you'll need to hide the status bar.

Apps in landscape mode do not display the iPhone status bar, unless you tell them to. That's great for your app — games require a more immersive experience and the status bar detracts from that.

- Go to the **Project Settings** screen and scroll down to **Deployment Info**. Under **Status Bar Style**, check **Hide status bar**.

This will ensure that the status bar is hidden during application launch.

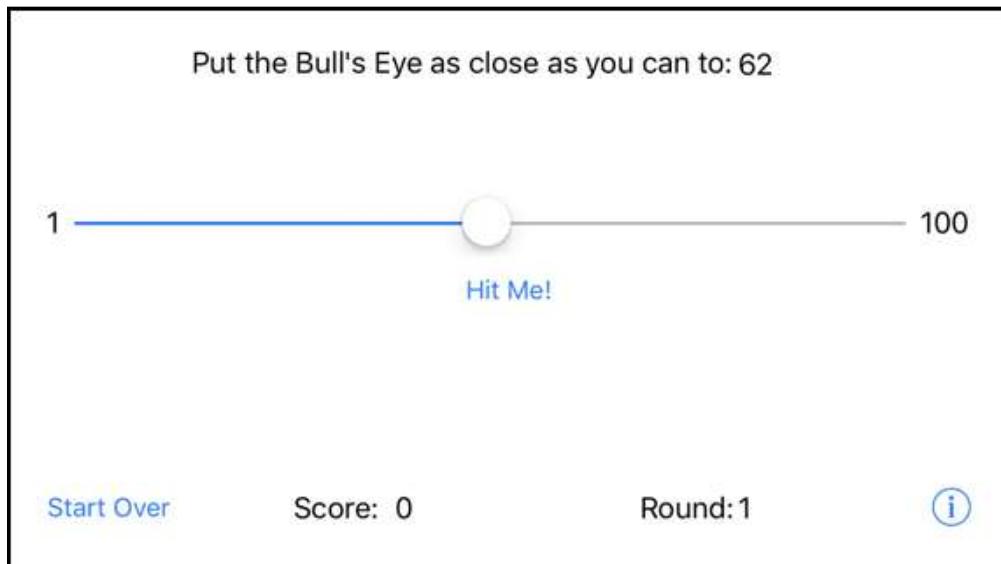


Hiding the status bar when the app launches

- That's it. Run the app and you'll see that the status bar is history.

Spicing up the graphics

Getting rid of the status bar is only the first step. You want to go from this:



Yawn...

To something that's more like this:



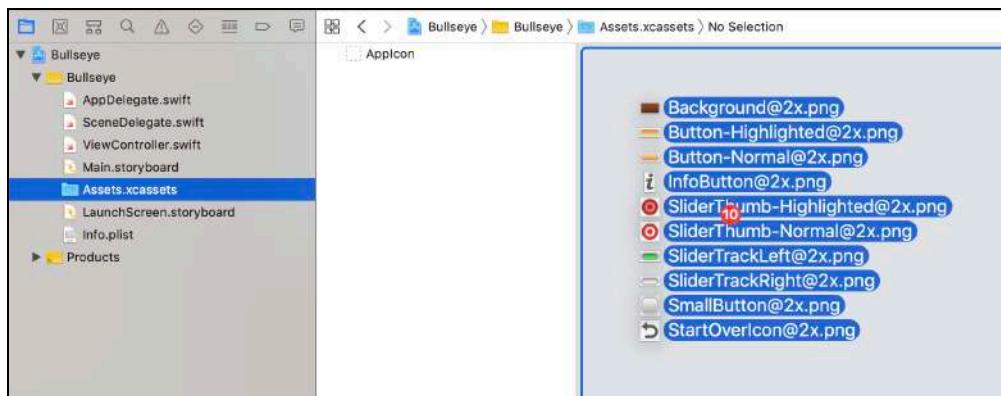
The actual controls won't change. You'll simply be using images to smarten up their look, and you will also adjust the colors and typefaces.

You can put an image in the background, on the buttons, and even on the slider, to customize the appearance of each. The images you use should generally be in PNG format, though JPG files would work too.

Adding the image assets

Just like you did before, you'll need to add the image assets to the project.

- Open the **Assets.xcassets** file.
- Drag the files in the provided **Images** folder to the project.



Choose Import to put existing images into the asset catalog

Putting up the wallpaper

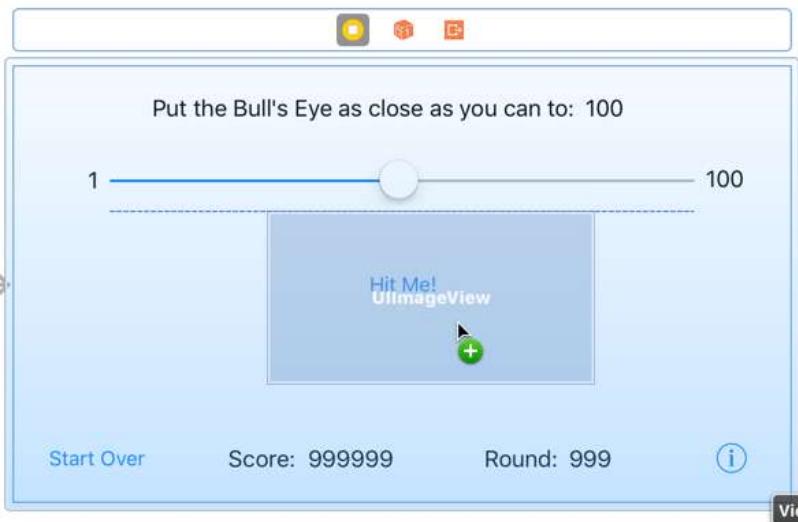
You'll begin by changing the drab white background in *Bullseye* to something more fancy.

- Open **Main.storyboard**, open the **Library** panel (via the top toolbar) and locate an **Image View**.



The Image View control in the Objects Library

- Drag the image view on top of the existing user interface. It doesn't really matter where you put it, as long as it's inside the Bullseye view controller.



Dragging the Image View into the view controller

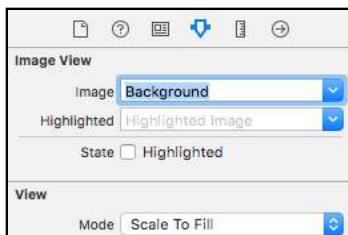
- With the image view still selected, go to the **Size inspector** (that's the one next to the Attributes inspector) and set X and Y to 0, Width to 667 and Height to 375.

This will make the image view cover the entire screen.

- Go to the **Attributes inspector** for the image view. At the top there is an option named **Image**. Click the downward arrow and choose **Background** from the list.

► Change Mode to Scale To Fill

This will put the image named “Background” from the asset catalog into the image view.



Setting the background image on the Image View

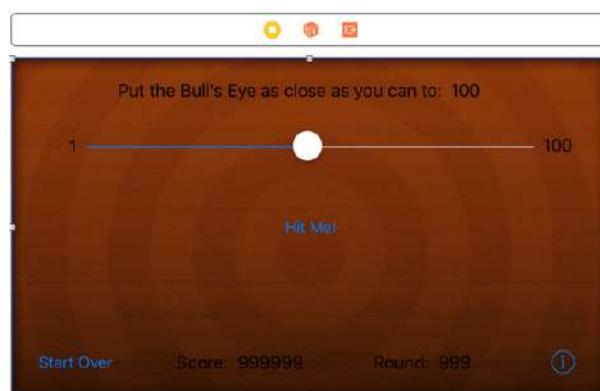
There is only one problem: the image now covers all the other controls. There is an easy fix for that; you have to move the image view behind the other views.

- With the image view selected, in the **Editor** menu in Xcode’s menu bar at the top of the screen, choose **Arrange** ▶ **Send to Back**.

Sometimes Xcode gives you a hard time with this and you might not see the Send to Back item enabled. If so, try de-selecting the Image View and then selecting it again. Now the menu item should be available.

Alternatively, pick up the image view in the Document Outline and drag it to the top of the list of views, just below Safe Area. The items in the Document Outline view are listed so that the backmost item is at the top of the list and the frontmost one is at the bottom.

Your interface should now look something like this:



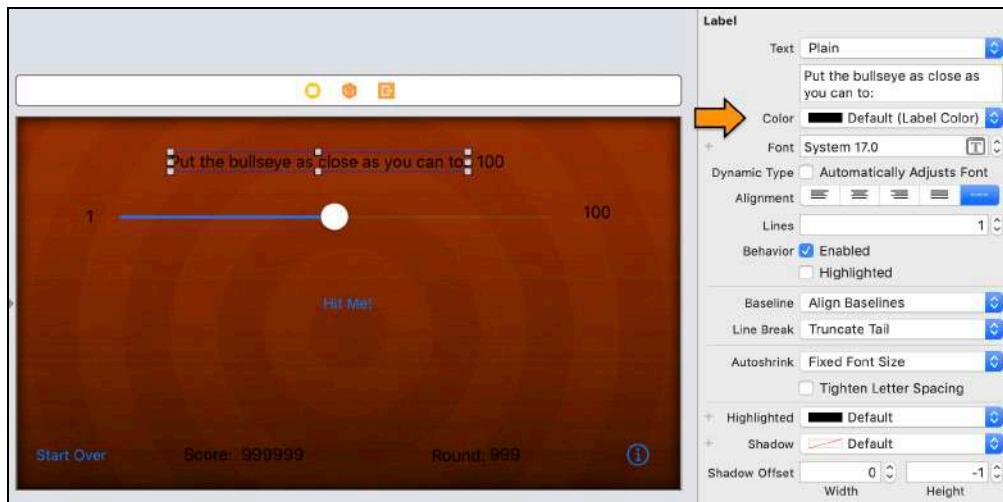
The game with the new background image

That takes care of the background. Run the app and marvel at the new graphics.

Changing the labels

Because the background image is quite dark, the black text labels have become hard to read. Fortunately, Interface Builder lets you change label color. While you're at it, you might change the font as well.

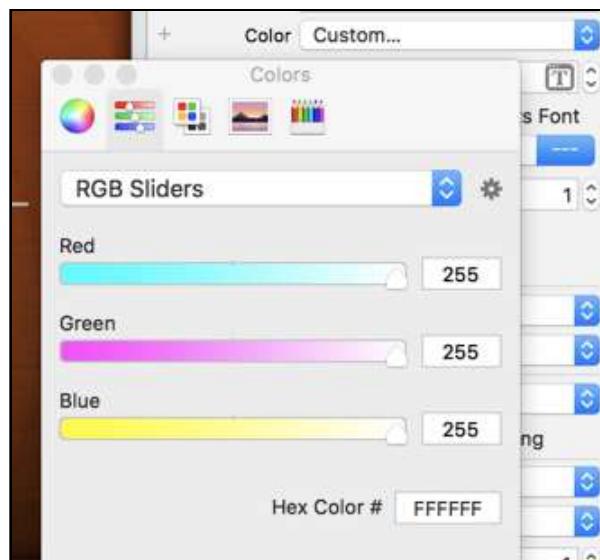
- Still in the storyboard, select the label at the top, open the **Attributes inspector** and click on the **Color** item to show a dropdown for color values. Select **Custom...** at the bottom of the list.



Setting the text color on the label

This opens the Color Picker, which has several ways to select colors. You'll use the sliders (second tab).

If all you see is a gray scale slider, then select **RGB Sliders** from the picker at the top.

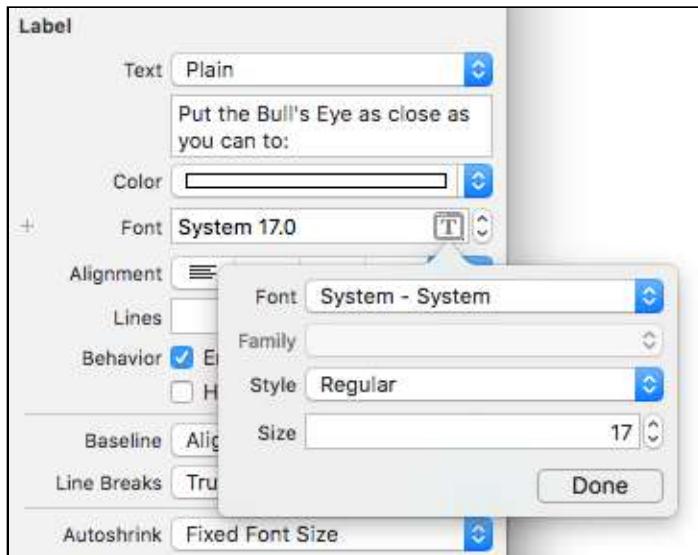


The Color Picker

- Pick a pure white color, Red: 255, Green: 255, Blue: 255, Opacity: 100%. Alternatively, you can simply pick **White Color** from the initial dropdown instead of opening the Color Picker at all, but it's good to know that the Color Picker is there in case you want to do custom colors.
- Click on the **Shadow** item from the Attributes inspector. This lets you add a subtle shadow to the label. By default this color is transparent (also known as "Clear Color") so you won't see the shadow. Using the Color Picker, choose a pure black color that is half transparent, Red: 0, Green: 0, Blue: 0, Opacity: 50%.
- Change the **Shadow Offset** to Width: 0, Height: 1. This puts the shadow below the label.

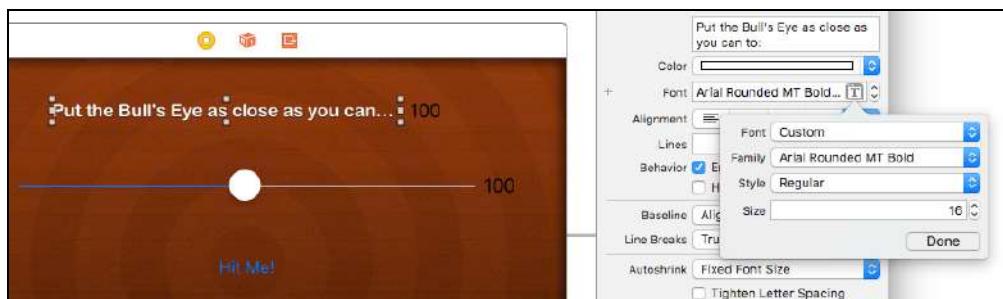
The shadow you've chosen is very subtle. If you're not sure that it's actually visible, then toggle the height offset between 1 and 0 a few times. Look closely and you should be able to see the difference. As I said, it's very subtle.
- Click on the [T] icon of the **Font** attribute. This opens the Font Picker.

By default, the System font is selected. That uses whatever is the standard system font for the user's device. The system font is nice enough but we want something more exciting for this game.



Font picker with the System font

- Choose **Font: Custom**. That enables the Family field. Choose **Family: Arial Rounded MT Bold**. Set the Size to 16.



Setting the label's font

- The label also has an attribute **Autoshrink**. Make sure this is set to **Fixed Font Size**.

If enabled, Autoshrink will dynamically change the size of the font if the text is larger than will fit into the label. That is useful in certain apps, but not in this one. Instead, you'll change the size of the label to fit the text rather than the other way around.

- With the label selected, press **⌘=** on your keyboard, or choose **Size to Fit Content** from the **Editor** menu.

If the Size to Fit Content menu item is disabled, then de-select the label and select it again. Sometimes Xcode gets confused about what is selected. Poor thing.

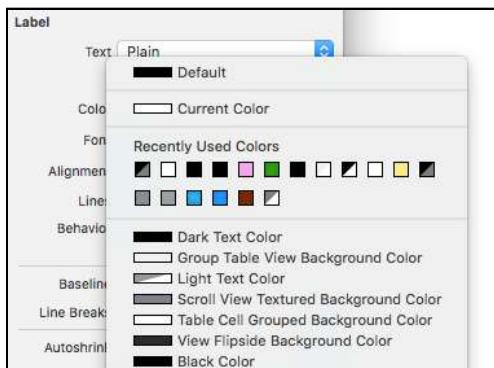
The label will now become slightly larger or smaller so that it fits snugly around the text. If the text got cut off when you changed the font, now all the text will show again.

You don't have to set these properties for the other labels one by one; that would be a big chore. You can speed up the process by selecting multiple labels and then applying these changes to that entire selection.

- Click on the **Score:** label to select it. Hold **⌘** and click on the **Round:** label. Now both labels will be selected. Repeat what you did above for these labels:

- Set Color to pure white, 100% opaque.
- Set Shadow to pure black, 50% opaque.
- Set Shadow Offset to width 0, height 1.
- Set Font to Arial Rounded MT Bold, size 16.
- Make sure Autoshrink is set to Fixed Font Size.

Tip: Xcode is smart enough to remember the colors you have used recently. Instead of going into the Color Picker all the time, you can simply choose a color from the Recently Used Colors menu which is part of the dropdown you get when you click on any color option:

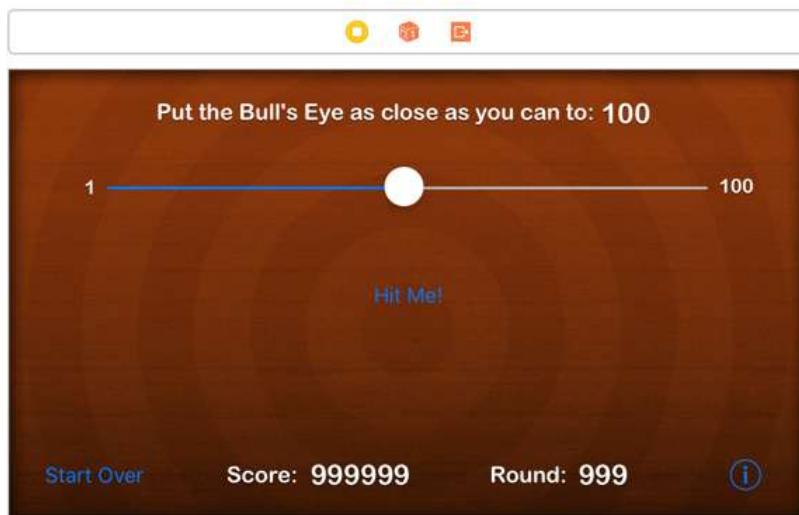


Quick access to recently used colors and several handy presets

Exercise: You still have a few labels to go. Repeat what you just did for the other labels. They should all become white, have the same shadow and have the same font. However, the two labels on either side of the slider (1 and 100) will have font size 14, while the other labels (the ones that will hold the target value, the score and the round number) will have font size 20 so they stand out more.

Because you've changed the sizes of some of the labels, your carefully constructed layout may have been messed up a bit. You may want to clean it up a little.

At this point, the game screen should look something like this:



What the storyboard looks like after styling the labels

All right, it's starting to look like something now. By the way, feel free to experiment with the fonts and colors. If you want to make it look completely different, then go right ahead. It's your app!

The buttons

Changing the look of the buttons works very much the same way.

- Select the **Hit Me!** button. In the **Size inspector** set its Width to 100 and its Height to 37.
- Center the position of the button on the inner circle of the background image.

- Go to the **Attributes inspector**. Change **Type** from System to **Custom**.

A “system” button just has a label and no border. By making it a custom button, you can style it any way you wish.

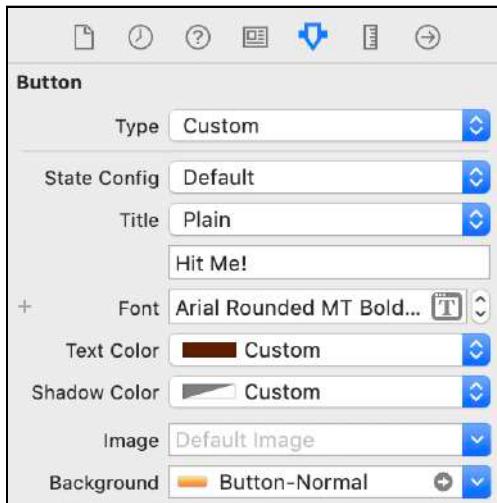
- Still in the **Attributes inspector**, press the arrow on the **Background** field and choose **Button-Normal** from the list.
- Set the **Font** to **Arial Rounded MT Bold**, size 20.
- Set the **Text Color** to red: 96, green: 30, blue: 0, opacity: 100%. This is a dark brown color.
- Set the **Shadow Color** to pure white, 50% opacity. The shadow offset should be Width 0, Height 1.

Blending in

Setting the opacity to anything less than 100% will make the color slightly transparent (with opacity of 0% being fully transparent). Partial transparency makes the color blend in with the background and makes it appear softer.

Try setting the shadow color to 100% opaque pure white and notice the difference.

This finishes the setup for the Hit Me! button in its “default” state:

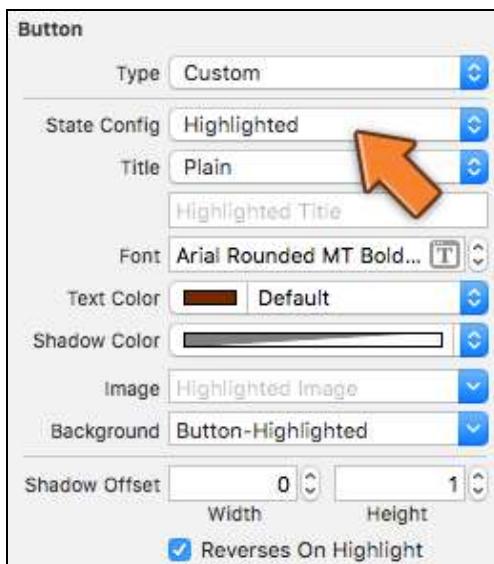


The attributes for the Hit Me! button in the default state

Buttons can have more than one state. When you tap a button and hold it down, it should appear “pressed down” to let you know that the button will be activated when you lift your finger. This is known as the *highlighted* state and is an important visual cue to the user.

- With the button still selected, click the **State Config** setting and pick **Highlighted** from the menu. Now the attributes in this section reflect the highlighted state of the button.
- In the **Background** field, select **Button-Highlighted**.
- Make sure the highlighted **Text Color** is the same color as before (red 96, green 30, blue 0, or simply pick it from the Recently Used Colors menu). Change the **Shadow Color** to half-transparent white again.
- Check the **Reverses On Highlight** option. This will give the appearance of the label being pressed down when the user taps the button.

You could change the other properties too, but don’t get too carried away. The highlight effect should not be too jarring.



The attributes for the highlighted Hit Me! button

To test the highlighted look of the button in Interface Builder you can toggle the **Highlighted** box in the **Control** section, but make sure to turn it off again or the button will initially appear highlighted when the screen is shown.

That's it for the Hit Me! button. Styling the Start Over button is very similar, except you will replace its title text with an icon.

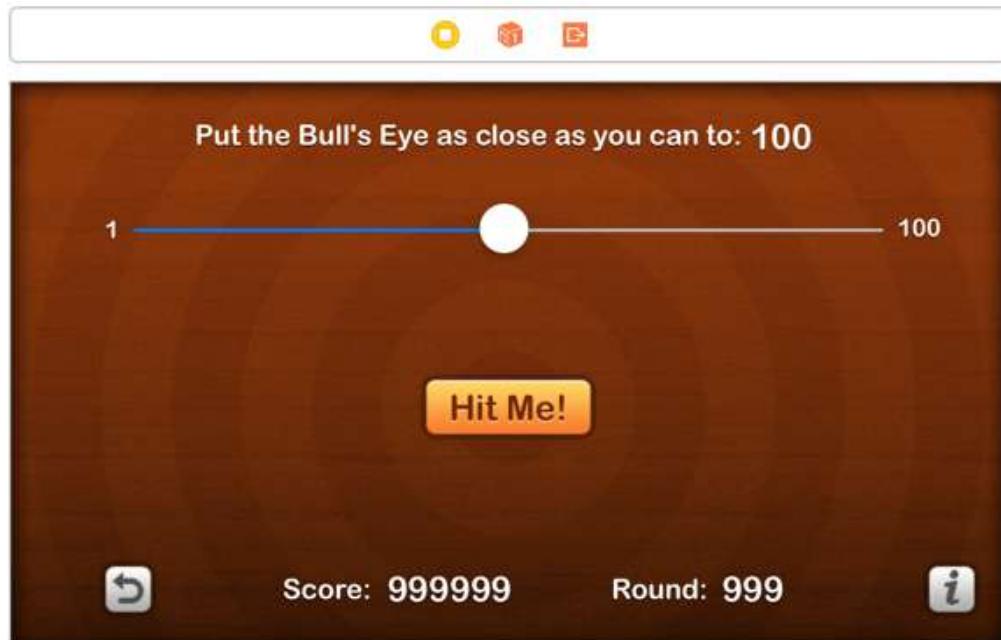
► Select the **Start Over** button and change the following attributes:

- Set Type to Custom.
- Remove the text "Start Over" from the button.
- For Image choose **StartOverIcon**.
- For Background choose **SmallButton**.
- Set Width and Height to 32.

You won't set a highlighted state on this button — let UIKit take care of this. If you don't specify a different image for the highlighted state, UIKit will automatically darken the button to indicate that it is pressed.

► Make the same changes to the ⓘ button, but this time choose **InfoButton** for the image.

The user interface is almost done. Only the slider is left...



Almost done!

The slider

Unfortunately, you can only customize the slider a little bit in Interface Builder. For the more advanced customization that this game needs – putting your own images on the thumb and the track – you have to resort to writing code.

Do note that everything you've done so far in Interface Builder you could also have done in code. Setting the color on a button, for example, can be done by sending the `setTitleColor()` message to the button. (You would normally do this in `viewDidLoad()`.)

However, I find that doing visual design work is much easier and quicker in a visual editor such as Interface Builder than writing the equivalent source code. But for the slider you have no choice.

► Go to `ViewController.swift`, and add the following to `viewDidLoad()`:

```
let thumbImageNormal = UIImage(named: "SliderThumb-Normal")!
slider.setThumbImage(thumbImageNormal, for: .normal)

let thumbImageHighlighted = UIImage(named: "SliderThumb-
Highlighted")!
slider.setThumbImage(thumbImageHighlighted, for: .highlighted)

let insets = UIEdgeInsets(top: 0, left: 14, bottom: 0, right:
14)

let trackLeftImage = UIImage(named: "SliderTrackLeft")!
let trackLeftResizable =
    trackLeftImage.resizableImage(withCapInsets:
insets)
slider.setMinimumTrackImage(trackLeftResizable, for: .normal)

let trackRightImage = UIImage(named: "SliderTrackRight")!
let trackRightResizable =
    trackRightImage.resizableImage(withCapInsets:
insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

This sets four images on the slider: two for the thumb and two for the track. (And if you're wondering what the “thumb” is, that's the little circle in the center of the slider, the one that you drag around to set the slider value.)

The thumb works like a button so it gets an image for the normal (un-pressed) state and one for the highlighted state.

The slider uses different images for the track on the left of the thumb (green) and the track to the right of the thumb (gray).

- Run the app. You have to admit it looks fantastic now!



The game with the customized slider graphics

To .png or not to .png

If you recall, the images that you imported into the asset catalog had filenames like **SliderThumb-Normal@2x.png** and so on.

When you create a `UIImage` object, you don't use the original filename but the name that is listed in the asset catalog, **SliderThumb-Normal**.

That means you can leave off the @2x bit and the **.png** file extension.

The About screen

Just like in the previous version of Bullseye, you'll add an About screen to the app. This new screen contains a *text view* with the gameplay rules and a button to close the screen.

Most apps have more than one screen, even very simple games. So, this is as good a time as any to learn how to add additional screens to your apps.

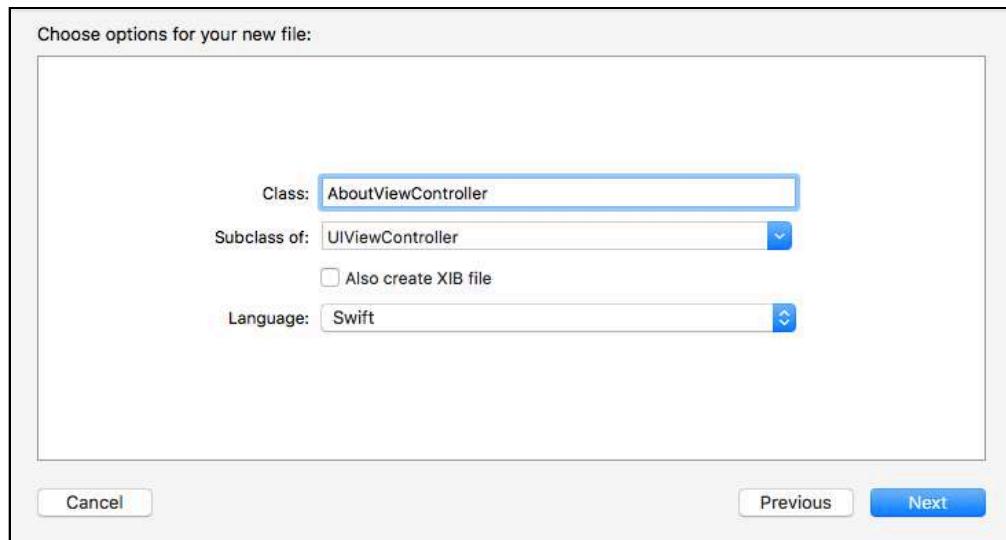
In UIKit each screen in your app will have its own view controller. If you think "screen," think "view controller."

Xcode automatically created the main `ViewController` object for you, but you'll have to create the view controller for the About screen yourself.

Adding a new view controller

► Go to Xcode's **File** menu and choose **New ▶ File...** In the window that pops up, choose the **Cocoa Touch Class** template (if you don't see it then make sure **iOS** is selected at the top).

Click **Next**. Xcode gives you some options to fill out:



The options for the new file

Choose the following:

- Class: **AboutViewController**.
- Subclass of: **UIViewController**.
- Also create XIB file: Leave this box unchecked.
- Language: **Swift**.

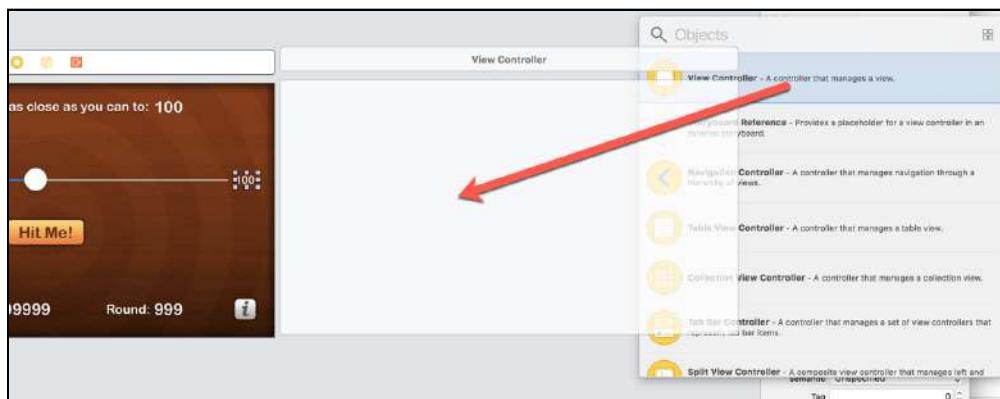
Click **Next** and then **Create**.

Xcode will create a new file and add it to your project. As you might have guessed, the new file is **AboutViewController.swift**.

Designing the view controller in Interface Builder

To design this new view controller, you need to pay a visit to Interface Builder.

- Open **Main.storyboard**. There is no scene representing the About view controller in the storyboard yet. So, you'll have to add this first.
- From the **Library**, choose **View Controller** and drag it on to the canvas, to the right of the main View controller.

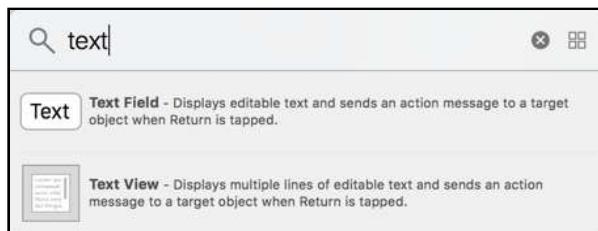


Dragging a new View Controller from the Objects Library

This new view controller is totally blank. You may need to rearrange the storyboard so that the two view controllers don't overlap. Interface Builder isn't very tidy about where it puts things.

- Drag a new **Button** on to the screen and give it the title **Close**. Put it somewhere around the bottom center of the view (use the blue guidelines to help with positioning).
- Drag a **Text View** on to the view and make it cover most of the space above the button.

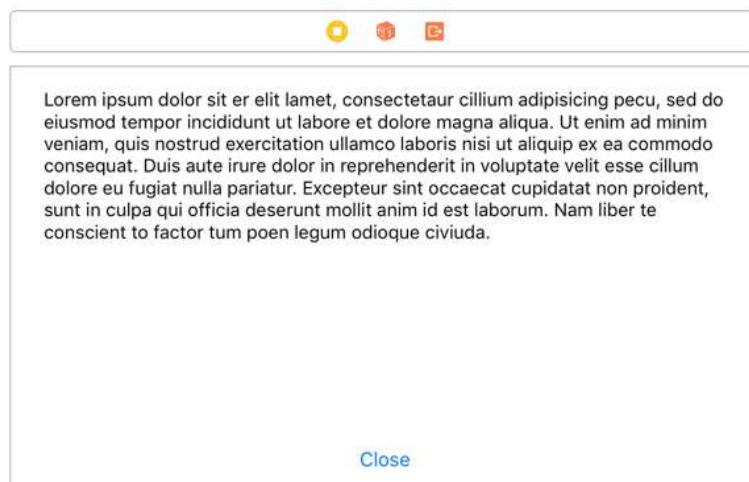
You can find these components in the Library. If you don't feel like scrolling, you can filter the components by typing in the field at the top:



Searching for text components

Note that there is also a Text Field, which is a single-line text component — that's not what you want. You're looking for Text View, which can contain multiple lines of text.

After dragging both the text view and the button on to the canvas, it should look something like this:



The About screen in the storyboard

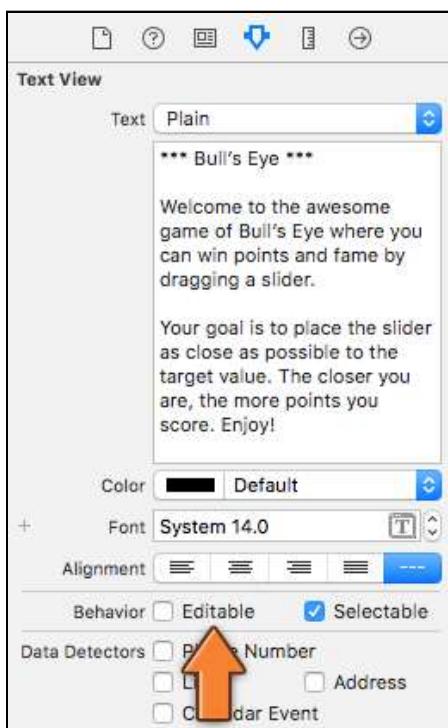
- Double-click the text view to edit its content. By default, the Text View contains a bunch of Latin placeholder text (also known as “Lorem Ipsum”). Replace the text with this:

```
*** Bullseye ***  
  
Welcome to the awesome game of Bullseye where you can win points  
and fame by dragging a slider.  
  
Your goal is to place the slider as close as possible to the  
target value. The closer you are, the more points you score.  
Enjoy!
```

You can also enter that text into the Attributes inspector's **Text** property for the text view if you find that easier.

- Make sure to uncheck the **Editable** checkbox in the Attribute Inspector. Otherwise, the user can actually type into the text view and you don't want that.

The design of the screen is done for now.



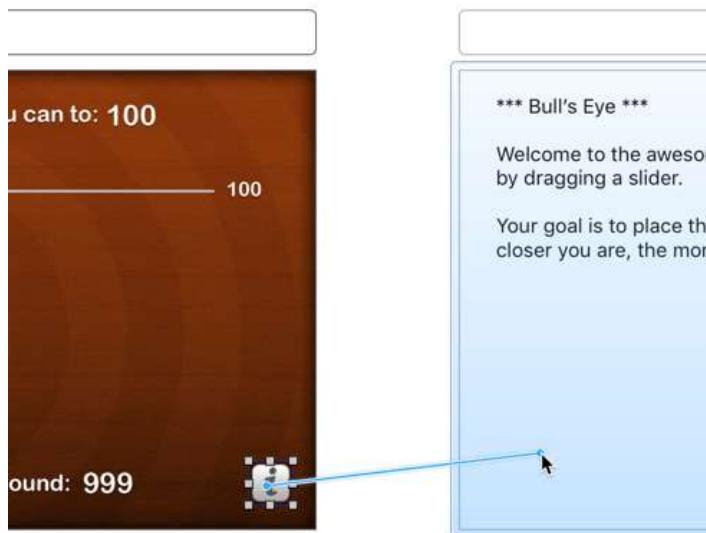
The Attributes inspector for the text view

Showing the new view controller

So how do you open this new About screen when the user presses the ⓘ button?

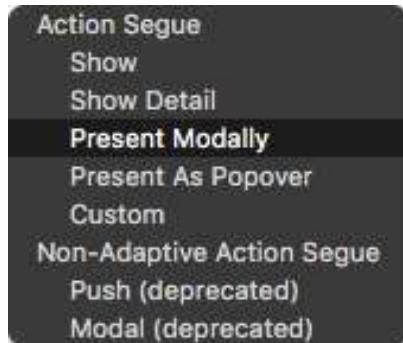
Storyboards have a neat trick for this: *segues* (pronounced “seg-way” like the silly scooters). A segue is a transition from one screen to another. They are really easy to add.

- Click the ⓘ button in the **View controller** to select it. Then hold down **Control** and drag over to the **About** screen.



Control-drag from one view controller to another to make a segue

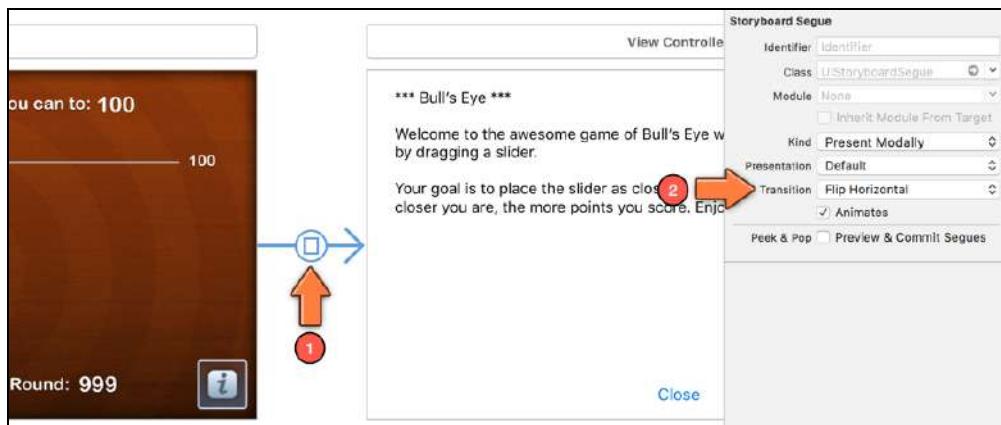
- Let go of the mouse button and a pop-up appears with several options. Choose **Present Modally**.



Choosing the type of segue to create

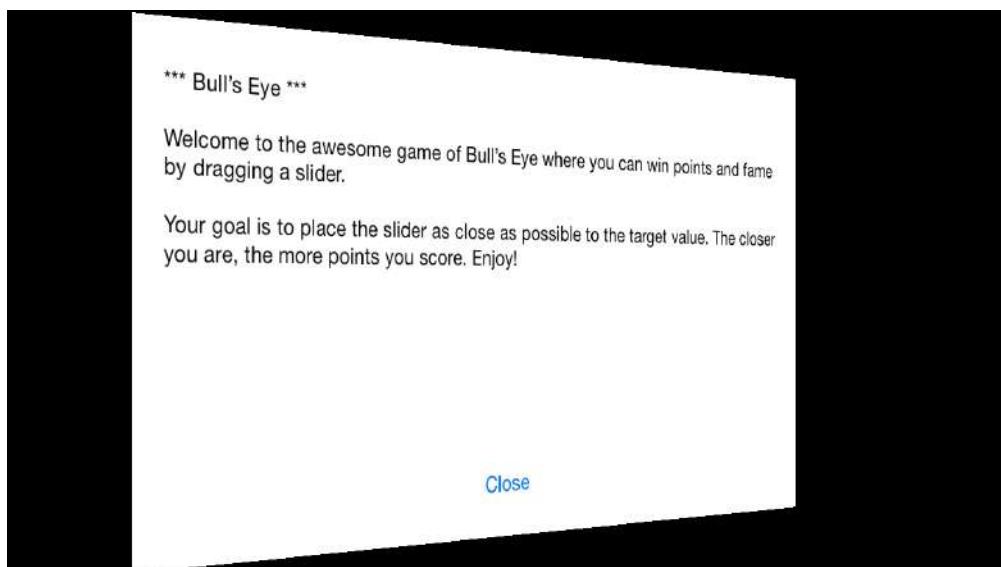
Now an arrow will appear between the two screens. This arrow represents the segue from the main scene to the About scene.

- Click the arrow to select it. Segues also have attributes. In the **Attributes inspector**, choose **Transition**, **Flip Horizontal**. That is the animation that UIKit will use to move between these screens.



Changing the attributes for the segue

- Now you can run the app. Press the ⓘ button to see the new screen.



The About screen appears with a flip animation

The About screen should appear with a neat animation. Good, that seems to work.

Dismissing the About view controller

Did you notice that there's an obvious issue here? Tapping the Close button seems to have no effect. Once the user enters the About screen they can never leave... that doesn't sound like good user interface design, does it?

The problem with segues is that they only go one way. To close this screen, you have to hook up some code to the Close button. As a budding iOS developer you already know how to do that: use an action method!

This time you will add the action method to `AboutViewController` instead of `ViewController`, because the Close button is part of the About screen, not the main game screen.

- Open `AboutViewController.swift` and replace its contents with the following:

```
import UIKit

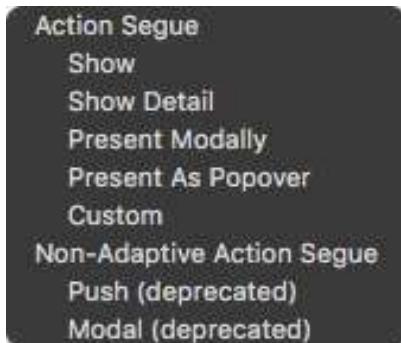
class AboutViewController: UIViewController {
    @IBAction func close() {
        dismiss(animated: true, completion: nil)
    }
}
```

The code in the `close()` action method tells UIKit to close the About screen with an animation.

If you had said `dismiss(animated: false, ...)`, then there would be no page flip and the main screen would instantly reappear. From a user experience perspective, it's often better to show transitions from one screen to another via an animation.

That leaves you with one final step, hooking up the Close button's Touch Up Inside event to this new `close` action.

► Open the storyboard and Control-drag from the **Close** button to the About scene's View Controller. Hmm, strange, the **close** action should be listed in this pop-up, but it isn't. Instead, this is the same pop-up you saw when you made the segue:



The “close” action is not listed in the pop-up

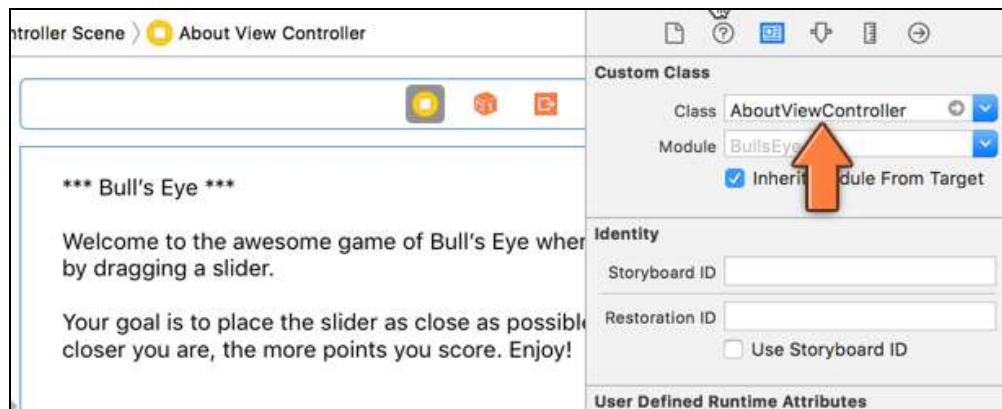
Exercise: Bonus points if you can spot the error. It's a very common – and frustrating! – mistake.

The problem is that this scene in the storyboard doesn't know yet that it is supposed to represent the `AboutViewController`.

Setting the class for a view controller

You first added the `AboutViewController.swift` source file, and then dragged a new view controller on to the storyboard. But, you haven't told the storyboard that the design for this new view controller belongs to `AboutViewController`. That's why in the Document Outline it just says View Controller and not About View controller. That's the design of the screen done for now.► Fortunately, this is easily remedied. In Interface Builder, select the About scene's **View controller** and go to the **Identity inspector** (that's the tab/icon to the left of the Attributes inspector).

- Under **Custom Class**, enter **AboutViewController**.



The Identity inspector for the About screen

Xcode should auto-complete this for you once you type the first few characters. If it doesn't, then double-check that you really have selected the View controller and not one of the views inside it. (The view controller should also have a blue border on the storyboard to indicate it is selected.)

Now you should be able to connect the Close button to the action method.

- Control-drag from the **Close** button to **About View controller** in the Document Outline (or to the yellow circle at the top of the scene in the storyboard). This should be old hat by now. The pop-up menu now does have an option for the **close** action (under Sent Events). Connect the button to that action.
- Run the app again. You should now be able to return from the About screen.

OK, that does get us a working About screen, but it does look a little plain doesn't it? What if you added some of the design changes you made to the main screen?

Exercise: Add a background image to the About screen. Also, change the Close button on the About screen to look like the Hit Me! button and play around with the Text View properties in the Attribute Inspector. **Tip:** The background color of the TextView is white by default and will hide whatever is behind it.

You should be able to do this by yourself now. Piece of cake! Refer back to the instructions for the main screen if you get stuck.

When you are done, you should have an About screen which looks something like this:



The new and improved About screen

That looks good, but it could be better. So how do you improve upon it?

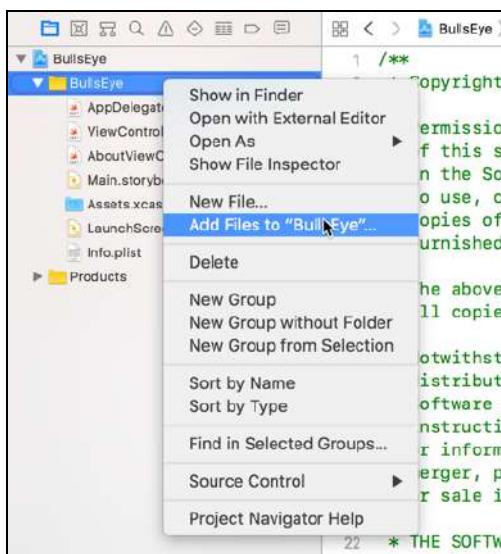
Using a web view for HTML content

- Now select the **text view** and press the **Delete** key on your keyboard. (Yep, you're throwing it away, and after all those changes, too! But don't grieve for the Text View too much, you'll replace it with something better.)
- Put a **WebKit View** in its place (as always, you can find this view in the Objects Library). There are two web view options — an older Web View, which is deprecated, or ready to be retired, and the WebKit View. Make sure that you select the WebKit View.

This view can show web pages. All you have to do is give it the URL to a web site or the name of a file to load. The WebKit View object is named `WKWebView`.

For this app, you will make it display a static HTML page from the application bundle, so it won't actually have to go online and download anything.

- Go to the **Project navigator** and right-click on the **Bullseye** group (the yellow folder). From the menu, choose **Add Files to “Bullseye”...**



Using the right-click menu to add existing files to the project

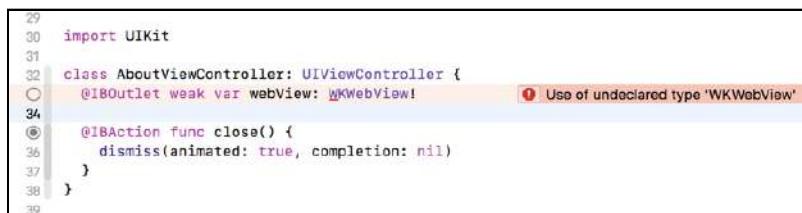
- In the file picker, select the **Bullseye.html** file from the Resources folder. This is an HTML5 document that contains the gameplay instructions.

Make sure that **Copy items if needed** is selected and that under **Add to targets**, there is a checkmark in front of **Bullseye**. (If you don't see these options, click the Options button at the bottom of the dialog.)

- Press **Add** to add the HTML file to the project.
- In **AboutViewController.swift**, add an outlet for the web view:

```
class AboutViewController: UIViewController {
    @IBOutlet weak var webView: WKWebView!
}
```

Xcode will complain soon after you add the above line. The error should look something like this:



Xcode complains about WKWebView

What does this error mean? It means that Xcode, or rather the compiler, does not know what `WKWebView` is.

But how can that be? We selected the component from Xcode's own Objects Library and so it should be supported, right?

The answer to this lies with this line of code at the top of both your view controller source files:

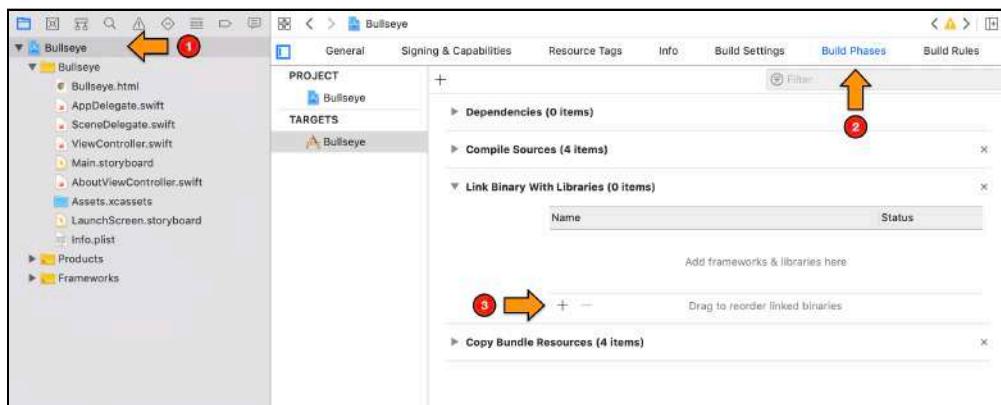
```
import UIKit
```

I'm sure you saw this line and wondered what it was about. That statement tells the compiler that you want to use the objects from a framework named `UIKit`.

Frameworks, or libraries if you prefer, bundle together one or more objects which perform a particular type of task (or tasks). The `UIKit` library provides all the UI components for iOS.

So why does `UIKit` not contain `WKWebView`, you ask? That's because the previously mentioned deprecated `WebView` is the one which is included with `UIKit`. The newer (and improved) `WKWebView` comes from a different framework called `WebKit`.

- Click on your `Bullseye`'s' project file
- Go the the **Build Phases** tab and expand the **Link Binary With Libraries** category.
- Click on the little + button and search for **WebKit.framework** and click **Add**.



Xcode complains about `WKWebView`

Now you have access to all the wonders of the WebKit framework. All that's left to do is actually use it.

- Add the following code at the top of **AboutViewController.swift**, right below the existing `import` statement:

```
import WebKit
```

That tells the compiler that we want to use objects from the WebKit framework and since now the compiler knows about all the objects in the WebKit framework, the Xcode error will go away.

- In the storyboard file, connect the `WKWebView` to this new outlet. The easiest way to do this is to Control-drag from **About View controller** (in the Document Outline) to the **Web View**.
- In **AboutViewController.swift**, add a `viewDidLoad()` implementation:

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

    if let url = Bundle.main.url(forResource: "Bullseye",
                                  withExtension: "html") {
        let request = URLRequest(url: url)
        webView.load(request)
    }
}
```

This displays the HTML file using the web view.

The code first gets the URL (Uniform Resource Locator) for the **Bullseye.html** file in the application bundle. A URL, as you might be familiar with from the Interwebs, is a way to identify the location of a resource, like a web page. Here, the URL provides the location of the HTML file in your application bundle.

It then creates a `URLRequest` using that URL since that's one of the easiest ways to send a load request to the web view.

Finally, the code asks the web view to load the contents specified by the URL request.



- Run the app and press the info button. The About screen should appear with a description of the gameplay rules, this time in the form of an HTML document:



The About screen in all its glory

Well done! You've created an (almost) identical version of the Bullseye game both in SwiftUI and in UIKit. In the next chapters of this section you'll improve the Bullseye game even further, by adding an high-scores screen.

You can find the project files under **19-The New Look** in the Source Code folder.

20

Chapter 20: Table Views

By Eli Ganim

Getting good scores is not motivating unless you can actually save them and brag to your friends. In this chapter you'll learn how to save the high scores and present them.

This is how the screen will look like when you're finished:

The reader of this book	50000
Manda	10000
Joey	5000
Adam	1000
Eli	500

This is how the screen would look at the end of this chapter

This chapter covers the following:

- **Table views and navigation controllers:** A basic introduction to navigation controllers and table views.
- **Add a table view:** Create your first UIKit table view and add a prototype cell to display data.
- **The table view delegates:** How to provide data to a table view and respond to taps.



Table views and navigation controllers

This screen will introduce you to two of the most commonly used UI elements in iOS apps: the table view and the navigation controller.

A **table view** is UIKit's equivalent of SwiftUI's **List**. This component is extremely versatile and the most important one to master in iOS development.

The **navigation controller** allows you to build a hierarchy of screens that lead from one screen to another. It adds a navigation bar at the top with a title and a back button.

In this screen, tapping an entry slides in the screen containing the information about the high score, like who made it and when it was achieved. Navigation controllers and table views are often used together.



The grey bar at the top is the navigation bar. The list of items is the table view.

Adding a table view

As table views are so important, you will start out by examining how they work.

Because smart developers split up the workload into small, simple steps, this is what you're going to do in this chapter:

1. Put a table view on the app's screen.
2. Put data into that table view.
3. Allow the user to tap a row in the table to show when that high score was reached.

Once you have these basics up and running, you'll keep adding new functionality over the next few chapters until you end up with a fully working high scores screen.

Creating a new screen

- Go to Xcode's **File** menu and choose **New ▶ File...**
- Choose the **Cocoa Touch Class** template. Click **Next**. Call the file **HighScoresViewController** and make it a subclass of **UITableViewController**.

A **table view controller** is a special type of view controller that makes working with table views easier.

- Click on **Main.storyboard** to open Interface Builder and drag a new **Table View Controller** from the Objects Library into the storyboard.
- Select the **View Controller** of the new scene you just added. Open the **Identity Inspector** from the right pane and update the class name to **HighScoresViewController**.

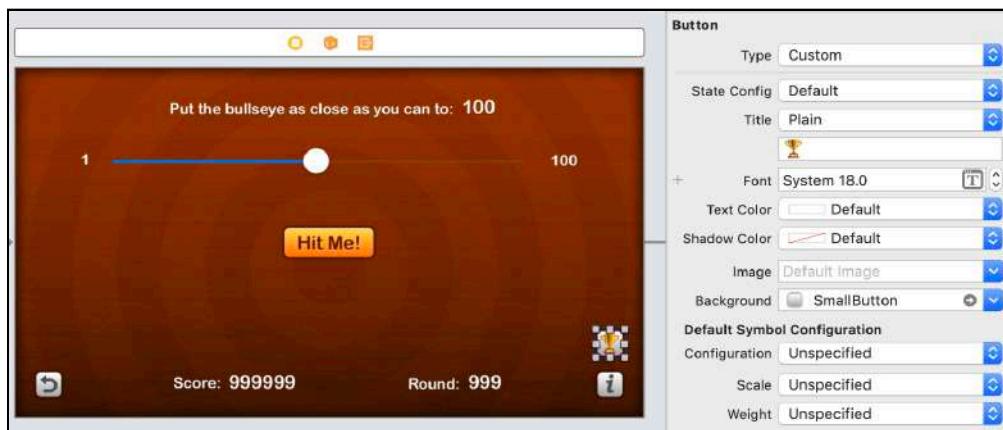
The name of the scene in the Document Outline on the left should change to “High Scores View Controller Scene”. As its name implies, and as you can see in the storyboard, the view controller contains a Table View object. We'll go into the difference between controllers and views soon, but for now, remember that the controller is the whole screen while the table view is the object that actually draws the list.



Connecting the new view controller

Right now there's no way to reach the new screen you just added. In order to fix that, you'll add a new button to the main screen of the game.

- Open the storyboard and add a new button to the main screen, just above the *about* button.
- Use this settings for the button. **Type:** Custom; **Background:** SmallButton.
- Change the text inside the button to be a trophy symbol: 🏆. You can find it under **Edit** ▶ **Emoji & Symbols** and then search for the word 'Trophy'.
- Set the button size to be 32x32.



The arrow points at the initial view controller

Now you're going to hook this button up to the high scores screen.

- Click the 🏆 button to select it. Then hold down **Control** and drag over to the **High Scores** screen.
- Let go of the mouse button and a pop-up appears with several options. Choose **Show**.
- Run the app on the Simulator and click on the trophy button.

You should see an empty list. This is the table view. You can drag the list up and down but it doesn't contain any data yet.

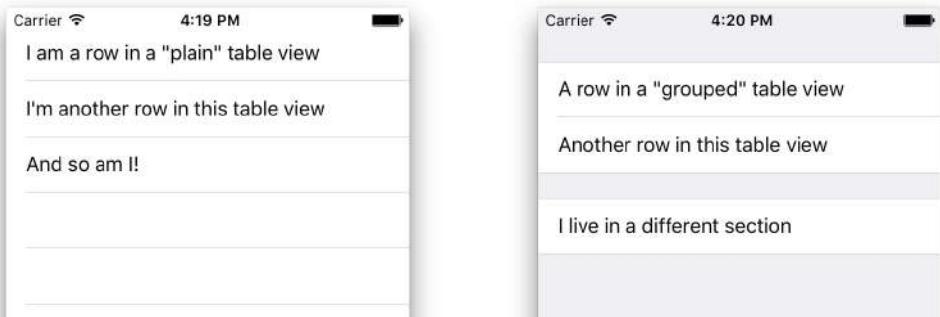
By the way, it doesn't really matter which Simulator you use. Table views resize themselves to fit the dimensions of the device, and the app will work equally well on the small iPhone 8 or the huge iPhone X.

Note: When you build the app, Xcode gives the warning “Prototype table cells must have reuse identifiers.” Don’t worry about this for now, you’ll fix it soon.

The anatomy of a table view

First, let’s talk a bit more about table views. A `UITableView` object displays a list of items.

There are two styles of tables: “plain” and “grouped.” They work mostly the same, but there are a few small differences. The most visible difference is that rows in the grouped style table are placed into boxes (the groups) on a light gray background.



A plain-style table (left) and a grouped table (right)

Note: I’m not sure why it’s named a *table*, because a table is commonly thought of as a spreadsheet-type object that has multiple rows and multiple columns, whereas the `UITableView` only has rows. It’s more of a list than a table, but I guess we’re stuck with the name now. UIKit also provides a `UICollectionView` object that works similar to a `UITableView` but allows for multiple columns.

The plain style is used for rows that all represent something similar, such as contacts in an address book where each row contains the name of one person.

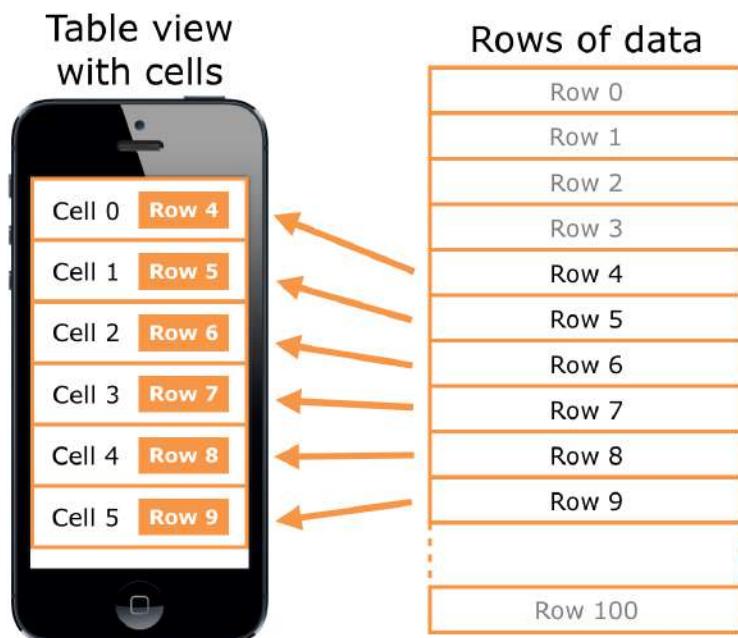
The grouped style is used when the items in the list can be organized by a particular attribute, like book categories for a list of books. The grouped style table could also be used to show related information which doesn't necessarily have to stand together — like the address information, contact information, and e-mail information for a contact.

You will use both table styles in the upcoming chapters.

The data for a table comes in the form of **rows**. You can potentially have many rows (even tens of thousands) but that kind of design isn't recommended. Most users will find it incredibly annoying to scroll through ten thousand rows to find the one they want. And who can blame them?

Tables display their data in **cells**. A cell is related to a row but it's not exactly the same. A cell is a view that shows a row of data that happens to be visible at that moment. If your table can show 10 rows at a time on the screen, then it only has 10 cells, even though there may be hundreds of rows of actual data.

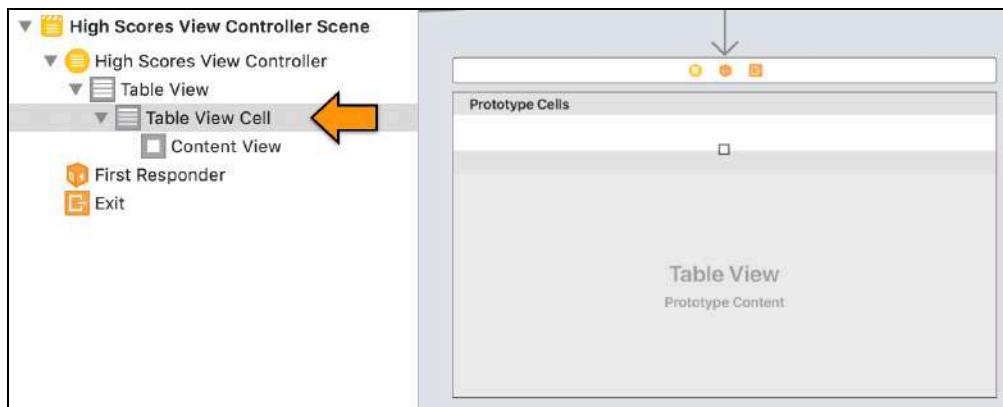
Whenever a row scrolls off the screen and becomes invisible, its cell will be re-used for a new row that becomes visible.



Cells display the contents of rows

Adding a prototype cell

Xcode has a very handy feature named *prototype cells* that lets you design your cells visually in Interface Builder.► Open the storyboard and click the empty cell (the white row below the Prototype Cells label) to select it.

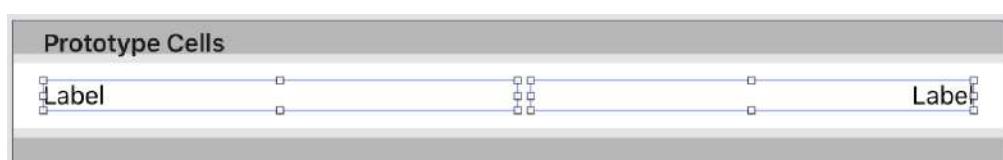


Selecting the prototype cell

Sometimes it can be hard to see exactly what is selected, so keep an eye on the Document Outline to make sure you've picked the right thing. (Or use the Document Outline to select the cell directly.)

- Drag a **Label** from the Objects Library on to the white area in the table view representing the cell. Make sure the label spans from the left edge (with a small margin) until the middle of the cell.
- Drag another **Label** and place it next to the previous label, so it spans from its right edge to the cell's right edge.
- In the **Attributes inspector**, change the alignment to **Right**.

The result should look similar to this:



Adding the label to the prototype cell

Note: If you simply drag the label on to the table view, it might not work. You need to drag the label on to the cell itself. You can check where the label ended up using the Document Outline. It has to be inside the Content View for the table view cell.

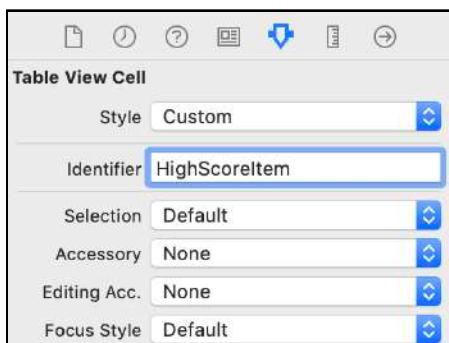
You also need to set a *reuse identifier* on the cell. This is an internal name that the table view uses to find free cells to reuse when rows scroll off the screen and new rows must become visible.

The table needs to assign cells for those new rows, and recycling existing cells is more efficient than creating new cells. This technique is what makes table views scroll smoothly.

Reuse identifiers are also important for when you want to display different types of cells in the same table. For example, one type of cell could have an image and a label and another could have a label and a button. You would give each cell type its own identifier, so the table view can assign the right cell for a given row type.

This screen has only one type of cell but you still need to give it an identifier.

- Type **HighScoreItem** into the Table View Cell's **Identifier** field (you can find this in the **Attributes inspector**).



Giving the table view cell a reuse identifier

Compiler warnings

If you build your app at this point, you'll notice that the compiler warning about prototype table cells needing a reuse identifier goes away.

But... you've got a new warning — one about views without any layout constraints clipping or overlapping other views. Sounds familiar?

Yes, this is the same warning you saw previously when you had views without any Auto Layout constraints! And you know how to find the affected views now, right?

- In the storyboard, click on the yellow warning circle for the table view to see the list of views with issues. It is the new label you just added to the prototype table cell.

That's simple enough to fix, right? Simply select the label, select the **Add New Constraints** icon at the bottom of the Interface Builder window, and add 4 constraints for the left, top, right, and bottom of the label. (You can go with the current defaults as long as you have the label positioned correctly.)

- Run the app and you'll see... nothing — exactly the same as before. The table is still empty.

This is because you only added a cell design to the table, not actual data. Remember that the cell is just the visual representation of the row, not the actual data. To add data to the table, you have to write some code.

The table view delegates

- Switch to **HighScoresViewController.swift** and add the following methods just before the closing bracket at the bottom of the file:

```
// MARK:- Table View Data Source
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
       (withIdentifier: "HighScoreItem",
        for: indexPath)
    )
    return cell
}
```

These methods look a bit more complicated than the ones you've seen in *Bullseye*, but that's because each takes two parameters and returns a value to the caller. Other than that, they work the same way as the methods you've dealt with before.

Protocols

The above two methods are part of `UITableView`'s **data source** protocol.

The data source is the link between your data and the table view. Usually, the view controller plays the role of data source and implements the necessary methods. So, essentially, the view controller is acting as a delegate on behalf of the table view. (This is the delegate pattern that we've talked about before — where an object does some work on behalf of another object.)

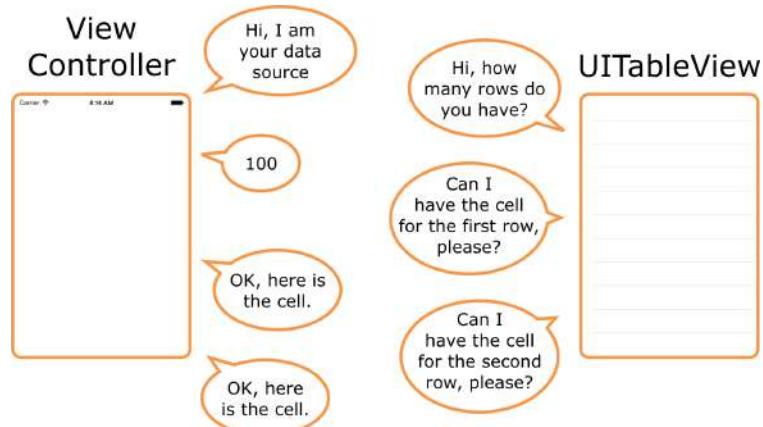
The table view needs to know how many rows of data it has and how it should display each of those rows. But you can't simply dump that data into the table view's lap and be done with it. You don't say: "Dear table view, here are my 100 rows, now go show them on the screen."

Instead, you say to the table view: "This view controller is now your data source. You can ask it questions about the data anytime you feel like it."

Once it is hooked up to a data source – i.e. your view controller – the table view sends a `numberOfRowsInSection` message to find out how many data rows there are.

And when the table view needs to draw a particular row on the screen it sends a `cellForRowAtIndex` message to ask the data source for a cell.

You see this pattern all the time in iOS: one object does something on behalf of another object. In this case, the `HighScoresViewController` works to provide the data to the table view, but only when the table view asks for it.



The dating ritual of a data source and a table view

Your implementation of `tableView(_:numberOfRowsInSection:)` – the first method that you added – returns the value 1. This tells the table view that you have just one row of data.

The return statement is very important in Swift. It allows a method to send data back to its caller. In the case of `tableView(_:numberOfRowsInSection:)`, the caller is the `UITableView` object and it wants to know how many rows are in the table.

The statements inside a method usually perform some kind of computation using instance variables and any data received through the method's parameters. When the method is done, `return` says, “Hey, I'm done. Here is the answer I came up with.” The return value is often called the *result* of the method.

For `tableView(_:numberOfRowsInSection:)` the answer is really simple: there is only one row, so `return 1`. Now that the table view knows it has one row to display, it calls the second method you added – `tableView(_:cellForRowAt:)` – to obtain a cell for that row. This method grabs a copy of the prototype cell and gives that back to the table view, again with a `return` statement.

Inside `tableView(_:cellForRowAt:)` is also where you would normally put the row data into the cell, but the app doesn't have any row data yet.

- Run the app and go to the high scores screen. You'll see there is a single cell in the table:



The table now has one row

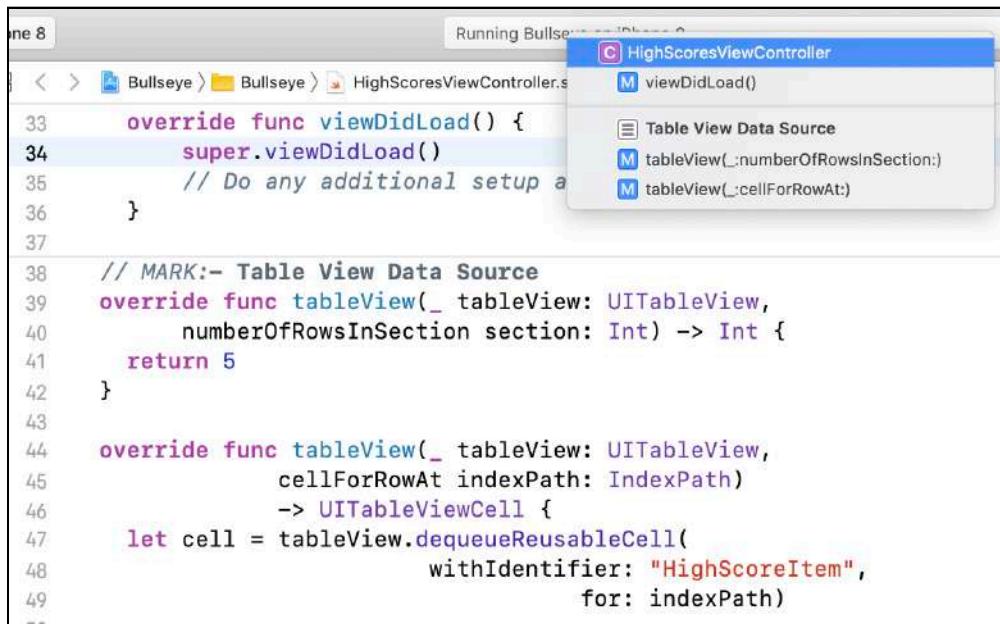
Method signatures

In the above text, you might have noticed some special notation for the method names, like `tableView(_:numberOfRowsInSection:)` or `tableView(_:cellForRowAt:)`. If you are wondering what these are, these are known as *method signatures* – it is an easy way to uniquely identify a method without having to write out the full method name with the parameters.

The method signature identifies where each parameter would be (and the parameter name, where necessary) by separating out the parameters with a colon.

In the method for `tableView(_:numberOfRowsInSection:)` for example, you might notice an underscore for the first parameter — that means, that method does not need to have the parameter name specified when calling the method — it is simply a convenience in Swift where the parameter can generally be inferred from the method name. You might have more questions about this — but we'll come back to that later.

If you are not sure about the signature for a method, take a look at the Xcode **Jump bar** (the tiny toolbar right above the source editor) and click on the last item of the file path elements to get a list of methods (and properties) in the current source file.



The screenshot shows a portion of an Xcode code editor. The main area contains Swift code for a `HighScoresViewController`. The jump bar at the top displays the file path: `Bullseye > Bullseye > HighScoresViewController.swift`. Below the path, the jump bar lists several methods under the class `C HighScoresViewController`:

- `M viewDidLoad()`
- `Table View Data Source`
- `M tableView(_:numberOfRowsInSection:)`
- `M tableView(_:cellForRowAt:)`

The code itself includes implementations for `viewDidLoad()`, `tableView(_:numberOfRowsInSection:)`, and `tableView(_:cellForRowAt:)`.

The Jump Bar shows the method signatures

Also, do note that in the above examples, `tableView` is not the method name — or rather, `tableView` by itself is not the method name. The method name is the `tableView` plus the parameter list — everything up to the closing bracket for the parameter list. That's how you get multiple unique methods such as `tableView(_:numberOfRowsInSection:)` and `tableView(_:cellForRowAt:)` even though they all look as if they are methods called `tableName` — the complete signature uniquely identifies the method.

Special comments

You might have noticed the following line in the code you just added:

```
// MARK:- Table View Data Source
```

If you were wondering what that was for, here's the scoop. Of course, you already know that line is a comment, because the line begins with `//`, but it's not *just* a comment. As the keyword at the beginning of the comment line, `MARK`, indicates, it is a marker. But a marker for what?

It's a marker to organize the code and for you to find a section of code (for example, a set of related methods, like for the table view data source) via the Xcode Jump Bar.

Take a look at the previous screenshot showing the Xcode Jump Bar. Do you notice the separator line in the middle of the list of methods? Do you notice the bolded text title right after? Does that title seem familiar?

The text you provide after the `MARK:` keyword defines how the section title is displayed in the menu. If you put in a hyphen (-), you get a separator line followed by any text after the hyphen as the section title.

If you don't provide a hyphen but provide some text, then you simply get a section title but no separator. If you provide neither, then you just get a section icon with no text and no separator. (Try these out.)

There are a couple of other comment tags besides `MARK:` that you can use in your Swift files. These are `TODO:` and `FIXME:`. The first is generally used to indicate portions of your code that need to be completed, while the latter is used to mark portions of code that need re-writing or fixing.

Consider using these tags to organize your code better. When you are in a hurry and need to find that particular bit of code in a long source file, they come in handy. I certainly use them all the time in my own code.

Testing the table view data source

Exercise: Modify the app so that it shows five rows.

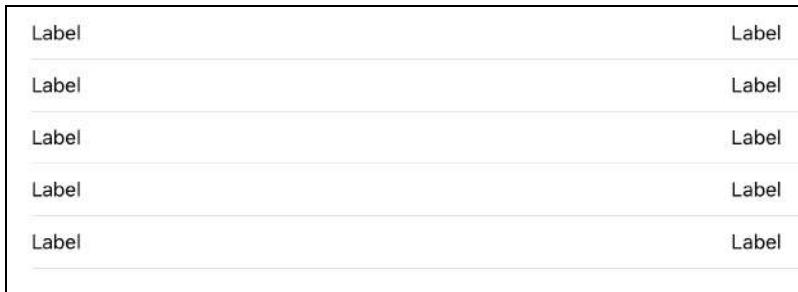
That shouldn't have been too hard:

```
override func tableView(_ tableView: UITableView,  
    numberOfRowsInSection section: Int) -> Int {  
    return 5  
}
```

If you were tempted to go into the storyboard and duplicate the prototype cell five times, then you were confusing cells with rows.

When you make `tableView(_:numberOfRowsInSection:)` return the number 5, you tell the table view that there will be five rows.

The table view then sends the `cellForRowAt` message five times, once for each row. Because `tableView(_:cellForRowAt:)` currently just returns a copy of the prototype cell, your table view will show five identical rows:



The table now has five identical rows

There are several ways to create cells in `tableView(_:cellForRowAt:)`, but by far the easiest approach is what you've done here:

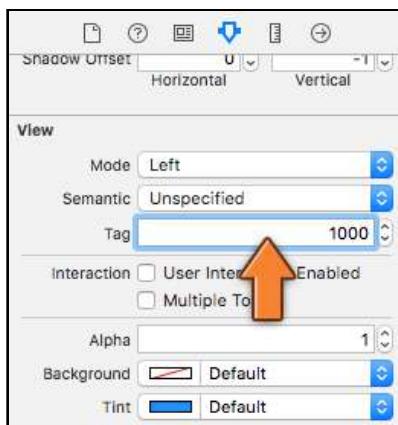
1. Add a prototype cell to the table view in the storyboard.
2. Set a reuse identifier on the prototype cell.
3. Call `tableView.dequeueReusableCell(withIdentifier:for:)`. This makes a new copy of the prototype cell if necessary, or, recycles an existing cell that is no longer in use.

Once you have a cell, you should set it up with the data from the corresponding row and give it back to the table view. That's what you'll do in the next section.

Putting row data into the cells

Currently, the rows (or rather the cells) all contain the placeholder text "Label." Let's add some unique text for each row.

- Open the storyboard and select the left **Label** inside the table view cell. Go to the **Attributes inspector** and set the **Tag** field to 1000.



Set the label's tag to 1000

A *tag* is a numeric identifier that you can give to a user interface control in order to uniquely identify it later. Why the number 1000? No particular reason. It should be something other than 0, as that is the default value for all tags. 1000 is as good a number as any.

- Do the same for the right label, but use 2000 as the tag instead.

Double-check to make sure you set the tag on the *Labels*, not on the Table View Cell or its Content View. It's a common mistake to set the tag on the wrong view and then the results won't be what you expected!

- In **HighScoresViewController.swift**, change `tableView(_:cellForRowAt:)` to the following:

```
override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
       (withIdentifier: "HighScoreItem",
```

```
        for: indexPath)

// Add the following code
let nameLabel = cell.viewWithTag(1000) as! UILabel
let scoreLabel = cell.viewWithTag(2000) as! UILabel

if indexPath.row == 0 {
    nameLabel.text = "The reader of this book"
    scoreLabel.text = "50000"
} else if indexPath.row == 1 {
    nameLabel.text = "Manda"
    scoreLabel.text = "10000"
} else if indexPath.row == 2 {
    nameLabel.text = "Joey"
    scoreLabel.text = "5000"
} else if indexPath.row == 3 {
    nameLabel.text = "Adam"
    scoreLabel.text = "1000"
} else if indexPath.row == 4 {
    nameLabel.text = "Eli"
    scoreLabel.text = "500"
}
// End of new code block

return cell
}
```

You've already seen the first line. It gets a copy of the prototype cell — either a new one or a recycled one — and puts it into a local constant named `cell`. (Recall that this is a constant because it's declared with `let`, not `var`. It is local because it's defined inside a method.)

The first new line that you've just added is:

```
let nameLabel = cell.viewWithTag(1000) as! UILabel
```

Here you ask the table view cell for the view with tag 1000. That is the tag you just set on the label in the storyboard. So, this returns a reference to the corresponding `UILabel`. Using tags is a handy trick to get a reference to a UI element without having to make an `@IBOutlet` variable for it.

Exercise: Why can't you simply add an `@IBOutlet` variable to the view controller and connect the cell's label to that outlet in the storyboard? After all, that's how you created references to the labels in *Bullseye*... so why won't that work here?

Answer: There will be more than one cell in the table and each cell will have its own label. If you connected the label from the prototype cell to an outlet on the view controller, that outlet could only refer to the label from *one* of these cells, not all of them. Since the label belongs to the cell and not to the view controller as a whole, you can't make an outlet for it on the view controller. Confused? We'll circle around to this topic soon, so don't worry about it for now. Back to the code. What is this `indexPath` thing?

`IndexPath` is simply an object that points to a specific row in the table. When the table view asks the data source for a cell, you can look at the row number inside the `indexPath.row` property to find out the row for which the cell is intended.

Note: As was mentioned before, it is also possible for tables to group rows into sections. In an address book app you might sort contacts by last name. All contacts whose last name starts with "A" are grouped into their own section, all contacts whose last name starts with "B" are in another section, and so on.

To find out which section a row belongs to, you'd look at the `indexPath.section` property. This app has no need for this kind of grouping, so you'll ignore the `section` property of `IndexPath` for now.

Now that you know about `indexPath`, the following code should make sense to you:

```
if indexPath.row == 0 {  
    nameLabel.text = "The reader of this book"  
    scoreLabel.text = "50000"  
} else if indexPath.row == 1 {  
    nameLabel.text = "Manda"  
    scoreLabel.text = "10000"  
} else if indexPath.row == 2 {  
    nameLabel.text = "Joey"  
    scoreLabel.text = "5000"  
} else if indexPath.row == 3 {  
    nameLabel.text = "Adam"  
    scoreLabel.text = "1000"  
} else if indexPath.row == 4 {  
    nameLabel.text = "Eli"  
    scoreLabel.text = "500"  
}
```

You have seen this `if – else if – else` structure before. It simply looks at the value of `indexPath.row`, which contains the row number, and changes the label's text accordingly. The cell for the first row gets the player name "The reader of this book" and the scores next to it would be "50000". The cell for the second row gets the



player name “Manda” with scores of “10000”, and so on. Look at that, you’re already ranked the highest!

Note: Computers generally start counting at 0 for lists of items. If you have a list of 4 items, they are counted as 0, 1, 2 and 3. It may seem a little silly at first, but that’s just the way programmers do things.

For the first row in the first section, `indexPath.row` is 0. The second row has row number 1, the third row is row 2, and so on.

Counting from 0 may take some getting used to, but after a while it becomes second nature and you’ll start counting at 0 even when you’re out for groceries.

- Run the app — it now has five rows, each with its own high score:

The reader of this book	50000
Manda	10000
Joey	5000
Adam	1000
Eli	500

The rows in the table now have their own text

That is how you write the `tableView(_:cellForRowAt:)` method to provide data to the table. You first get a `UITableViewCell` object and then change the contents of that cell based on the row number of the `indexPath`.

Tapping on the rows

When you tap on a row, the cell color changes to indicate it is selected. The cell remains selected till you tap another row. You are going to change this behavior so that when you lift your finger the row is deselected.



The reader of this book	50000
Manda	10000
Joey	5000
Adam	1000
Eli	500

A tapped row stays gray

Taps on rows are handled by the table view's **delegate**. Remember you read before that in iOS you often find objects doing something on behalf of other objects? The data source is one example of this, but the table view also depends on another little helper, the table view delegate.

The concept of delegation is very common in iOS. An object will often rely on another object to help it out with certain tasks. This *separation of concerns* keeps the system simple, as each object does only what it is good at and lets other objects take care of the rest. The table view offers a great example of this.

Because every app has its own requirements for what its data looks like, the table view must be able to deal with lots of different types of data. Instead of making the table view very complex, or requiring that you modify it to suit your own apps, the UIKit designers have chosen to delegate the duty of providing the cells to display to another object, the data source.

The table view doesn't really care who its data source is or what kind of data your app deals with, just that it can send the `cellForRowAt` message and receive a cell in return. This keeps the table view component simple and moves the responsibility for handling the data to where it belongs: your code.

Likewise, the table view knows how to recognize when the user taps a row, but what it should do in response depends on the app. In this app, you'll transition to a different view controller; another app will likely do something totally different.

Using the delegation system, the table view can simply send a message that a tap occurred and let the delegate sort it out.

Usually, components will have just one delegate. But the table view splits up its delegate duties into two separate helpers: the `UITableViewDataSource` for putting rows into the table, and the `UITableViewDelegate` for handling taps on the rows and several other tasks.

- To see this, open the storyboard and **Control-click** on the table view to bring up its connections.



The table's data source and delegate are hooked up to the view controller

You can see that the table view's data source and delegate are both connected to the view controller. That is standard practice for a `UITableViewController`. (You can also use table views in a basic `UIViewController` but then you'll have to connect the data source and delegate manually.)

- Add the following method to `HighScoresViewController.swift`:

```
// MARK:- Table View Delegate
override func tableView(_ tableView: UITableView,
                      didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
}
```

The `tableView(_:didSelectRowAt:)` method is one of the table view delegate methods and gets called whenever the user taps on a cell. Run the app and tap a row – the cell briefly turns gray and then becomes de-selected again.

Currently the high scores are hard-coded and never update. You need some way to keep track of new high scores. That means it's time to expand the data source and make it use a proper *data model*, which is the topic of the next section.

Methods with multiple parameters

Most of the methods you used in the *Bullseye* app took only one parameter or did not have any parameters at all, but these new table view data source and delegate methods take two:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    . . .  
}  
  
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    . . .  
}  
  
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {  
    . . .  
}
```

The first parameter is the `UITableView` object on whose behalf these methods are invoked. This is done for convenience, so you won't have to make an `@IBOutlet` in order to send messages back to the table view.

For `numberOfRowsInSection` the second parameter is the section number. For `cellForRowAt` and `didSelectRowAt` it is the index-path.

Methods are not limited to just one or two parameters, they can have many. But for practical reasons two or three is usually more than enough, and you won't see many methods with more than five parameters.

In other programming languages a method typically looks like this:

```
Int numberOfRowsInSection(UITableView tableView, Int section) {  
    . . .  
}
```

In Swift we do things a little differently, mostly to be compatible with the iOS frameworks, which are all written in the Objective-C programming language. Let's take a look again at `numberOfRowsInSection`:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    . . .  
}
```

The method signature for the above method, as discussed before, is `tableView(_:numberOfRowsInSection:)`. If you say that out loud (without the underscores and colons, of course), it actually makes sense. It asks for the number of rows in a particular section of a particular table view.



The first parameter looks like this:

```
_ tableView: UITableView
```

The name of this parameter is `tableView`. The name is followed by a colon and the parameter's type, `UITableView`.

The second parameter looks like this:

```
numberOfRowsInSection section: Int
```

This one has two names, `numberOfRowsInSection` and `section`.

You use the first name, `numberOfRowsInSection`, when calling the method. This is the *external* parameter name. Inside the method itself you use the second name, `section`, known as the *local* parameter name. The data type of this parameter is `Int`.

The `_` underscore is used when you don't want a parameter to have an external name. You'll often see the `_` on the first parameter of methods that come from Objective-C frameworks. With such methods the first parameter only has one name but the other parameters have two. Strange? Yes.

It makes sense if you've ever programmed in Objective-C, but no doubt it looks weird if you're coming from another language. Once you get used to it, you'll find that this notation is actually quite readable.

Sometimes people with experience in other languages get confused because they think that `HighScoresViewController.swift` contains three functions that are all named `tableView()`. But that's not how it works in Swift: the names of the parameters are part of the full method name. That's why these three methods are actually named:

```
tableView(_ :numberOfRowsInSection :)  
tableView(_ :cellForRowAt :)  
tableView(_ :didSelectRowAt :)
```

By the way, the return type of the method is at the end, after the `->` arrow. If there is no arrow, as in `tableView(_ :didSelectRowAt :)`, then the method is not supposed to return a value.

Phew! That was a lot of new stuff to take in, so I hope you're still with me. If not, then take a break and start at the beginning again. You're being introduced to a whole bunch of new concepts all at once and that can be overwhelming.



But don't worry, it's OK if everything doesn't make perfect sense yet. As long as you get the gist of what's going on, you're good to go.

If you want to check your work up to this point, you can find the project files for the app under **20-Table Views** in the Source Code folder.



Chapter 21: The Data Model

By Eli Ganim

In the previous chapter, you created a table view for the high scores and got it to display rows of items. However, this was all done using hard-coded, fake data. This would not do for a real high score screen since your users want to see their own real high scores up there.

To manage and display this information efficiently, you need a data model that allows you to store (and access) the high scores easily. That's what you're going to do in this chapter.

This chapter covers the following:

- **Model-View-Controller:** A quick explanation of the MVC fundamentals that are central to iOS programming.
- **The data model:** Creating a data model to hold the high scores data.



Model-View-Controller

First, a tiny detour into programming-concept-land so that you understand some of the principles behind using a data model. No book on programming for iOS can escape an explanation of **Model-View-Controller**, or MVC for short.

MVC is one of the three fundamental design patterns of iOS. You've already seen the other two: *Delegation*, making one object do something on behalf of another, and *target-action*, connecting events such as button taps to action methods.

The Model-View-Controller pattern states that the objects in your app can be split into three groups:

- **Model objects:** These objects contain your data and any operations on the data. For example, if you were writing a cookbook app, the model would consist of the recipes. In a game, it would be the design of the levels, the player score and the positions of the monsters.

The operations that the data model objects perform are sometimes called the *business rules* or the *domain logic*. For the high score screen, the high scores themselves form the data model.

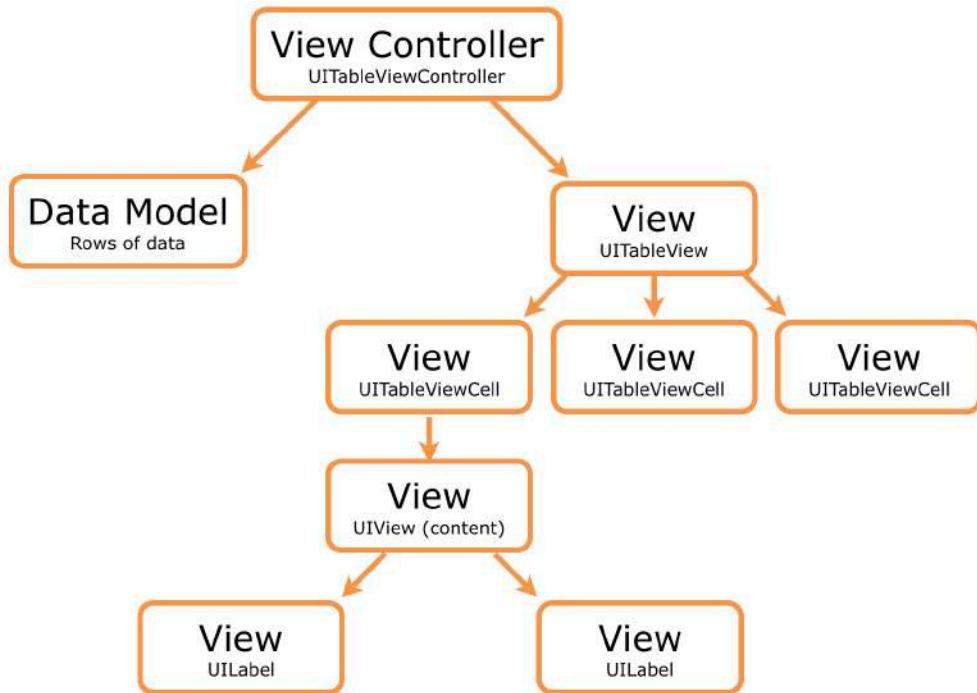
- **View objects:** These make up the visual part of the app: Images, buttons, labels, text fields, table view cells and so on. In a game, the views form the visual representation of the game world, such as the monster animations and a frag counter.

A view can draw itself and responds to user input, but it typically does not handle any application logic. Many views, such as `UITableView`, can be re-used in many different apps because they are not tied to a specific data model.

- **Controller objects:** The controller is the object that connects your data model objects to the views. It listens to taps on the views, makes the data model objects do some calculations in response and updates the views to reflect the new state of your model. The controller is in charge. On iOS, the controller is called the “view controller.”



Conceptually, this is how these three building blocks fit together:



How Model-View-Controller works

The view controller has one main view, accessible through its `view` property, that contains a bunch of subviews. It is not uncommon for a screen to have dozens of views all at once. The top-level view usually fills the whole screen. You design the layout of the view controller's screen in the storyboard.

In the high score screen, the main view is the `UITableView` and its subviews are the table view cells. Each cell also has several subviews of its own, namely the text labels.

Generally, a view controller handles one screen of the app. If your app has more than one screen, each of these is handled by its own view controller and has its own views. Your app flows from one view controller to another.

You will often need to create your own view controllers. However, iOS also comes with ready-to-use view controllers, such as the image picker controller for photos, the mail compose controller that lets you write an email and, of course, the table view controller for displaying lists of items.

Views vs. view controllers

Remember that a view and a view controller are two different things.

A view is an object that draws something on the screen, such as a button or a label. The view is what you see. The view controller is what does the work behind the scenes. It is the bridge that sits between your data model and the views.

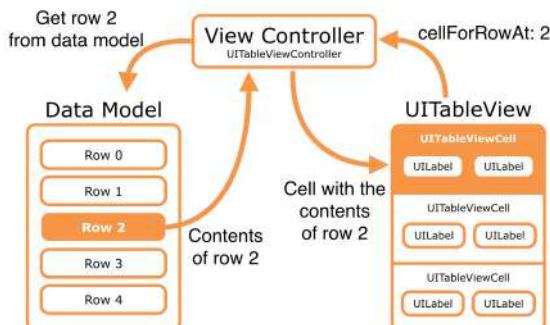
A lot of beginners give their view controllers names such as `FirstView` or `MainView`. That is very confusing! If something is a view controller, its name should end with “ViewController,” not “View.” I sometimes wish Apple had left the word “view” out of “view controller” and just called it “controller” as that is a lot less misleading.

The data model

So far, you’ve put a bunch of fake data into the table view. The data consists of a text string and a number. As you saw in the previous chapter, you cannot use the cells to remember the data as cells get re-used all the time and their old contents get overwritten.

Table view cells are part of the view. Their purpose is to display the app’s data, but that data actually comes from somewhere else: The data model. Remember this well: The rows are the data, the cells are the views.

The table view controller is the thing that ties them together through the act of implementing the table view’s data source and delegate methods.



The table view controller (data source) gets the data from the model and puts it into the cells

The data model for this app will be a list of high score items. Each of these items will get its own row in the table.

For each high score, you need to store two pieces of information: The name of the high scorer (Like “Manda”, “Adam”, etc) and the score.

That is two pieces of information per row, so you need two variables for each row.

The first iteration

First, you'll see the cumbersome way to program this. It will work, but it isn't very smart. Even though this is not the best approach, you should still follow along and copy-paste the code into Xcode and run the app so that you understand how this approach works.

Understanding why this approach is problematic will help you appreciate the proper solution better.

- In **HighScoresViewController.swift**, add the following constants right after the `class HighScoresViewController` line:

```
class HighScoresViewController: UITableViewController {  
    let row0name = "The reader of this book"  
    let row1name = "Manda"  
    let row2name = "Joey"  
    let row3name = "Adam"  
    let row4name = "Eli"  
    let row0score = 50000  
    let row1score = 10000  
    let row2score = 5000  
    let row3score = 1000  
    let row4score = 500  
    . . .
```

These constants are defined outside of any method, they are not “local”, so they can be used by all of the methods in `HighScoresViewController`.

- Change the data source methods to:

```
override func tableView(_ tableView: UITableView,  
    numberOfRowsInSection section: Int) -> Int {  
    return 5  
}  
  
override func tableView(_ tableView: UITableView,  
    cellForRowAt indexPath: IndexPath)  
-> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(  
        ...)
```

```
withIdentifier: "HighScoreItem",
for: indexPath)
let nameLabel = cell.viewWithTag(1000) as! UILabel
let scoreLabel = cell.viewWithTag(2000) as! UILabel

if indexPath.row == 0 {
    nameLabel.text = row0name
    scoreLabel.text = String(row0score)
} else if indexPath.row == 1 {
    nameLabel.text = row1name
    scoreLabel.text = String(row1score)
} else if indexPath.row == 2 {
    nameLabel.text = row2name
    scoreLabel.text = String(row2score)
} else if indexPath.row == 3 {
    nameLabel.text = row3name
    scoreLabel.text = String(row3score)
} else if indexPath.row == 4 {
    nameLabel.text = row4name
    scoreLabel.text = String(row4score)
}
return cell
}
```

► Run the app. It still shows the same five rows as originally.

What have you done here? For every row, you have added 2 constants with the name and score for that row. Together, these constants are your data model. You could have used variables instead of constants, but since the values won't change for this particular example, it's better to use constants.

In `tableView(_:cellForRowAt:)` you look at `indexPath.row` to figure out which row to display and put the text from the corresponding constant into the cell.

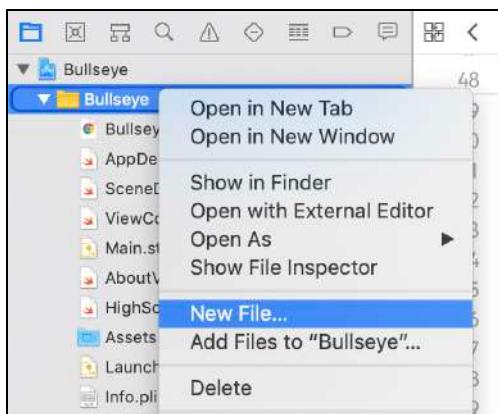
Simplifying the code

Let's combine the name and score into a new object of your own!

The object

► Select the **Bullseye** group in the project navigator and right-click it. Choose **New File...** from the pop-up menu.





Adding a new file to the project

Under the **Source** section, choose **Swift File**.

Click **Next** to continue. Save the new file as **HighScoreItem**. You don't really need to add the **.swift** file extension since it will be automatically added for you.

Click **Create** to add the new file to the project.

- Add the following to the new **HighScoreItem.swift** file, below the **import** line:

```
class HighScoreItem {  
    var name = ""  
    var score = 0  
}
```

The **name** property will store the name of the high scorer, the text that will appear in the table view cell's label, and the **score** property will store the score.

Note: You may be wondering what the difference is between the terms *property* and *instance variable* — we've used both to refer to an object's data items. You'll be glad to hear that these two terms are interchangeable.

In Swift terminology, a **property** is a variable or constant that is used in the context of an object. That's exactly what an instance variable is.

In Objective-C, properties and instance variables are closely related but not quite the same thing. In Swift, they are the same.

That's all for **HighScoreItem.swift** for now. The `HighScoreItem` object currently only serves to combine the `name` and the `score` variables into one object. Later you'll do more with it.

Using the object

Before you try using an array, you'll replace the `name` and `score` instance variables in the view controller with these new `HighScoreItem` objects to see how that approach would work.

► In **HighScoresViewController.swift**, remove the old properties and replace them with `HighScoreItem` objects:

```
class HighScoresViewController: UITableViewController {  
    var row0item = HighScoreItem()  
    var row1item = HighScoreItem()  
    var row2item = HighScoreItem()  
    var row3item = HighScoreItem()  
    var row4item = HighScoreItem()
```

These replace the `row0name`, `row0score`, etc. instance variables.

Wait a minute though... We've had variable declarations with a type, or with explicit values like an empty string or a number, but what are these? These variables are being assigned with what looks like a method!

And you are right about the method. It's a special method that all classes have called an *initializer* method. An initializer method creates a new instance of the given object, in this case `HighScoreItem`. This creates an empty instance of `HighScoreItem` with the default values you defined when you added the class implementation — an empty string ("") for `name` and 0 for `score`.

Instead of the above, you could have used what's known as a *type annotation* to simply indicate the type of `row0Item` like this:

```
var row0item: HighScoreItem
```

If you did that, `row0item` won't have a value yet, it would just be an empty container for a `HighScoreItem` object. And you'd still have to create the `HighScoreItem` instance later in your code. For example, in `viewDidLoad`.

The way you've done the code now, you initialize the variables above immediately with an empty instance of `HighScoreItem` and let Swift's type inference do the work in letting the compiler figure out the type of the variables. Handy, right?



Just to clarify the above a bit more, the data type is like the brand name of a car. Just saying the words “Porsche 911” out loud doesn’t magically get you a new car. You actually have to go to the dealer to buy one.

The parentheses () behind the type name are like going to the object dealership to buy an object of that type. The parentheses tell Swift’s object factory: “Build me an object of the type `HighScoreItem`.”

It is important to remember that just declaring that you have a variable does not automatically make the corresponding object for you. The variable is just the container for the object. You still have to instantiate the object and put it into the container. The variable is the box and the object is the thing inside the box.

Until you order an actual `HighScoreItem` object from the factory and put that into `row0item`, the variable is empty. And empty variables are a big no-no in Swift.

Fixing existing code

Because some methods in the view controller still refer to the old variables, Xcode will throw up multiple errors at this point. Before you can run the app again, you need to fix these errors. So, let’s do that now.

Note: I generally encourage you to type in the code from this book by hand, instead of copy-pasting, because that gives you a better feel for what you’re doing, but in the following instances it’s easier to just copy-paste from the PDF.

Unfortunately, copying from the PDF sometimes adds strange or invisible characters that confuse Xcode. It’s best to first paste the copied text into a plain text editor such as TextMate and then copy-paste from the text editor into Xcode.

Of course, if you’re reading the print edition of this book, copy-pasting from the book isn’t going to work. But you can still use copy-paste to save yourself some effort. Make the changes on one line and then copy that line to create the other lines. Copy-paste is a programmer’s best friend, but don’t forget to update the lines you pasted to use the correct variable names!

- In `tableView(_:cellForRowAt:)`, replace the `if` statements with the following:

```
if indexPath.row == 0 {  
    nameLabel.text = row0item.name
```

```
        scoreLabel.text = String(row0item.score)
    } else if indexPath.row == 1 {
        nameLabel.text = row1item.name
        scoreLabel.text = String(row1item.score)
    } else if indexPath.row == 2 {
        nameLabel.text = row2item.name
        scoreLabel.text = String(row2item.score)
    } else if indexPath.row == 3 {
        nameLabel.text = row3item.name
        scoreLabel.text = String(row3item.score)
    } else if indexPath.row == 4 {
        nameLabel.text = row4item.name
        scoreLabel.text = String(row4item.score)
    }
```

Basically, all of the above changes do one thing. Instead of using the separate `row0name` and `row0score` variables, you now use `row0item.name` and `row0item.score`.

That takes care of all of the errors and you can even build and run the app. But if you do, you'll notice that you get a table with 5 zeros in it.

So what went wrong?

Setting up the objects

Remember how the new `row0item` etc. variables are initialized with empty instances of `HighScoreItem`? That means that the text for each variable is empty. You still need to set up the values for these new variables!

► Modify `viewDidLoad` in `HighScoreViewController.swift` as follows:

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

    // Add the following lines
    row0item.name = "The reader of this book"
    row0item.score = 50000
    row1item.name = "Manda"
    row1item.score = 10000
    row2item.name = "Joey"
    row2item.score = 5000
    row3item.name = "Adam"
    row3item.score = 1000
    row4item.name = "Eli"
    row4item.score = 500
}
```

This code simply sets up each of the new `HighScoreItem` variables that you created.



Essentially, it's doing the same thing as before. Except, this time, the `name` and `score` variables are not separate instance variables of the view controller. Instead, they are properties of a `HighScoreItem` object.

- Run the app just to make sure that everything works now.

Putting the `name` and `score` properties into their own `HighScoreItem` object already improved the code, but it is still a bit unwieldy.

Using arrays

With the current approach, you need to keep around a `HighScoreItem` instance variable for each row. That's not ideal, especially if you want more than just a handful of rows.

Time to bring that array into play!

- In `HighScoresViewController.swift`, remove all of the instance variables and replace them with a single array variable named `items`:

```
class HighScoresViewController: UITableViewController {  
    var items = [HighScoreItem]()
```

Instead of five different instance variables, one for each row, you now have just one variable for the array.

This looks similar to how you declared the previous variables but this time there are square brackets around `HighScoreItem`. Those square brackets indicate that the variable is going to be an array containing `HighScoreItem` objects. And the brackets at the end `()` simply indicate that you are creating an instance of this array. It will create an empty array with no items in the array.

- Modify `viewDidLoad` as follows:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    // Replace previous code with the following  
    let item1 = HighScoreItem()  
    item1.name = "The reader of this book"  
    item1.score = 50000  
    items.append(item1)  
  
    let item2 = HighScoreItem()  
    item2.name = "Manda"  
    item2.score = 10000  
    items.append(item2)
```

```
let item3 = HighScoreItem()
item3.name = "Joey"
item3.score = 5000
items.append(item3)

let item4 = HighScoreItem()
item4.name = "Adam"
item4.score = 1000
items.append(item4)

let item5 = HighScoreItem()
item5.name = "Eli"
item5.score = 500
items.append(item5)

}
```

This is not that different from before, except that you now have to first create — or *instantiate* — each `HighScoreItem` object and add each instance to the array. Once the above code completes, the `items` array contains five `HighScoreItem` objects. This is your new data model.

Simplifying the code – again

Now that you have all your rows in the `items` array, you can simplify the table view data source and delegate methods once again.

- Change this methods:

```
override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
       (withIdentifier: "HighScoreItem",
         for: indexPath)

    let item = items[indexPath.row]           // Add this

    let nameLabel = cell.viewWithTag(1000) as! UILabel
    let scoreLabel = cell.viewWithTag(2000) as! UILabel

    // Replace everything after the above line with the following
    nameLabel.text = item.name
    scoreLabel.text = String(item.score)
    return cell
}
```

That's a lot simpler than what you had before! This method is now only a handful of lines long.



The most important part is the line:

```
let item = items[indexPath.row]
```

This asks the array for the `HighScoreItem` object at the index that corresponds to the row number. Once you have that object, you can simply look at its `name` and `score` properties and do whatever you need to do.

If the user were to add 100 high score items to this list, none of this code would need to change. It works equally well with five items as with a hundred (or a thousand).

Speaking of the number of items, you can now change `numberOfRowsInSection` to return the actual number of items in the array, instead of a hard-coded number.

► Change the `tableView(_:numberOfRowsInSection:)` method to:

```
override func tableView(_ tableView: UITableView,  
                      numberOfRowsInSection section: Int) -> Int {  
    return items.count  
}
```

Not only is the code a lot shorter and easier to read, it can now also handle an arbitrary number of rows. That is the power of arrays!

► Run the app and see for yourself. It should still work exactly the same as before, but the internal structure of the code is way better.

Exercise: Add a few more rows to the table. You should only have to change `viewDidLoad` for this to work.

If you want to check your work, you can find the project files for the current version of the app in the folder **21-The Data Model** in the Source Code folder.



Chapter 22: Navigation Controllers

Eli Ganim

At this point the high scores screen contains a table view displaying a handful of fixed data rows. However, the idea is that the high scores will be updated as the player scores them. Therefore, you need to implement the ability to add items.

In this chapter you'll expand the app to have a **navigation bar** at the top. Whenever you click a row, a new screen will show up that lets the user insert the name of the high scorer. When you tap Done, the new item will be added to the list.

This chapter covers the following:

- **Navigation controller:** Add a navigation controller to the app to allow navigation between screens.
- **Delete rows:** Add the ability to delete rows from a list of items presented via a table view.
- **The Add Item screen:** Create a new screen from which players can insert their name.

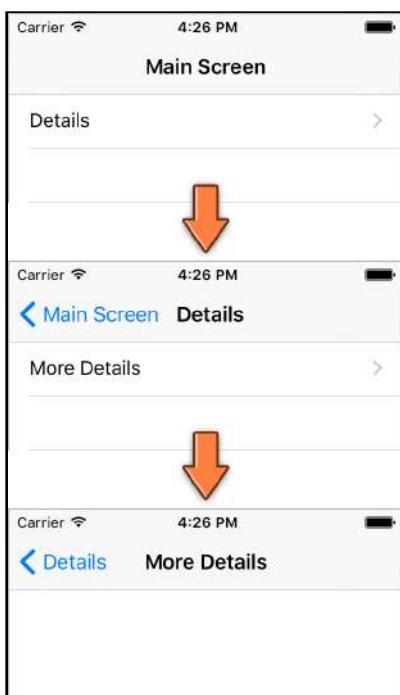


Navigation controller

First, let's add the navigation bar. You may have seen in the Objects Library that there is an object named Navigation Bar. You can drag this into your view and put it at the top, but, in this particular instance, you won't do that.

Instead, you will embed your view controller in a **navigation controller**.

Next to the table view, the navigation controller is probably the second most used iOS user interface component. It is the thing that lets you go from one screen to another:



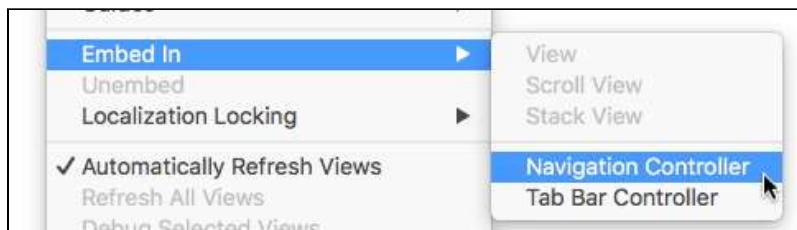
A navigation controller in action

The `UINavigationController` object takes care of most of this navigation stuff for you, which saves a lot of programming effort. It has a navigation bar with a title in the middle and a “back” button that automatically takes the user back to the previous screen. You can put a button (or several buttons) of your own on the right.

Adding a navigation controller

Adding a navigation controller is really easy.

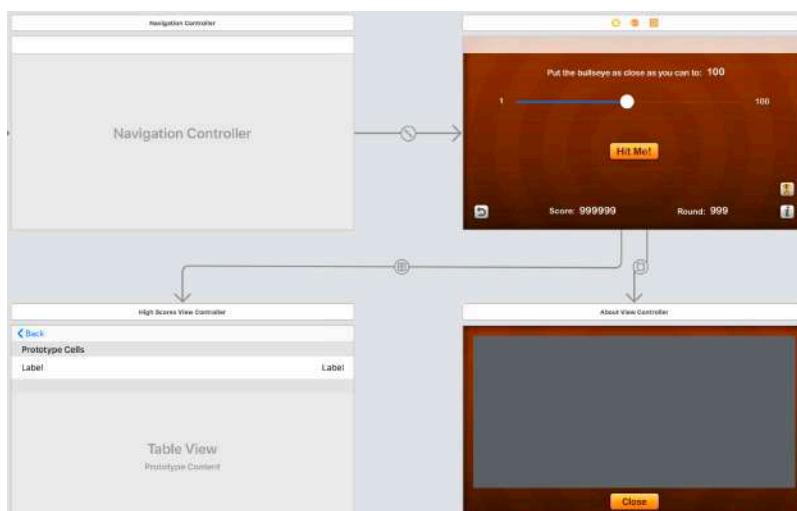
- Open **Main.storyboard** and select the **View Controller Scene**.
- From the menu bar at the top of the screen, choose **Editor** ▶ **Embed In** ▶ **Navigation Controller**.



Putting the view controller inside a navigation controller

Several things have happened in Interface Builder now:

1. Interface Builder has added a new **Navigation Controller Scene** and made a relationship between it and the main view controller.
2. There's a navigation bar at the top of the main screen (just like in the SwiftUI version of *Bullseye*)
3. The **High Scores View Controller** also has a navigation bar with a Back button.



The navigation controller is now linked with your view controller

Why was a navigation bar added to the **High Scores View Controller Scene**, but not to **About View Controller Scene**? When you connected the main view controller to the *About screen* a few chapters ago, you chose the *Present Modally* segue. When you connected the *High Scores Screen*, you chose *Show*.

Segue types

What are the possible Segues and what do they mean? Here is a brief explanation of each type of segue:

- **Show:** Pushes the new view controller onto the navigation stack so that the new view controller is at the top of the navigation stack. It also provides a back button to return to the previous view controller. If the view controllers are not embedded in a navigation controller, then the new view controller will be presented modally (see Present Modally in the list below as to what this means).

Example: Navigating folders in the *Mail* app

- **Show Detail:** For use in a split view controller (you'll learn more about those when developing the last app in this book). The new view controller replaces the detail view controller of the split view when in an expanded two-column interface. Otherwise, if in single-column mode, it will push in a navigation controller.

Example: In *Messages*, tapping a conversation will show the conversation details — replacing the view controller on the right when in a two-column layout, or push the conversation when in a single column layout

- **Present Modally:** Presents the new view controller to cover the previous view controller — most commonly used to present a view controller that covers the entire screen on iPhone, or on iPad it's common to present it as a centered box that darkens the presenting view controller. Usually, if you had a navigation bar at the top or a tab bar at the bottom, those are covered by the modal view controller too.

Example: Selecting Touch ID & Passcode in *Settings*

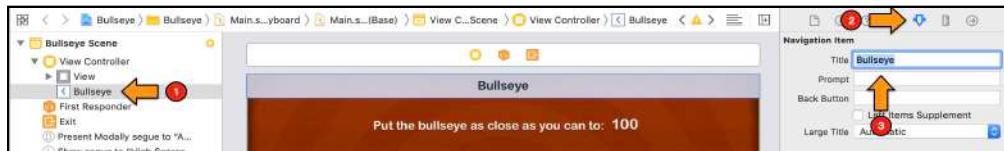
- **Present as Popover:** When run on an iPad, the new view controller appears in a popover, and tapping anywhere outside of this popover will dismiss it. On an iPhone, will present the new view controller modally over the full screen.

Example: Tapping the + button in *Calendar*

- **Custom:** Allows you to implement your own custom segue and have control over its behavior. (You will learn more about this in a later chapter.)
- Run the app and try it out. Navigate to the About screen and then to the High Score screen and witness the difference between the two segue types.

Setting the navigation bar title

- Go back to the storyboard, select **Navigation Item** under **View Controller Scene** in the Document Outline, switch to the Attributes Inspector on the right-hand pane, and set the value of **Title** to **Bullseye**.



Changing the title in the navigation bar

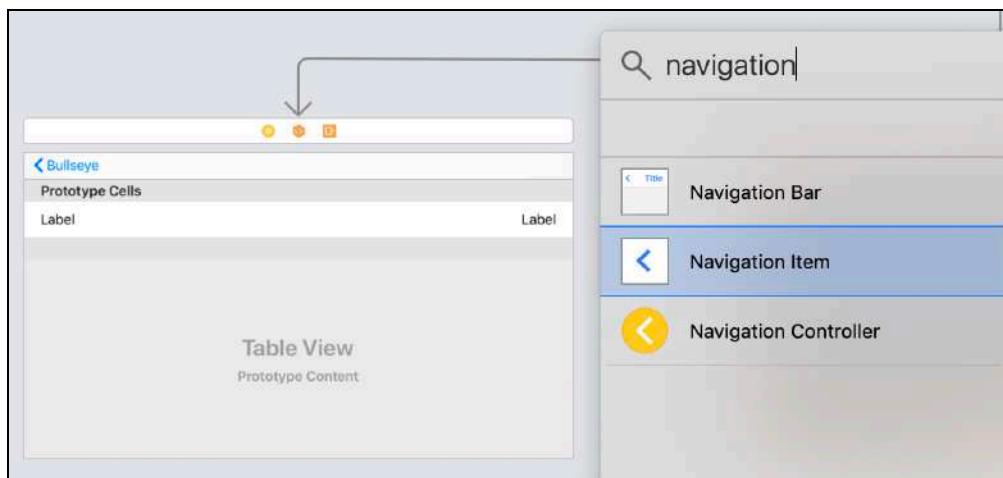
What you're doing here is changing a **Navigation Item** object that was automatically added to the view controller when you chose the **Embed In** command.

The Navigation Item object contains the title and buttons that appear in the navigation bar when this view controller becomes active. Each embedded view controller has its own Navigation Item that it uses to configure what shows up in the navigation bar.

If you run the app now, you'll see that the title in the navigation controller of the main screen is now *Bullseye*. However, if you open the high scores screen you'll see it has no title.

When the navigation controller slides a new view controller in, it replaces the contents of the navigation bar with the new view controller's Navigation Item. You'll add a **Navigation Item** to the high scores view controller and set its title.

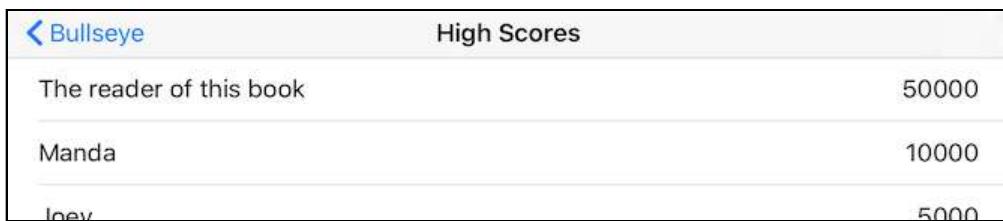
- Go to the storyboard and select the High Scores scene
- Drag a **Navigation Item** from the object library into the scene



Add a navigation item to the view controller

- Change the Navigation Item's title to "High Scores".

Run the app, open the high scores screen and verify the title was indeed updated:



Navigation bar with title

Deleting rows

Imagine you let a friend enjoy the amazing Bullseye game on your iPhone and he reaches a high score you can't beat. That would be really annoying!

For that purpose you need a way to delete high scores from the list. A common way to do this in iOS apps is “swipe-to-delete.” You swipe your finger over a row and a Delete button slides into view. A tap on the Delete button confirms the removal, tapping anywhere else will cancel.

High Scores		
The reader of this book	50000	
	10000	Delete
Joey	5000	
Adam	1000	

Swipe-to-delete in action

Swipe-to-delete

Swipe-to-delete is very easy to implement.

- Add the following method to **HighScoresViewController.swift**. You should put this with the other table view delegate methods, to keep things organized.

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
    // 1
    items.remove(at: indexPath.row)

    // 2
    let indexPaths = [indexPath]
    tableView.deleteRows(at: indexPaths, with: .automatic)
}
```

When the `commitEditingStyle` method is present in your view controller (it is a method defined by the table view data source protocol), the table view will automatically enable swipe-to-delete. All you have to do is:

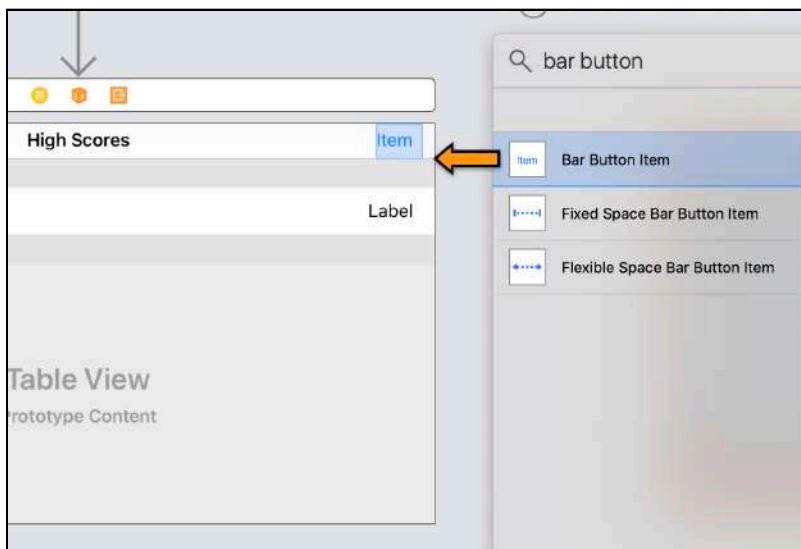
1. Remove the item from the data model.
 2. Delete the corresponding row from the table view.
- Run the app to try it out!

Adding a navigation button

Now that you can remove items from the list, it would be useful to also have a way to reset the high scores list to its initial state. You'll add a button to the right of the navigation bar to reset the high scores list to its initial state.



- Open the storyboard.
- Go to the Objects Library and look for **Bar Button Item**. Drag it into the right-side slot of the navigation bar. (Be sure to use the navigation bar on the High Scores View Controller, not the one from the navigation controller!)



Dragging a Bar Button Item into the navigation bar

By default, this new button is named "Item". Let's rename it to "Reset".

- In the **Attributes inspector** for the bar button item, update the title to Reset.

OK, that gives us a button. If you open the high scores screen, the navigation bar should look like this:



The app with the reset button

Making the navigation button do something

If you tap on your new reset button, it doesn't actually do anything. That's because you haven't hooked it up to an action. You got plenty of exercise with this for *Bullseye*, so it should be child's play for you by now.

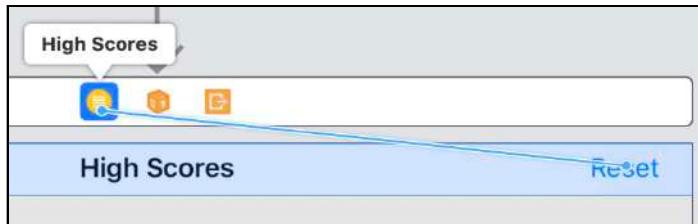
- Add a new action method to **HighScoresViewController.swift**:

```
// MARK:- Actions
```

```
@IBAction func resetHighScores() {  
}
```

You're leaving the method empty for the moment, but it needs to be there so you have something to connect the button to.

- Open the storyboard and connect the Reset button to this action. To do this, **Control-drag** from the reset button to the yellow circle in the bar above the view (this circle represents the High Scores View Controller):



Control-drag from Reset button to High Scores View Controller

Actually, you can Control-drag from the Add button to almost anywhere in the same scene to make the connection.

- After dragging, pick **resetHighScores** from the pop-up (under **Sent Actions**):
- Let's give **resetHighScores()** something to do. Back in **HighScoresViewController.swift**, move all the **HighScoreItem** initialization code from **viewDidLoad()** to **resetHighScores()** and call it from **viewDidLoad()**. The final code should look like this (some items were removed for brevity):

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    resetHighScores()  
}  
  
// MARK:- Actions  
@IBAction func resetHighScores() {  
    items = [HighScoreItem]()  
    let item1 = HighScoreItem()  
    item1.name = "The reader of this book"  
    item1.score = 50000  
    items.append(item1)  
  
    . . .  
  
    let item5 = HighScoreItem()  
    item5.name = "Eli"  
    item5.score = 500  
    items.append(item5)
```

```
    tableView.reloadData()  
}
```

Note that at the beginning you clear the `items` array by assinging it an empty array. You then add the default 5 items and eventually call `reloadData` on the table view, so that it will be refreshed.

Saving and loading high scores

You probably noticed that the high scores data resets every time you restart the app. That's because you're not saving or loading the data.

First, you need to make **HighScoreItem** conform to `Codable` so that you can write it to a file.

► Open **HighScoreItem.swift** and update the class definition to this:

```
class HighScoreItem : Codable
```

Next, you'll create a helper class to save and load the data and use it to fetch items when the high scores screen loads.

► Create a new Swift file and name it **PersistencyHelper.swift**. Put this content in the new file:

```
class PersistencyHelper {  
    static func saveHighScores(_ items: [HighScoreItem]) {  
        let encoder = PropertyListEncoder()  
        do {  
            let data = try encoder.encode(items)  
            try data.write(to: dataFilePath(), options:  
DataWritingOptions.atomic)  
        } catch {  
            print("Error encoding item array: \  
(\(error.localizedDescription))")  
        }  
    }  
  
    static func loadHighScores() -> [HighScoreItem] {  
        var items = [HighScoreItem]()  
        let path = dataFilePath()  
        if let data = try? Data(contentsOf: path) {  
            let decoder = PropertyListDecoder()  
            do {  
                items = try decoder.decode([HighScoreItem].self, from:  
data)  
            }  
        }  
        return items  
    }  
}
```



```
        } catch {
            print("Error decoding item array: \(error.localizedDescription)")
        }
    }
    return items
}

static func dataFilePath() -> URL {
    let paths =
FileManager.default.urls(for: .documentDirectory,
                           in: .userDomainMask)
    return paths[0].appendingPathComponent("HighScores.plist")
}
```

This should all be familiar to you, since you've done exactly the same thing in *Checklists*. You have one method to save the high scores to file and one that loads them from the file. The third method simply creates the path to the plist as a URL.

Now it's time to use these methods. First, you want to load the high scores.

- Open **HighScoresViewController.swift** and add `loadHighScores()` to `viewDidLoad()`. If there's no high scores file (or if loading fails for any reason), you fallback to the default list of high scores:

```
override func viewDidLoad() {
    super.viewDidLoad()
    items = PersistenceHelper.loadHighScores()
    if (items.count == 0) {
        resetHighScores()
    }
}
```

Next, you want to save the high scores whenever an item is deleted or the list is reset.

- Add `PersistenceHelper.saveHighScores(items)` at the end of `resetHighScores()` and `tableView(_:commit:forRowAt:)`.

Adding new high scores

There's one piece missing: How do you add new high scores to the list? Obviously, it needs to happen when a game ends.



In Bullseye everyone's a winner. Even if your score is really low - you still make it to the high scores list (albeit at the bottom of the list).

Exercise: Where's the right place to detect when a game ends, and how would you add the new high score?

The score needs to be added when a game ends, which is right before a new game starts.

- Open **ViewController.swift** and add this at the top of the method `startNewGame()`:

```
@IBAction func startNewGame() {
    addHighScore(score)
    . . .
```

Next, you need to implement the new method.

- Add this code somewhere in **ViewController.swift**:

```
func addHighScore(_ score:Int) {
    // 1
    guard score > 0 else {
        return;
    }

    // 2
    let highscore = HighScoreItem()
    highscore.score = score
    highscore.name = "Unknown"

    // 3
    var highScores = PersistencyHelper.loadHighScores()
    highScores.append(highscore)
    highScores.sort { $0.score > $1.score }
    PersistencyHelper.saveHighScores(highScores)
}
```

Here's what this piece of code is doing:

1. Make sure the score is higher than 0, since you don't want to store games in which the player didn't score any points.
2. Create a new `HighScoreItem` with the score and set the player name to "Unknown".



- Load the high scores from the file, add the new score, sort the list and save it back to the file.

Run the app and give it a try. Play a game, click on the "Start Over" button to end the game and head over to the high scores screen to see your score.

The Edit High Score screen

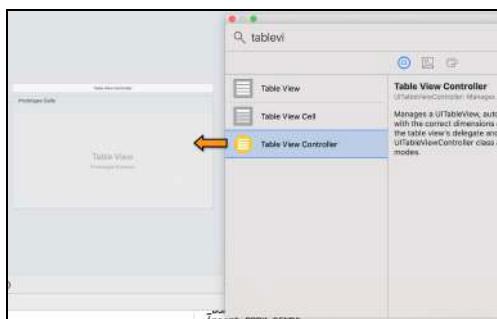
You've learned how to add new high scores, but all of them contain the same player name - "Unknown". You will need to provide a way to change the name. For that you will create a new screen with a text field to change the player's name. It will look like this:



The Edit High score screen

Adding a new view controller to the storyboard

- Go to the Objects Library and drag a new **Table View Controller** (not a regular view controller) on to the storyboard canvas.

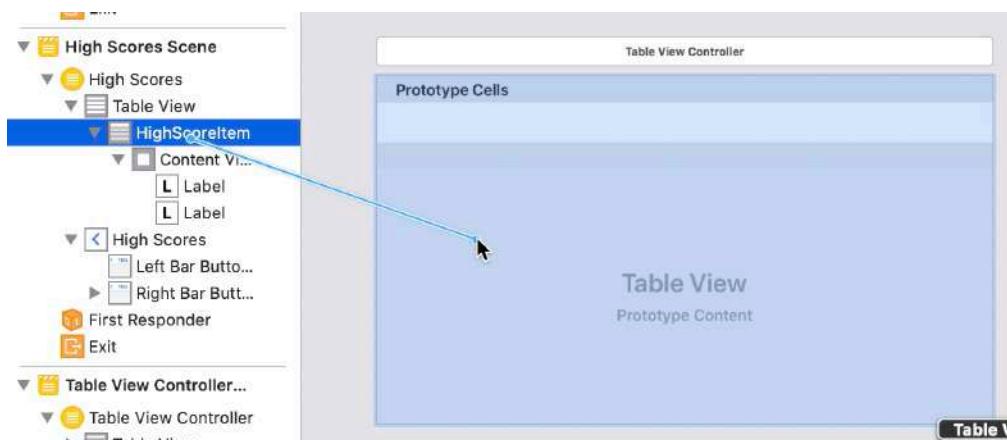


Dragging a new Table View Controller into the canvas

You may need to zoom out to fit everything properly. Right-click on the canvas to get a pop-up with zoom options, or use the **- 100% +** controls at the bottom of the Interface Builder canvas. (You can also double-click on an empty spot in the canvas

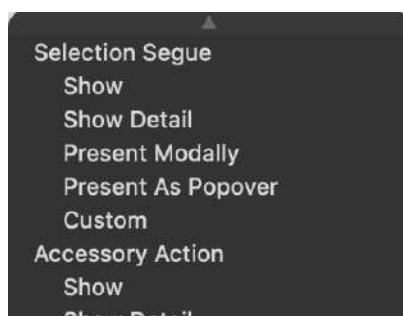
to zoom in or out. Or, if you have a Trackpad, simply pinch with two fingers to zoom in or out.)

- With the new view controller in place, select **Table View** and change its view's background to **Group Table View Background**.
- Select the prototype cell from the High Scores View Controller. **Control-drag** to the new view controller. It might be difficult to capture the correct object here, so instead you can control-drag from **HighScoreItem** in the outline to the left.



Control-drag from the Add button to the new table view controller

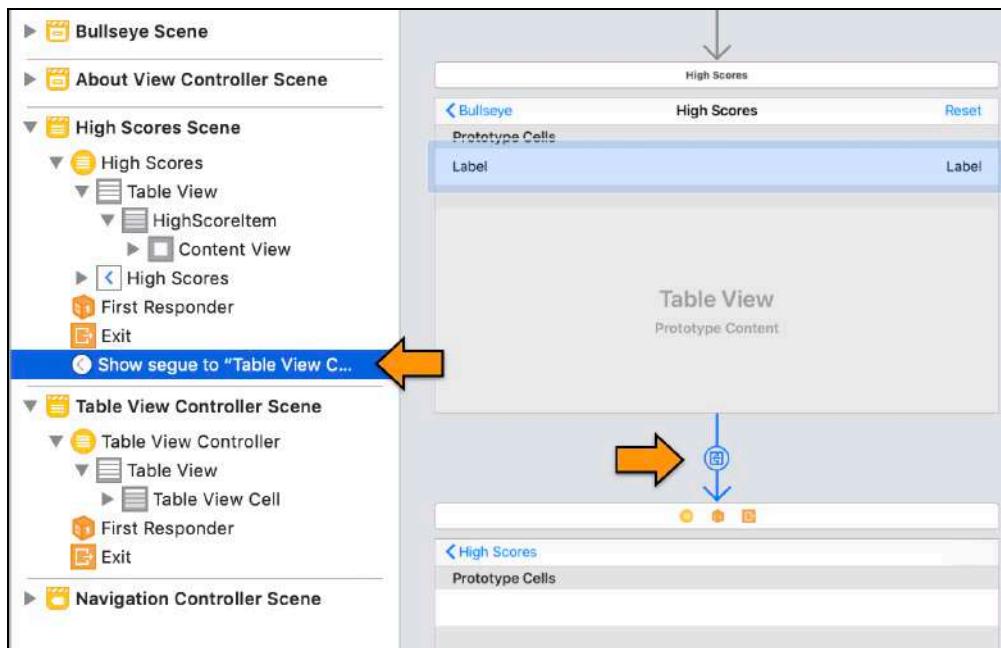
Let go of the mouse and a list of options pops up.



The Action Segue popup

- Choose **Show** from the menu.

The segue is represented by the arrow between the two view controllers:



A new segue is added between the two view controllers

- Run the app to see what it does.

When you press any of the cells, a new empty table view slides in from the right. You can press the back button – the one that says “High Scores” – at the top to go back to the previous screen.

Note: Xcode may be giving you the warning, “Prototype table cells must have reuse identifiers”. You might remember this issue from before – you will fix this issue soon.

Customizing the navigation bar

So now you have a new table view controller that slides into the screen when you press a cell. However, this screen is empty. Data input screens usually have a navigation bar with a Cancel button on the left and a Done button on the right. In some apps the button on the right is called Save or Send. Pressing either of these buttons will close the screen, but only Done will save your changes.

- First, drag a **Navigation Item** from the Objects Library on to the new scene.
- Next, drag two **Bar Button Items** on to the navigation bar, one to the left slot (removing the existing back button) and one to the right slot.



The navigation bar items for the new screen

- In the **Attributes inspector** for the left button choose **System Item: Cancel**.
- For the right button choose **Done** for both **System Item** and **Style** attributes.

Don't type anything into the button's Title field. The Cancel and Done buttons are built-in button types that automatically use the proper text. If your app runs on an iPhone where the language is set to something other than English, these predefined buttons are automatically translated into the device's language.

- Double-click the navigation bar for the new table view controller to edit its title and change it to **Edit High Score**. You can also change this via the Attributes inspector as you did before.
- Run the app, click on the high scores button, tap any cell and you'll see that your new screen has Cancel and Done buttons.



The Cancel and Done buttons in the app

Making your own view controller class

You created a custom view controller for the About screen. Do you remember how to do it on your own? If not, here are the steps:

- Right-click on the Bullseye group (the yellow folder) in the project navigator and choose **New File...** Choose the **Cocoa Touch Class** template.
- In the next dialog, set the Class to **EditHighScoreViewController** and Subclass to **UITableViewController** (when you change the subclass, the class name will

automatically change — so either set the subclass first or change the class name back after the change). Leave the language at **Swift** (or change it back if it is not set to Swift).

- Save the file to your project folder, which should be the default location.
- The file should have a lot of source and commented code — this is known as *boilerplate code*, or code that is generally always needed. In this particular case, you don't need most of it. So remove everything except for `viewDidLoad` (and remove the comments from inside `viewDidLoad` as well) so that your code looks like this:

```
import UIKit

class EditHighScoreViewController: UITableViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

This tells Swift that you have a new object for a table view controller that goes by the name of `EditHighScoreViewController`. You'll add the rest of the code soon. First, you have to let the storyboard know about this new view controller.

- In the storyboard, select the Edit High Score Scene and go to the **Identity inspector**. Under **Custom Class**, type `EditHighScoreViewController`.

This tells the storyboard that the view controller from this scene is actually your new `EditHighScoreViewController` object.

Make sure that it is really the view controller that is selected before you change the fields in the Identity inspector (the scene needs to have a blue border). A common mistake is to select the table view and change that.

Making the navigation buttons work

There's still one issue — the Cancel and Done buttons ought to close the Add Item screen and return the app to the main screen, but tapping them has no effect yet.

Exercise: Do you know why the Cancel and Done buttons do not return you to the main screen?

Answer: Because those buttons have not yet been hooked up to any actions!



You will now implement the necessary action methods in **EditHighScoreViewController.swift**.

- Add these new `cancel()` and `done()` action methods:

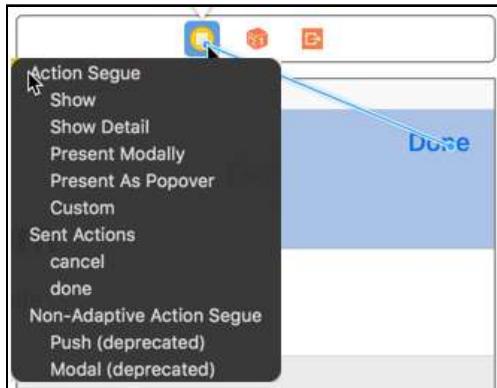
```
// MARK:- Actions
@IBAction func cancel() {
    navigationController?.popViewControllerAnimated(true)
}

@IBAction func done() {
    navigationController?.popViewControllerAnimated(true)
}
```

This tells the navigation controller to close the Add Item screen with an animation and to go back to the previous screen, which in this case is the main screen.

You still need to hook up the Cancel button to the `cancel()` action and the Done button to the `done()` action.

- Open the storyboard and find the Add Item View controller. **Control-drag** from the bar buttons to the yellow circle icon and pick the proper action from the pop-up menu.



Control-dragging from the bar button to the view controller

- Run the app to try it out. The Cancel and Done buttons now return the app to the main screen.

What do you think happens to the `EditHighScoreViewController` object when you dismiss it? After the view controller disappears from the screen, its object is destroyed and the memory it was using is reclaimed by the system.

Every time the user opens the Edit High Score screen, the app makes a new instance of it. This means a view controller object is only alive for the duration that the user is interacting with it; there is no point in keeping it around afterwards.

Container view controllers

You've read that one view controller represents one screen, but here you actually have two view controllers for each screen: a Table View controller that sits inside a navigation controller.

The navigation controller is a special type of view controller that acts as a container for other view controllers. It comes with a navigation bar and has the ability to easily go from one screen to another, by sliding them in and out of sight. The container essentially "wraps around" these screens.

The navigation controller is just the frame that contains the view controllers that do the real work, which are known as the "content" controllers. Here, the `HighScoresViewController` provides the content for the first screen; the content for the second screen comes from the `EditHighScoreViewController`.

Another often-used container is the Tab Bar controller, which you'll see in the next app.

On the iPad, container view controllers are even more commonplace. View controllers on the iPhone are full-screen but on the iPad they often occupy only a portion of the screen, such as the content of a popover or one of the panes in a split-view.

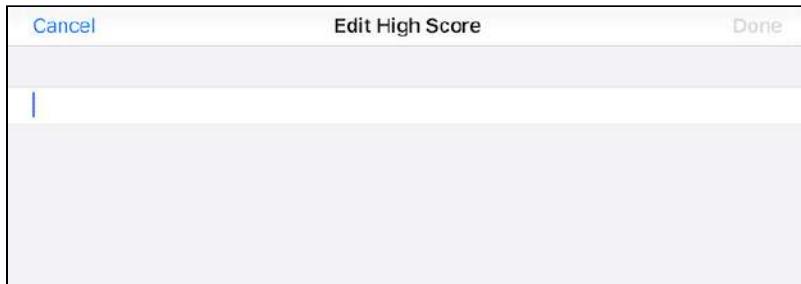
This completes the implementation of the navigation functionality for your app. If at any point you got stuck, you can refer to the project files for the app from the **22-Navigation Controllers** folder in the Source Code folder.

Chapter 23: Edit High Score Screen

Eli Ganim

Now that you have the navigation flow from your main screen to the Edit High Score screen working, it's time to actually implement the edit functionality for this screen!

Let's change the look of the Edit screen. Currently, it is an empty table with a navigation bar on top — but it's going to look like this:



What the Add Item screen will look like when you're done

This chapter covers the following:

- **Static table cells:** Add a static table view cell to the table to display the text field for data entry.
- **Read from the text field:** Access the contents of the text field.
- **Polish it up:** Improve the look and functionality of the Edit High Score screen.

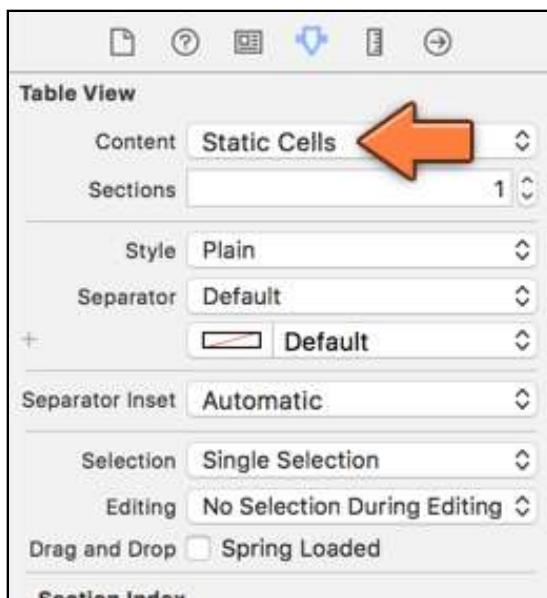


Static table cells

First, you need to add a table view cell to handle the data input for the Edit High Score screen. As is generally the case with UI changes, you start with the storyboard.

Storyboard changes

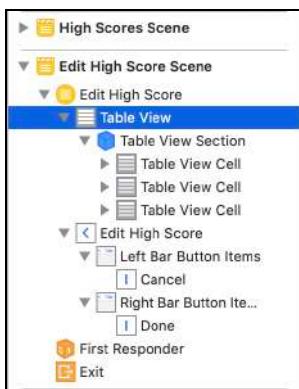
- Open the storyboard and select the **Table View** object inside the Edit High Score scene.
- In the **Attributes inspector**, change the **Content** setting from Dynamic Prototypes to **Static Cells**.



Changing the table view to static cells

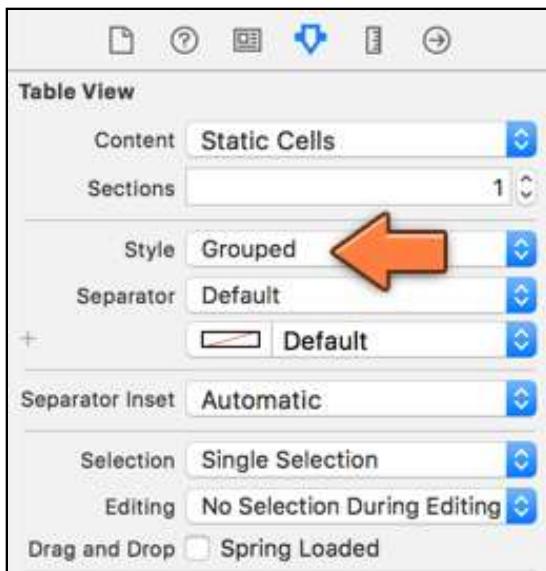
You use static cells when you know beforehand how many sections and rows the table view will have. This is handy for screens that require the user to enter data, such as the one you're building here. With static cells, you can design the rows directly in the storyboard. For a table with static cells, you don't need to provide a data source and you can hook up the labels and other controls from the cells directly to outlets on the view controller.

As you can see in the Document Outline, the table view now has a table view section object under it and three table view Cells in that section. You may need to expand the table view item first by clicking the disclosure triangle.



The table view has a section with three static cells

- Select the bottom two cells and delete them by pressing the **delete** key on your keyboard. You only need one cell for now.
- Select the table view again and in the **Attributes inspector** set its **Style** to **Grouped**.



The table view with grouped style

Next up, you'll add a text field component inside the table view cell that lets the user type text.

- Drag a **text field** object into the cell and size it up nicely. You might want to add left, top, right and bottom Auto Layout constraints to the text field if you don't want

any Xcode warnings. You know how to do that on your own, right? Hint: use the **Add New Constraints** button at the bottom of the Interface Builder screen after you've sized/positioned the field as you want.

- In the **Attributes inspector** for the text field, set the **Border Style** to **no border** by selecting the dotted box:

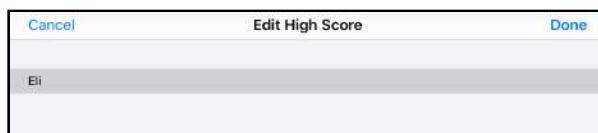


Adding a text field to the table view cell

- Run the app and click on any high score to open the Edit High Score screen. Tap on the cell with the text field and you'll see the keyboard slide in from the bottom of the screen.

Disabling cell selection

Look what happens when you tap just outside of the text field's area but still in the cell. Try tapping in the margins that surround the text field:



Whoops, that looks a little weird

The row turns gray because you selected it. Oops, that's not what you want. You should disable selections for this row. You can do this easily via code by adding the following table view delegate method to **EditHighScoreViewController.swift**:

```
// MARK:- Table View Delegates
override func tableView(_ tableView: UITableView,
    willSelectRowAt indexPath: IndexPath)
    -> IndexPath? {
    return nil
}
```

When the user taps on a cell, the table view sends the delegate a `willSelectRowAt` message that says: “Hi delegate, I am about to select this particular row.”

By returning the special value `nil`, the delegate answers: “Sorry, but you’re not allowed to!”

The `tableView(_:willSelectRowAt:)` method is supposed to return an `IndexPath` object. However, you can also make it return `nil`, indicating no value/object.

That’s what the `?` behind `IndexPath` is for. The question mark tells the Swift compiler that you can also return `nil` from this method. Note that returning `nil` from a method is only allowed if there is a question mark (or exclamation point) behind the return type. A type declaration with a question mark behind it is known as an *optional*. You’ll learn more about optionals in the next chapter.

The special value `nil` represents “no value” but it’s used to mean different things throughout the iOS SDK. Sometimes it means “nothing found” or “don’t do anything.” Here it means that the row should not be selected when the user taps it.

How do you know what `nil` means for a certain method? You can find that in the documentation of the method in question.

In the case of `willSelectRowAt`, the iOS documentation says:

Return Value: An `indexPath` object that confirms or alters the selected row.
Return an `IndexPath` object other than `IndexPath` if you want another cell to be selected. Return `nil` if you don’t want the row selected.

This means you can either:

1. Return the same `IndexPath` you were given. This confirms that this row can be selected.
2. Return another `IndexPath` to select a different row.
3. Return `nil` to prevent the row from being selected, which is what you did.



Working with the text field

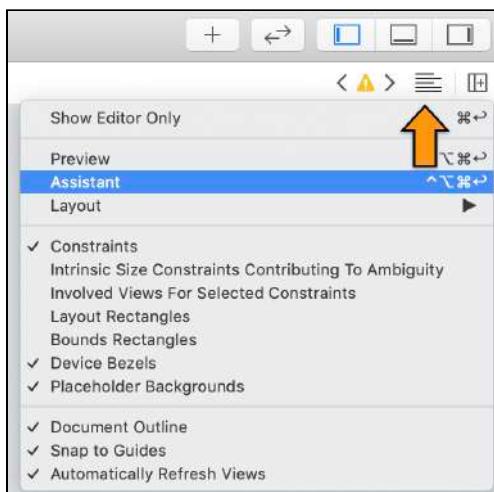
You have a text field in a table view cell that the user can type into. How do you populate it with the current name from the `HighScoreItem`? And how do you read the text that the user has typed?

Adding an outlet for the text field

You already know how to refer to controls from within your view controller: Use an outlet. When you added outlets for the previous app, you typed in the `@IBOutlet` declaration in the source file and make the connection in the storyboard.

You're going to see a trick now that will save you some typing. You can let Interface Builder do all of this automatically by control-dragging from the control in question directly into your source code file!

- First, go to the storyboard and select the **Edit High Score View Controller**. Then, open the **Assistant editor** using the toolbar button on the top right.

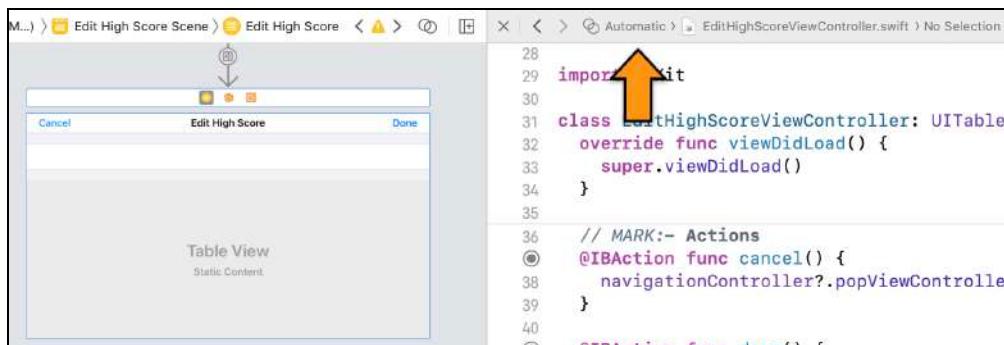


Click the toolbar button to open the Assistant editor

This may make the screen a little crowded. — there might now be up to five horizontal panels open. If you're running out of space, you might want to close the Project navigator, the Utilities pane and/or the Document Outline using the relevant toolbar buttons.

The Assistant editor opens a new pane on the right of the screen by default. It might give you horizontal split views instead if you have changed your default view settings.

In the Jump Bar, below the toolbar, it should say **Automatic** and the Assistant editor should be displaying the **EditHighScoreViewController.swift** file:

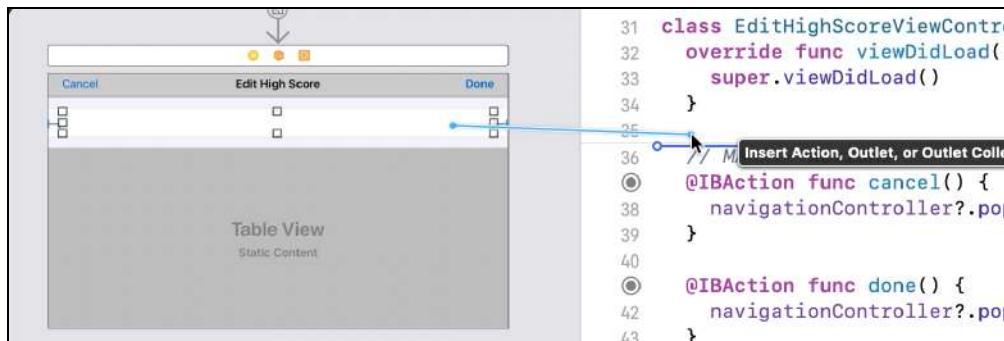


The Assistant editor

“Automatic” means the Assistant editor tries to figure out what other file is related to the one you’re currently editing. When you’re editing a storyboard, the related file is generally the selected view controller’s Swift file.

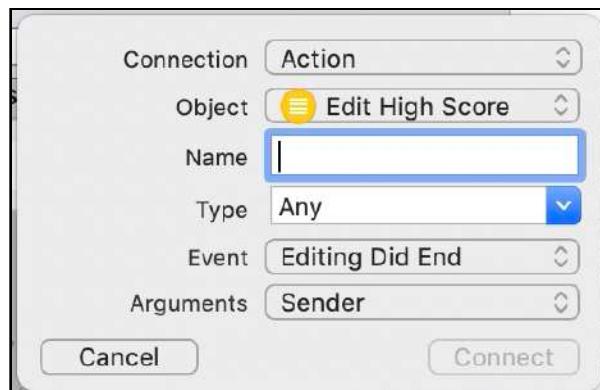
Sometimes Xcode can be a little dodgy here. If it shows you something other than **EditHighScoreViewController.swift**, then click in the Jump Bar and manually select the correct file.

- With the storyboard and the Swift file side-by-side, select the text field. Then, **Control-drag** from the text field into the Swift file.



Control-dragging from the text field into the Swift file

When you let go, a pop-up appears:



The pop-up that lets you add a new outlet

- Choose the following options:
 - Connection: Outlet
 - Name: **textField**
 - Type: UITextField
 - Storage: Weak

Note: If “Type” does not say UITextField, but instead says UITableView or UITableViewCell, then you selected the wrong thing.

Make sure you’re control-dragging from the text field inside the cell, not the cell itself. Granted, it’s kinda hard to see being white on white. If you’re having trouble selecting the text field, click that area several times in succession.

You can also control-drag from “No Border Style Text Field” in the Document Outline.

- Press **Connect** and, voila, Xcode automatically inserts an @IBOutlet for you and connects it to the text field object.

In code it looks like this:

```
@IBOutlet weak var textField: UITextField!
```

Just by dragging, you have successfully hooked up the text field object with a new property named `textField`. How easy was that?

Reading from the text field

Now, you'll modify the `done()` action to write the contents of this text field to the Xcode Console, the pane at the bottom of the screen where `print()` messages show up. This is a quick way to verify that you can actually read what the user typed.

- In `EditHighScoreViewController.swift`, change `done()` to:

```
@IBAction func done() {  
    // Add the following line  
    print("Contents of the text field: \(textField.text!)")  
  
    navigationController?.popViewController(animated: true)  
}
```

You can make these changes directly inside the Assistant editor. It's very handy that you can edit the source code and the storyboard side-by-side.

- Run the app, go to the high scores screen, click on any high score to navigate to the Edit High Score screen and type something in the text field. When you press Done, the Edit High Score screen should close and Xcode should reveal the Debug pane with a message like this:

```
Contents of the text field: Hello, world!
```

Great, so that works! `print()` should be an old friend by now. It's one of the faithful debugging companions.

Note: Because the iOS Simulator already outputs a lot of debug messages of its own, it may be a bit hard to find your `print()` messages in the Console. Luckily, there is a filter box at the bottom that lets you search for your own messages – just type in what you're looking for into the filter box.

Polishing it up

Before you write the code to take the text and update the high score item, let's improve the design and workings of the Edit High Score screen a little.



Giving the text field focus on-screen opening

For instance, it would be nice if you didn't have to tap on the text field to bring up the keyboard. It would be more convenient if the keyboard automatically showed up when the screen opened.

- To accomplish this, add a new method to **EditHighScoreViewController.swift**.

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
    textField.becomeFirstResponder()  
}
```

The view controller receives the `viewDidAppear()` message just before it becomes visible. That is a perfect time to make the text field active. You do this by sending it the `becomeFirstResponder()` message.

If you've done programming on other platforms, this is often called "giving the control focus." In iOS terminology, the control becomes the *first responder*.

- Run the app and go to the Edit High Score screen. You can start typing right away.

Again, note that the keyboard may not appear on the Simulator. Press `⌘+K` to bring it up. The keyboard will always appear when you run the app on an actual device, though.

It's often little features like these that make an app a joy to use. Having to tap on the text field before you can start typing gets old really fast. In this fast-paced age, using their phones on the go, users don't have the patience for that. Such minor annoyances may be reason enough for users to switch to a competitor's app. I always put a lot of effort into making my apps as frictionless as possible.

Styling the text field

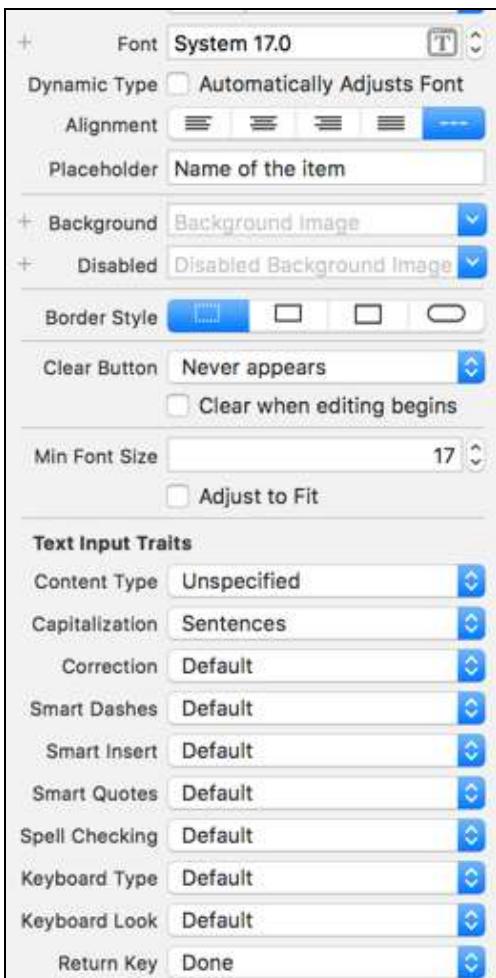
With that in mind, let's style the input field a bit.

- Open the storyboard and select the text field. Go to the **Attributes inspector** and set the following attributes:

- Placeholder: **High scorer name**
- Font: System 17
- Adjust to Fit: Uncheck this
- Capitalization: Sentences



- Return Key: Done



The text field attributes

There are several options here that let you configure the keyboard that appears when the text field becomes active.

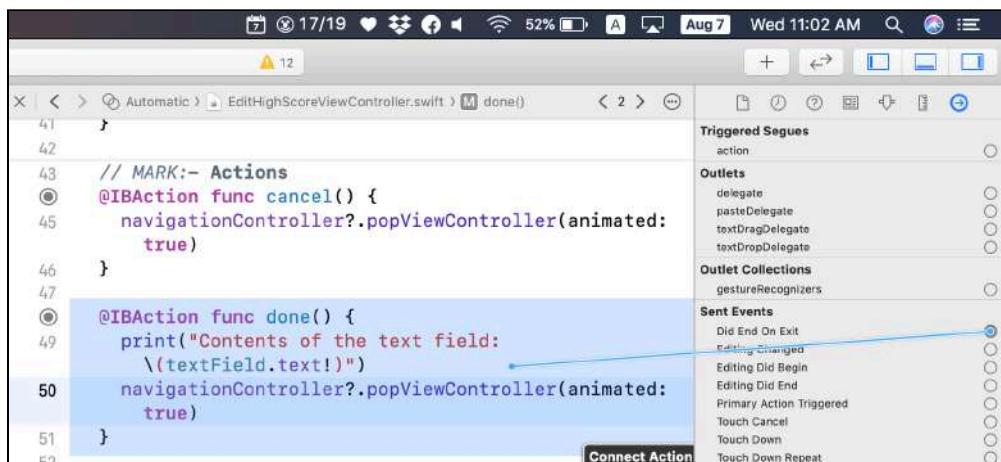
If this were a field that only allowed numbers, for example, you would set the Keyboard Type to Number Pad. If it were an email address field, you'd set it to E-mail Address. For our purposes, the Default keyboard is appropriate.

You can also change the text that is displayed on the keyboard's "Return" key. By default, it says "Return" but you set it to "Done." This is just the text on the button, it doesn't automatically close the screen. You still have to make the keyboard's Done button trigger the same action as the Done button from the navigation bar.

Handling the keyboard Done button

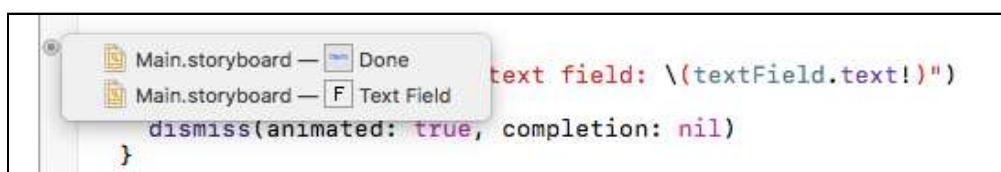
- Make sure the text field is selected and open the **Connections inspector**. Drag from the **Did End on Exit** event to the view controller and pick the **done** action.

If you still have the Assistant editor open, you can also drag directly to the source code for the `done()` method.



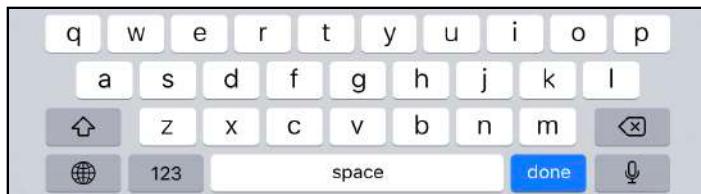
Connecting the text field to the done() action method

To see the connections for the `done` action, click on the circle in the gutter next to the method name. The pop-up shows that `done()` is now connected to both the bar button and the text field:



Viewing the connections for the done() method

- Go to the Edit High Score screen. Pressing Done on the keyboard will now close the screen and print the text to the debug area.



The keyboard now has a big blue Done button

Disallowing empty input

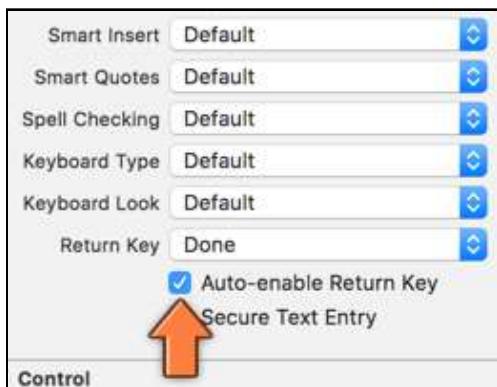
Now that you have user input working, It's always good to validate what the user entered to make sure that the input is acceptable. For instance, what should happen if the user taps the Done button on the Edit High Score screen without entering any text?

Having a high score that has no name is not very useful. So, to prevent this, you should disable the Done button when no text has been typed yet.

Of course, you have two Done buttons to take care of: One on the keyboard and one in the navigation bar. Let's start with the Done button from the keyboard as this is the simplest one to fix.

- On the **Attributes inspector** for the text field, check **Auto-enable Return Key**.

That's it. Now, when you run the app, the Done button on the keyboard is disabled when there is no text in the text field. Try it out!



The Auto-enable Return Key option disables the return key when there is no text

For the Done button in the navigation bar, you have to do a little more work. You have to check the contents of the text field after every keystroke to see if it is now empty or not. If it is, then you disable the button.

The user can always press Cancel, but Done only works when there is text. To listen to changes to the text field — which may come from taps on the keyboard but also from cut/paste — you need to make the view controller a delegate for the text field.

The text field will send events to its delegate to let it know what is going on. The delegate, which will be the `EditHighScoreViewController`, can then respond to these events and take appropriate actions.

A view controller is allowed to be the delegate for more than one object. The `EditHighScoreViewController` is already a delegate, and data source, for the `UITableView` because it is a `UITableViewController`. Now, it will also become the delegate for the text field object: `UITextField`.

These are two different delegates and you make the view controller play both roles. Later on, you'll add even more delegates for this app.

Becoming a delegate

Delegates are used everywhere in the iOS SDK, so it's good to remember that it always takes three steps to become a delegate:

1. You declare yourself capable of being a delegate. To become the delegate for `UITextField` you need to include `UITextFieldDelegate` in the class line for the view controller. This tells the compiler that this particular view controller can actually handle the notification messages that the text field sends to it.
2. You let the object in question, in this case the `UITextField`, know that the view controller wishes to become its delegate. If you forget to tell the text field that it has a delegate, it will never send you any notifications.
3. Implement the delegate methods. It makes no sense to become a delegate if you're not responding to the messages you're being sent!

Often, delegate methods are optional, so you don't need to implement all of them. For example, `UITextFieldDelegate` actually declares seven different methods but you only care about `textField(_:shouldChangeCharactersIn:replacementString:)` for this app.

- In **EditHighScoreViewController.swift**, add `UITextFieldDelegate` to the class declaration:

```
class EditHighScoreViewController: UITableViewController,  
UITextFieldDelegate {
```

The view controller now says: “I can be a delegate for text field objects.”

You also have to let the text field know that you have a delegate for it.

- Go to the storyboard and select the text field.

There are several different ways in which you can hook up the text field’s delegate outlet to the view controller. One way is to go to its **Connections inspector** and drag from **delegate** to the view controller’s little yellow icon:



Drag from the Connections inspector to connect the text field delegate

Configuring the Done button

You also have to add an outlet for the Done bar button item so you can send it messages from within the view controller to enable or disable it.

- Open the **Assistant editor** and make sure **EditHighScoreViewController.swift** is visible in the assistant pane.
- **Control-drag** from the Done bar button into the Swift file and let go. Name the new outlet `doneBarButton`.

This adds the following outlet:

```
@IBOutlet weak var doneBarButton: UIBarButtonItem!
```

- Add the following to **EditHighScoreViewController.swift**, at the bottom and before the final curly brace:

```
// MARK:- Text Field Delegates
```

```
func textField(_ textField: UITextField,  
              shouldChangeCharactersIn range: NSRange,  
              replacementString string: String) -> Bool {  
  
    let oldText = textField.text!  
    let stringRange = Range(range, in: oldText)!  
    let newText = oldText.replacingCharacters(in: stringRange,  
                                              with: string)  
  
    if newText.isEmpty {  
        doneBarButton.isEnabled = false  
    } else {  
        doneBarButton.isEnabled = true  
    }  
    return true  
}
```

This is one of the `UITextField` delegate methods. It is invoked every time the user changes the text, whether by tapping on the keyboard or via cut/paste.

First, you figure out what the new text will be:

```
let oldText = textField.text!  
let stringRange = Range(range, in:oldText)!  
let newText = oldText.replacingCharacters(in: stringRange, with:  
                                         string)
```

The `textField(_:shouldChangeCharactersIn:replacementString:)` delegate method doesn't give you the new text, only which part of the text should be replaced (the range) and the text it should be replaced with (the replacement string). You need to calculate what the new text will be by taking the text field's text and doing the replacement yourself. This gives you a new string object that you store in the `newText` constant.

NSRange vs. Range and NSString vs. String

In the above code, you get a parameter as `NSRange` and you convert it to a `Range` value. If you're wondering what a range is, the clue is in the name. A range object gives you a range of values. Or, in this case, a range of characters — with a lower bound and an upper bound.

So, why did we convert the original `NSRange` value to a `Range` value, you ask? `NSRange` is an Objective-C structure whereas `Range` is its Swift equivalent. They are similar, but not exactly the same.

So, while an `NSRange` parameter is used by the `UITextField` — which internally and historically is Objective-C based — in its delegate method, in our Swift code, if we



wanted to do any String operations, such as `replacingCharacters`, then we need a Range value instead. Swift methods generally use Range values and do not understand NSRange values. This is why we converted the NSRange value to a Swift-understandable Range value.

There was a different way to approach this problem as well, though it might not be as "Swift-y." We could have converted the Swift `String` value into its Objective-C equivalent: `NSString`. Since Swift is still young, its `String` handling methods aren't as good ... but they are getting better. `NSString` is considered by some to be more powerful and often easier to use than Swift's own `String`.

`String` and `NSString` are "bridged," meaning that you can use `NSString` in place of `String`. `NSString` too has a `replacingCharacters(in:with:)` method and that method takes an `NSRange` as a parameter!

So, you could have simply converted the `String` value to an `NSString` value and then used the `NSString replacingCharacters(in:with:)` method with the passed in range value instead of the above code. But personally, I prefer to use Swift types and classes in my code as much as possible. So, I opted to go with the solution above.

By the way, `String` isn't the only object that is bridged to an Objective-C type. Another example is `Array` and its Objective-C counterpart `NSArray`. Because the iOS frameworks are written in a different language than Swift, sometimes these little Objective-C holdovers pop up when you least expect them. Once you have the new text, you check if it's empty and enable or disable the Done button accordingly:

```
if newText.isEmpty {  
    doneBarButton.isEnabled = false  
} else {  
    doneBarButton.isEnabled = true  
}
```

However, you could simplify the above code even further. Since `newText.isEmpty` returns a `true` or `false` value, you can discard the `if` condition and use the value returned by `newText.isEmpty` to decide whether the Done button should be enabled or not.

```
doneBarButton.isEnabled = !newText.isEmpty
```

Basically, if the text is not empty, enable the button. Otherwise, don't enable it. That's much more compact and concise, right?

Remember this trick: Whenever you see code like this:

```
if some condition {  
    something = true  
} else {  
    something = false  
}
```

You can write it simply as:

```
something = (some condition)
```

In practice, it doesn't really matter which version you use. I prefer the shorter one. That's what the pros do. Just remember that comparison operators such as == and > always return true or false, so the extra if really isn't necessary.

► Run the app and type some text into the text field. Now, remove that text and you'll see that the Done button in the navigation bar properly gets disabled when the text field becomes empty.

You can find the project files for the app up to this point under **23-Edit High Score Screen** in the Source Code folder.



24

Chapter 24: Delegates & Protocols

Eli Ganim

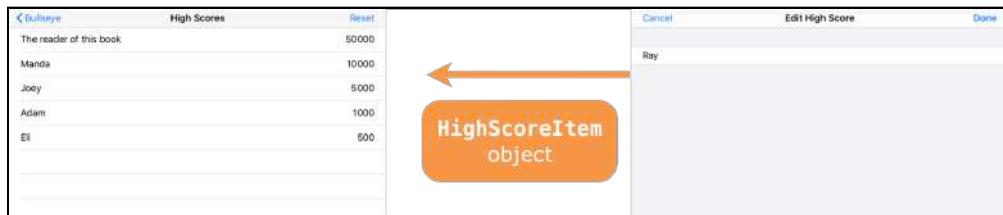
You now have an Edit High Score screen showing a keyboard that lets the user enter text. The app also properly validates the input so that you'll never end up with text that's empty.

But how do you get this text into `HighScoreItem` and add it to `items` on the High Scores screen? That's the topic that this chapter will explore.



Updating HighScoreItem

For editing to work, you'll have to get the Edit High Score screen to notify the High Scores View Controller of the updated HighScoreItem. This is one of the fundamental tasks that every iOS app needs to do: Send messages from one view controller to another.



Sending a HighScoreItem object to the screen with the items array

The messy way

Exercise: How would you tackle this problem? `done()` needs to update `HighScoreItem` with the text from the text field, which is easy, then update it in the Table view in `HighScoreViewController`, which is not so easy.

Maybe you came up with something like this:

```
class EditHighScoreViewController: UITableViewController, ...  
{  
    // This variable refers to the other view controller  
    var highScoresViewController: HighScoresViewController  
  
    @IBAction func done() {  
        highScoreItem.name = textField.text!  
  
        // Directly call a method from HighScoresViewController  
        highScoresViewController.update(item)  
    }  
}
```

In this scenario, `EditHighScoreViewController` has a variable that refers to the `HighScoresViewController`, and `done()` calls its `update()` method with the new `HighScoreItem`. This will work, but it's not the iOS way.

The big downside to this approach is that it shackles these two view controller objects together. As a general principle, if screen A launches screen B then you don't want screen B to know too much about the screen that invoked it, A. The less B knows of A, the better.

Giving `EditHighScoreViewController` a direct reference to `HighScoresViewController` prevents you from opening the Edit High Score screen from somewhere else in the app. It can only ever talk back to `HighScoresViewController`. That's a big disadvantage.

You won't need to do this in `Bullseye`, but in many apps, it's common for one screen to be accessible from multiple places. Examples include a login screen that appears after the app has logged a user out for inactivity, or a details screen that shows more information about a tapped item no matter where that item is in the app. You'll see an example of this in the next app.

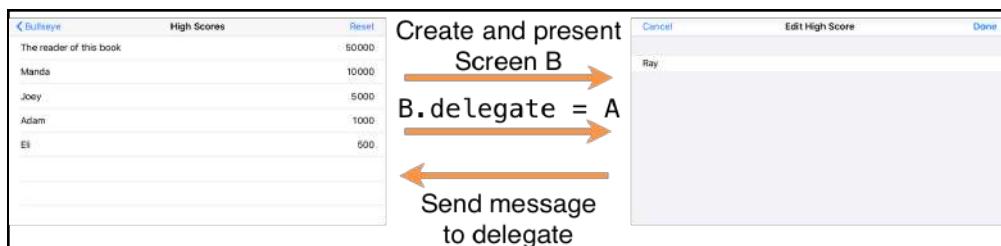
Therefore, it's best if `EditHighScoreViewController` doesn't know anything about `HighScoresViewController`. But if that's the case, how can you make the two communicate?

The solution is to make your own **delegate**.

The delegate way

You've already seen delegates in a few different places: The Table view has a delegate that responds to taps on the rows. The text field has a delegate that you used to validate the length of the text. The app also has something named the `AppDelegate` (see the project navigator).

It seems like you can't turn a corner in this place without bumping into a delegate. The delegate pattern is commonly used to handle the situation you find yourself in: Screen A opens screen B. At some point screen B needs to communicate back to screen A, usually when it closes. The solution is to make screen A the delegate of screen B so that B can send its messages to A whenever it needs to.



Screen A launches screen B and becomes its delegate

The cool thing about the delegate pattern is that screen B doesn't really know anything about screen A. It knows that *some* object is its delegate, but it doesn't really care who that is. Just like how UITableView doesn't really care about your view controller, only that it delivers Table view cells when the Table view asks for them.

This principle, where screen B is independent of screen A and yet can still talk to it, is called **loose coupling** and is good software design practice.

You'll use the delegate pattern to let the `EditHighScoreViewController` send notifications back to the `HighScoresViewController` without it having to know anything about the latter. Delegates go hand-in-hand with **protocols**, a prominent feature of the Swift language.

The delegate protocol

- At the top of `EditHighScoreViewController.swift`, add the following after the `import` line but before the `class` line — it's not part of the `EditHighScoreViewController` object:

```
protocol EditHighScoreViewControllerDelegate: class {
    func editHighScoreViewControllerDidCancel(
        controller: EditHighScoreViewController)
    func editHighScoreViewController(
        controller: EditHighScoreViewController,
        didFinishEditing item: HighScoreItem)
}
```

This defines `EditHighScoreViewControllerDelegate`. You should recognize that the lines inside `protocol { ... }` block as method declarations, but unlike the previous methods you've seen, they don't have any source code in them. The protocol just lists the names of the methods.

Think of the delegate protocol as a contract between screen B, or the Edit High Score View Controller in this case, and any screens that wish to use it.

Are you wondering why you have the keyword `class` after the colon in the protocol name? You might have noticed that the syntax for the protocol declaration looks similar to the one you've used to declare classes previously, giving the name of your class followed by a colon and then specifying the class your class inherited from.

You're seeing exactly the same thing here with protocols: You can have one protocol inherit from another protocol, but you can also specify a particular type of object to adopt your protocol. The `class` keyword identifies that you want to limit `EditHighScoreViewControllerDelegate` to class types.

If you're asking why that is, it's because you mark any references to this protocol as weak. To have weak references, you need a protocol that can only be used with a **reference type**.

You'll read about weak references a bit further along in this chapter; this might all become a little bit clearer at that point.

Protocols [TODO: delete me?]

In Swift, a **protocol** has nothing to do with computer networks or meeting royalty, it's simply a name for a group of methods.

A protocol doesn't usually implement any of the methods it declares. It just says: Any object that conforms to this protocol must implement methods X, Y and Z. There are special cases where you might want to provide a default implementation for a protocol, but that's an advanced topic that you don't need to get into right now.

The two methods listed in `EditHighScoreViewControllerDelegate` are:

- `editHighScoreViewControllerDidCancel(_:)`
- `editHighScoreViewController(_:didFinishAdding:)`

Delegates often have very long method names!

The first method is for when the user presses **Cancel**, while the second is for when they press **Done**. In the latter case, `didFinishAdding` passes along the updated `HighScoreItem`.

For `HighScoresViewController` to conform to this protocol, it must provide implementations for these two methods. From then on, you can refer to `HighScoresViewController` using the protocol name instead of the class name.

If you've programmed in other languages, you may recognize protocols as being very similar to **interfaces**.

In `EditHighScoreViewController`, you can use the following to refer back to `HighScoresViewController`:

```
var delegate: EditHighScoreViewControllerDelegate
```

The variable `delegate` is nothing more than a reference to *some* object that implements the methods of `EditHighScoreViewControllerDelegate`. You can send messages to the object that the `delegate` variable references without knowing what kind of object it really is.

Of course, *you* know the object referenced by `delegate` is the `HighScoresViewController`, but `EditHighScoreViewController` doesn't need to be aware of that. All it sees is some object that implements its delegate protocol.

If you wanted, you could make some other object implement the protocol; `EditHighScoreViewController` would be perfectly fine with that. That's the power of delegation: You have removed – or **abstracted** away – the dependency between the `EditHighScoreViewController` and the rest of the app.

It may seem a little overkill for a simple app such as this, but delegates are one of the cornerstones of iOS development. The sooner you master them, the better!

Notifying the delegate

You're not done in `EditHighScoreViewController.swift` yet. The view controller needs a property that it can use to refer to the delegate; you'll take care of that now.

► Add this inside the `EditHighScoreViewController` class, below the outlets:

```
weak var delegate: EditHighScoreViewControllerDelegate?
```

It looks like a regular instance variable declaration, with two differences: `weak` and the question mark.

Delegates are usually declared as being **weak**. This is not a statement of their moral character, but rather a way to describe the relationship between the view controller and its delegate. Delegates are also **optional**, as indicated by the question mark, which you learned a bit about in the previous chapter.

You'll learn more about what that means in a moment.

► Add this below the `delegate` declaration:

```
var highScoreItem: HighScoreItem!
```



You'll use this to store the item you're editing.

- Replace the `cancel()` and `done()` actions with the following:

```
@IBAction func cancel() {
    delegate?.editHighScoreViewControllerDidCancel(self)
}

@IBAction func done() {
    highScoreItem.name = textField.text!

    delegate?.editHighScoreViewController(self, didFinishEditing:
    highScoreItem)
}
```

Now, look at the changes you made. When the user taps the Cancel button, you send the `editHighScoreViewControllerDidCancel(_:_)` message back to the delegate.

You do something similar for the Done button, except that the message is `editHighScoreViewController(_:_didFinishEditing:_)` and you pass along `HighScoreItem`, which has the text string from the text field.

Note: It's customary for the delegate methods to have a reference to their owner as the first (or only) parameter.

Doing this is not required, but it's a good idea. For example, in the case of Table views, an object may be a delegate or data source for more than one Table view. In that case, you'll need to be able to distinguish between those Table views. To allow for this, the Table view delegate methods have a parameter for the `UITableView` that sent the notification. Having this reference also saves you from having to make an `@IBOutlet` for the Table view.

That explains why you pass `self` to your delegate methods. Recall that `self` refers to the object itself: `EditHighScoreViewController`, in this case. It's also why all the delegate method names start with `editHighScoreViewController`.

- Run the app and try the Cancel and Done buttons. They no longer work!

Hopefully, you're not too surprised! The Edit High Score screen now depends on a delegate to make it close, but you haven't told it who its delegate is yet.

That means the `delegate` property has no value and the messages aren't being sent to anyone – there is no one listening for them.



Optionals

You read a few times before that variables and constants in Swift must always have a value. In other programming languages, the special symbol `nil` or `NULL` is often used to indicate that a variable has no value. Swift doesn't allow this for normal variables.

The problem with `nil` and `NULL` is that they frequently cause apps to crash. If an app attempts to use a variable that is `nil` when you expect it to have a value, the app will crash. This is the dreaded **null pointer dereference** error.

Swift avoids these crashes by preventing you from using `nil` with regular variables.

However, sometimes a variable does need to have "no value." In that case, you can make it an **optional**. You mark something as optional in Swift using either a question mark `?` or an exclamation point `!`.

Only variables that are optional can have a `nil` value.

You've already seen the question mark used with `IndexPath?`, the return type of `tableView(_:willSelectRowAt:)`. Returning `nil` from this method is a valid response; it means that the table should not select a particular row.

The question mark tells Swift that it's OK for the method to return `nil` instead of an actual `IndexPath` object.

Variables that refer to a delegate are usually marked as optional, too. You can tell because there's a question mark behind the type:

```
weak var delegate: EditHighScoreViewControllerDelegate?
```

Thanks to the `?`, it's perfectly acceptable for a delegate to be `nil`.

You may be wondering why the delegate would ever be `nil`. Doesn't that negate the idea of having a delegate in the first place? Well, there are two reasons.

Often, delegates are truly optional; a `UITableView` works fine even if you don't implement any of its delegate methods, although you do need to provide at least some of its data source methods.

More importantly, when you load `EditHighScoreViewController` from the storyboard and instantiate it, it won't know right away who its delegate is. Between the time the view controller is loaded and the delegate is assigned, the `delegate` variable will be `nil`. And variables that can be `nil`, even if only temporarily, must be optionals.

When `delegate` is `nil`, you don't want `cancel()` or `done()` to send any of the messages. Doing that would crash the app because there is no one to receive the messages. Swift has a handy shorthand for skipping the work when `delegate` is not set:

```
delegate?.editHighScoreViewControllerDidCancel(self)
```

Here the `?` tells Swift not to send the message if `delegate` is `nil`. You can read this as, “Is there a delegate? Then send the message.” This practice is called **optional chaining** and it's used a lot in Swift.

In this app, `delegate` should never be `nil` – that would get users stuck on the Edit High Score screen. However, Swift doesn't know that, so you'll have to pretend that it can happen anyway and use optional chaining to send messages to the delegate.

Optionals aren't common in other programming languages, so they may take some getting used to. I find that optionals do make programs clearer – most variables never have to be `nil`, so it's good to prevent them from becoming `nil` and avoid potential sources of bugs. Remember, if you see `?` or `!` in a Swift program, you're dealing with optionals. In the course of this app, I'll come back to this topic a few more times and explain the finer points of using optionals in more detail.

Conforming to the delegate protocol

Before you can give `EditHighScoreViewController` its delegate, you need to make `HighScoresViewController` suitable to play the role of a delegate.

► In `HighScoresViewController.swift`, change the class line to the following (this all goes on one line):

```
class HighScoresViewController: UITableViewController,  
    EditHighScoreViewControllerDelegate {
```

This tells the compiler that `HighScoresViewController` now promises to follow the `EditHighScoreViewControllerDelegate` protocol. Or, in programming terminology, that it **conforms** to the `EditHighScoreViewControllerDelegate` protocol.

Xcode should now throw up an error: “Type `HighScoresViewController` does not conform to protocol `EditHighScoreViewControllerDelegate`.”



A screenshot of Xcode showing a code editor with a red callout box highlighting a warning. The warning message says: "Type 'HighScoresViewController' does not conform to protocol 'EditHighScoreViewControllerDelegate'". Below the message is a button labeled "Fix".

Xcode warns about not conforming to protocol

That is correct: You still need to add the methods that are listed in `EditHighScoreViewControllerDelegate`. With the latest version of Xcode, there's an easy way to get started fixing this issue — see that **Fix** button? Simply click it.

Xcode will add in the stubs, or the bare minimum code, for the missing methods. You'll have to add in the actual implementation for each method, of course.

- Add the implementations for the protocol methods to `HighScoresViewController`:

```
// MARK:- Edit High Score ViewController Delegates
func editHighScoreViewControllerDidCancel(
    controller: EditHighScoreViewController) {
    navigationController?.popViewControllerAnimated(true)
}
func editHighScoreViewController(
    controller: EditHighScoreViewController,
    didFinishEditing item: HighScoreItem) {
    navigationController?.popViewControllerAnimated(true)
}
```

Currently, both methods simply close the Edit High Score screen, which is what the `EditHighScoreViewController` used to do with its `cancel()` and `done()` actions. You've simply moved that responsibility to the delegate. You haven't added the code that updates the `HighScoreItem` object in the Table view yet. You'll do that in a moment, but there's something else you need to do first.

5.

You've done the first four steps, so there's just one more thing you need to do — step 5: tell `EditHighScoreViewController` that `HighScoresViewController` is its delegate.

The proper place to do that is in the `prepare(for:sender:)` method, also known as **prepare-for-segue**.

UIKit invokes `prepare(for:sender:)` right before it performs a segue from one screen to another. Remember that the segue is the arrow between two view controllers in the storyboard.

Using `prepare-for-segue` allows you to pass data to the new view controller before displaying it. You'll usually do this by setting one or more of the new view controller's properties.

► Add this method to `HighScoresViewController.swift`:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    // 1
    let controller = segue.destination as!
    EditHighScoreViewController
    // 2
    controller.delegate = self
    // 3
    if let indexPath = tableView.indexPath(for: sender as!
    UITableViewCell) {
        controller.highScoreItem = items[indexPath.row]
    }
}
```

This is what the above code does, step by step:

1. You find the new view controller that you want to display in `segue.destination`. `destination` is of type `UIViewController`, since the new view controller could be any view controller subclass.

To handle that issue, you **cast** `destination` to `EditHighScoreViewController` to get a reference to an object with the right type. The `as!` keyword is known as a **type cast** or a **force downcast** since you are casting an object of one type to a different type.

Do note that if you downcast objects of completely different types, you might get a `nil` value. Casting works here because `EditHighScoreViewController` is a sub-class of `UIViewController`.[TODO: FPE: Is this spacing OK? I can't see how it renders, so I can't verify.]

2. Once you have a reference to `EditHighScoreViewController`, you set its `delegate` property to `self`. This tells `EditHighScoreViewController` that from now on, the object identified as `self` is its delegate. But what is “`self`” here? Well, since you're editing `HighScoresViewController.swift`, `self` refers to `HighScoresViewController`.

3. The parameter `sender` contains a reference to the control that triggered the segue. In this case, it refers to the Table view cell which you tapped.

You use that `UITableViewCell` object to find the Table view row number by looking up the corresponding index path using `tableViewIndexPath(for:)`.

The return type of `indexPath(for:)` is `IndexPath?`, an optional, meaning it could return `nil`. That's why you need to unwrap this optional value with `if let` before you can use it.

Once you have the index path, you obtain the `HighScoreItem` object to edit and you assign it to `EditHighScoreViewController`'s `highScoreItem` property.

Excellent! `HighScoresViewController` is now the delegate of `EditHighScoreViewController`. It took some work, but you're almost set now.

Now that you have a reference to the item you're editing, you can make the screen actually show the name of the current high scorer before you edit it.

- Open `EditHighScoreViewController.swift` and add this to `viewDidLoad()`:

```
textField.text = highScoreItem.name
```

- Run the app, click on any high score and see that the edit item screen now has the current name already in the text field.

Updating the table view

If you change the name and click **Done**, you'll update the item itself, but the Table view will still show the old value. That's because you didn't tell the Table view to refresh the cell after the data changed. Time to fix it!

- Change the implementation of the `didFinishEditing` delegate method in `HighScoresViewController.swift` to the following:

```
func editHighScoreViewController(_ controller: EditHighScoreViewController, didFinishEditing item: HighScoreItem) {
    // 1
    if let index = items.firstIndex(of: item) {
        // 2
        let indexPath = IndexPath(row: index, section: 0)
        let indexPaths = [indexPath]
        // 3
        tableView.reloadRows(at: indexPaths, with: .automatic)
```

```
    }
    // 4
    PersistenceHelper.saveHighScores(items)
    navigationController?.popViewController(animated:true)
}
```

Here's what this new code does:

1. To update the cell, you need the `IndexPath` of that cell. The cell index is the same as the index of `HighScoreItem` in the `items` array. You can use `firstIndex(of:)` to return that index.

Now, it won't happen here, but it's possible that you could use `index(of:)` on an object that's not actually in the array. To account for that possibility, `index(of:)` doesn't return a normal value, it returns an optional. If the object is not part of the array, the returned value is `nil`.

That's why you need to use `if let` here to unwrap the return value from `index(of:)`.

2. You create a new array of `IndexPath` objects to update. In this case, there's only one cell you want to update so you create an array with a single index path.
3. You tell the Table view to update that specific cell. Keep in mind that, should you ever need to, you can also use `insertRows(at:with:)` and `deleteRows(at:with:)`.
4. Eventually, you save the updated list of high scores and pop [TODO: FPE: Is "pop" right word? Should it be "populate" instead?] the view controller

► Try to build the app. Oops, Xcode has found another reason to complain:

```
if let index = items.firstIndex(of: item) {
    let indexPath = IndexPath(item: index!, section: 0)
    let indexPaths = [indexPath]
    tableView.reloadRows(at: indexPaths, with: .automatic)
}
```

New Xcode error

Xcode displays this error because you can't use `firstIndex(of:)` on just any array, or collection of objects. An object has to be "equatable" if you are to use `firstIndex(of:)` on an array of that object type.

That's because `firstIndex(of:)` needs a way to compare the object that you're looking for against the objects in the array, to see if they are equal.

Your `HighScoreItem` object does not have any functionality for that yet. There are a few ways you can fix this, but in this case, you'll use the easy one.

- In `HighScoreItem.swift`, change the class line to:

```
class HighScoreItem : NSObject, Codable {
```

If you've programmed in Objective-C before, you'll be familiar with `NSObject`.

Almost all objects in Objective-C programs are based on `NSObject`. It's the most basic building block that iOS provides, and it offers a bunch of useful functionality that standard Swift objects don't have.

You can write many Swift programs without having to resort to `NSObject` but in times like these, it comes in handy.

Building `HighScoreItem` on top of `NSObject` is enough to satisfy the “equatable” requirement. In case you're interested, the other way to do this would have been to specify that `HighScoreItem` conforms to the `Equatable` protocol. But then you'd have to implement an additional method to indicate how the comparison of two `HighScoreItem` instances would happen. So going with `NSObject` conformance is easier, for the time being.

- Build and run the app again and verify that editing items works now. Excellent!

Weak

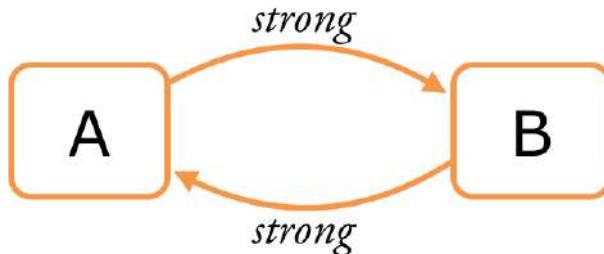
I promised you an explanation of the `weak` keyword. Relationships between objects can be weak or strong. You use weak relationships to avoid an **ownership cycle**.

When object A has a strong reference to object B, and at the same time object B also has a strong reference back to A, then these two objects are involved in a dangerous kind of romance: An ownership cycle.

Normally, you destroy, or **deallocate**, an object when there are no more strong references to it. But because A and B have strong references to each other, they keep each other alive.

The result is a potential **memory leak**, where an object isn't destroyed, even though it should be, and the memory for its data is never reclaimed. With enough such leaks, iOS will run out of available memory and your app will crash. I told you it was dangerous!

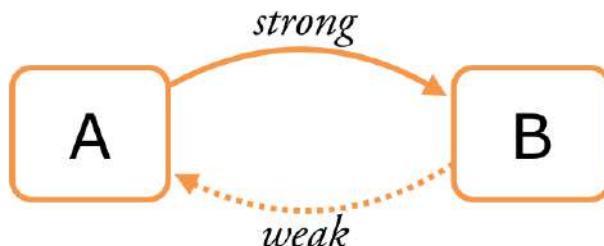
Due to the strong references between them, A owns B and at the same time, B also owns A:



To avoid ownership cycles you can make one of these references weak.

In the case of a view controller and its delegate, screen A usually has a strong reference to screen B, but B only has a weak reference back to its delegate, A.

Because of the weak reference, B no longer owns A:



Now there is no ownership cycle.

Such cycles can occur in other situations, too, but they are most common with delegates. Therefore, you should always make delegates weak.

There is another relationship type that is similar to weak and that you can also use delegates: unowned. The difference is that weak variables can be `nil`. However, you don't need to worry about that right now.

Usually, you also declare `@IBOutlets` with the `weak` keyword. This isn't done to avoid an ownership cycle, but to make it clear that the view controller isn't really the owner of the views from the outlets.

In the course of this book, you'll learn more about `weak`, `strong`, `optionals` and the relationships between objects. These are important concepts in Swift, but they may take a while to make sense. If you don't understand them immediately, don't lose

any sleep over it!

You can find the project files for the app up to this point under **24-Delegates and Protocols** in the Source Code folder.



Chapter 25: The Final App

Eli Ganim

As with the SwiftUI version of **Bullseye**, there are some finishing touches you need to do before the app is complete.

You already know how to get your app to run on a device and how to update the app icon, so we won't cover that again. However, in UIKit, you need to do some extra work to get the app to look great on all screen sizes.

You've been developing and testing for a 4.7" screen like those found on devices such as the iPhone 8. But what about other iPhones such as the 5.5-inch iPhone Plus or the 5.8-inch iPhone X, which have bigger screens? Or the iPad with its various screen sizes? Will the game work correctly on all these different screen sizes?

This chapter covers the following:

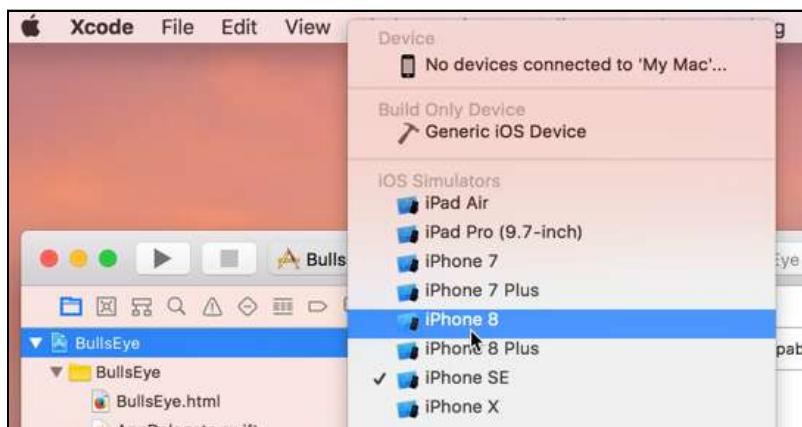
- **Supporting different screen sizes:** Ensuring that the app will run correctly on all the different iPhone and iPad screen sizes.
- **Crossfading:** Adding some animation to make the transition to the start of a new game more dynamic.



Supporting different screen sizes

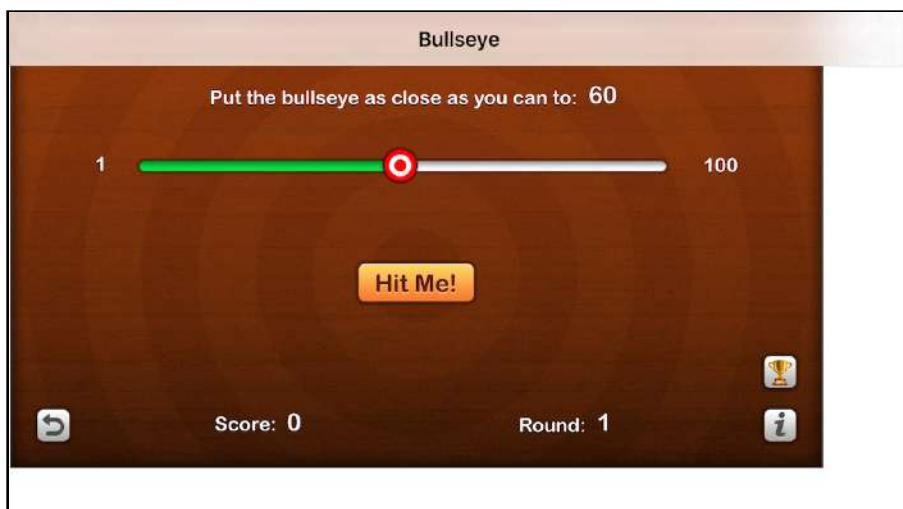
First, check if there is, indeed, an issue running Bullseye on a device with a larger screen.

- To see how the app looks on a larger screen, run the app on an iPhone simulator like the **iPhone 8 Plus**. You can switch between simulators using the selector at the top of the Xcode window:



Using the scheme selector to switch to the iPhone 8 simulator

The result might not be what you expected:



On the iPhone 8 simulator, the app doesn't fill the entire screen

Obviously, this won't do. Not everybody is going to be using a 4.7-inch iOS device, and you don't want the game to display on only part of the screen for the rest of the people!

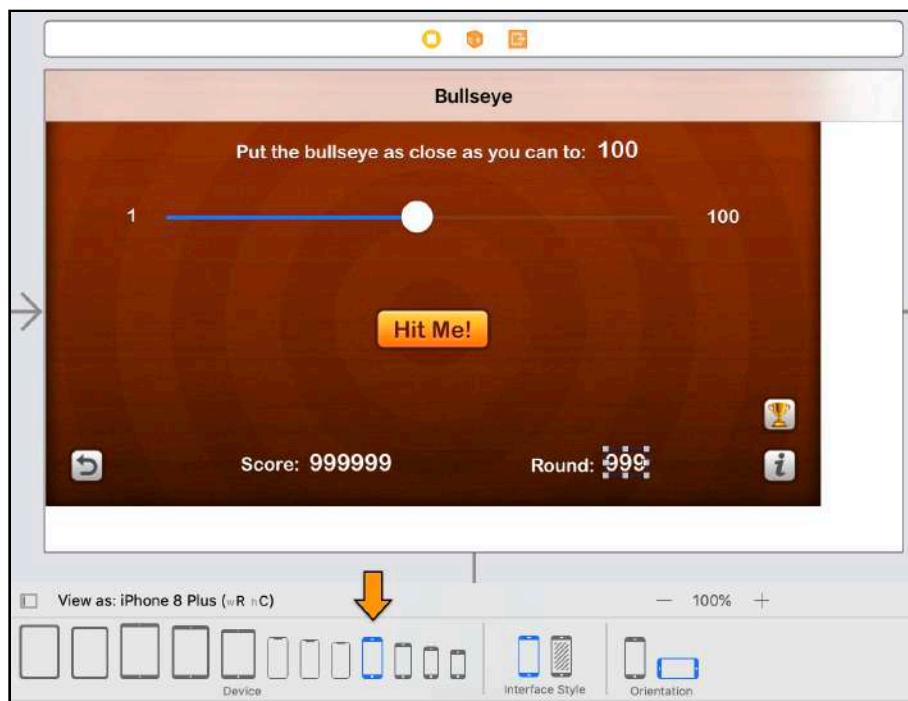
This is a good opportunity to learn about **Auto Layout**, a core UIKit technology that makes it easy to support many different screen sizes in your apps, including the larger screens of the 4.7-inch, 5.5-inch and 5.8-inch iPhones as well as the iPad.

Tip: You can use the **Window > Scale** menu to resize a simulator if it doesn't fit on your screen. Some of those simulators, like the one for the iPad, can be monsters! Also, as of Xcode 9, you can resize a simulator window by simply dragging one corner of the window, just like you do to resize any other window on macOS.

Interface Builder has a few handy tools to help you make the game fit on any screen.

The background image

► Go to **Main.storyboard**. Open the **View as:** panel at the bottom and choose the **iPhone 8 Plus** device. You may need to change the orientation back to landscape.



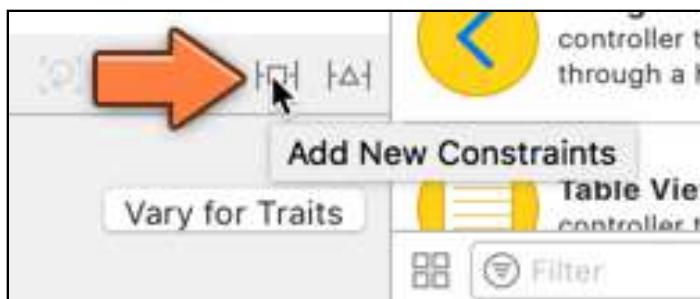
Viewing the storyboard on iPhone 8 Plus

The storyboard should look like your screen from the iPhone 8 Plus simulator. This shows you how changes on the storyboard affect the bigger iPhone screens.

First, let's fix the background image. At its normal size, the image is too small to fit on the larger screens.

This is where Auto Layout comes to the rescue.

- In the storyboard, select the **Background image view** on the main **view controller** and click the small **Add New Constraints** button at the bottom of the Xcode window:

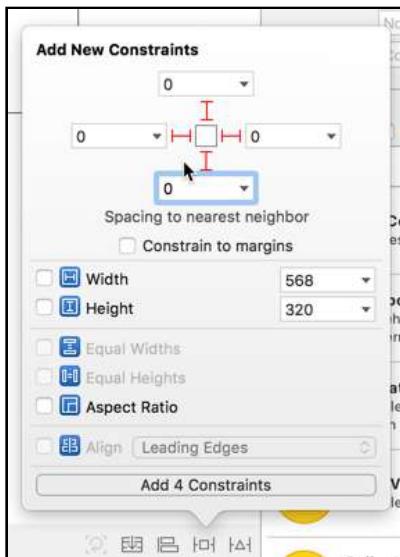


The Add New Constraints button

This button lets you define relationships, called **constraints**, between the currently-selected view and other views in the scene. When you run the app, UIKit evaluates these constraints and calculates the final layout of the views. This probably sounds a bit abstract, but you'll soon see how it works in practice. In order for the background image to stretch from edge to edge on the screen, the left, top, right and bottom edges of the image should be flush against the screen's edges. You can use Auto Layout to do this.

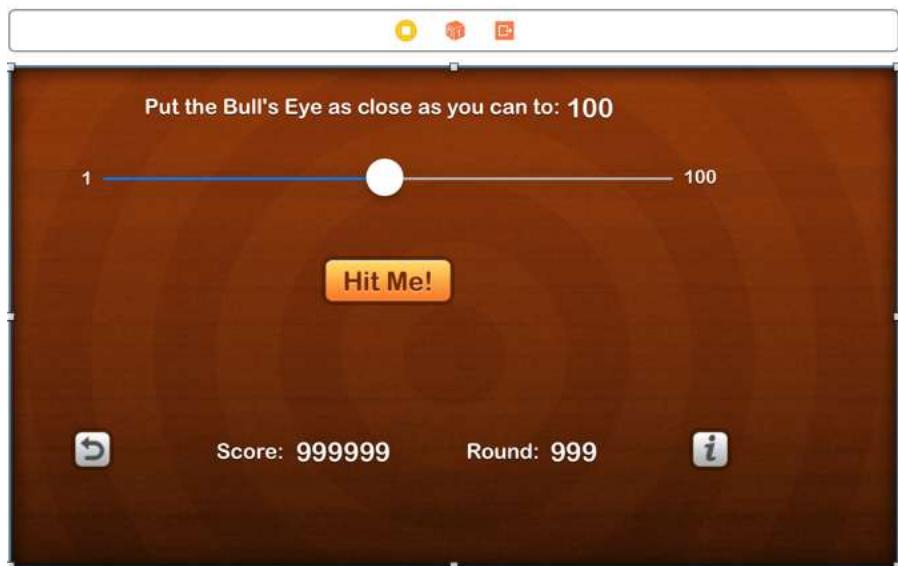
- In the **Add New Constraints** menu, set the **left**, **top**, **right** and **bottom** spacing to zero and make sure that you've enabled the red I-beam markers next to (or above/below) each item.

The red I-beams indicate which constraints you've enabled when adding new constraints:



Using the Add New Constraints menu to position the background image

- Press **Add 4 Constraints** to finish. The background image will now cover the view fully. Press **Undo** and **Redo** a few times to see the difference.



The background image now covers the whole view

You might have also noticed that the Document Outline now has a new item called **Constraints**:



The new Auto Layout constraints appear in the Document Outline

There should be four constraints listed there, one for each edge of the image.

- Run the app again on the iPhone 8 Plus Simulator and also on the iPhone 8 Simulator. In both cases, the background should display correctly now. Of course, the other controls are still off-center, but you'll fix that soon.

If you use the **View as** panel to switch the storyboard back to the iPhone 8, the background should display correctly there, too.

The About screen

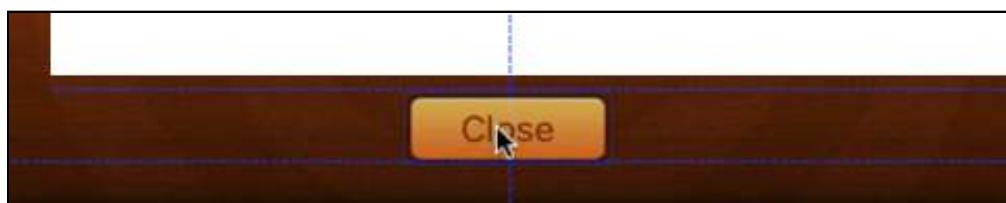
Let's repeat the background image fix for the About screen, too.

- Use the **Add New Constraints** button to pin the About screen's background image view to the parent view.

The background image is now fine. Of course, the Close button and web view are still completely off.

- In the storyboard, drag the **Close** button so that it snaps to the center of the view as well as to the bottom guide.

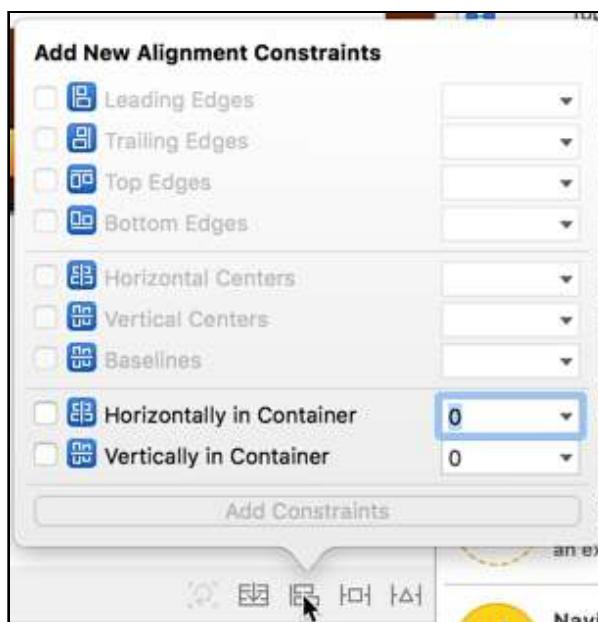
Interface Builder shows a handy guide, the dotted blue line, near the edges of the screen. This is useful for aligning objects by hand.



The dotted blue lines are guides that help position your UI elements

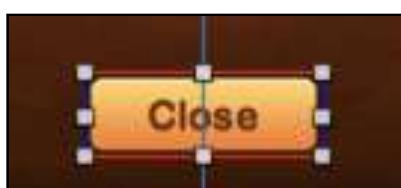
You want to create a centering constraint that keeps the Close button in the middle of the screen, regardless of how wide the screen is.

- Click the **Close** button to select it. From the **Align** menu, which is to the left of the Add New Constraints button, choose **Horizontally in Container** and click **Add 1 Constraint**.



The Align menu

Interface Builder now draws a blue bar to represent the constraint. It draws a red box around the button as well.



The Close button has red borders

That's a problem: The red box indicates that something is wrong with the constraints, which usually means that there aren't enough of them. The thing to remember is this: For each view, there must always be enough constraints to define both its position and its size.

The Close button already knows its size – you typed it into the Size inspector earlier.

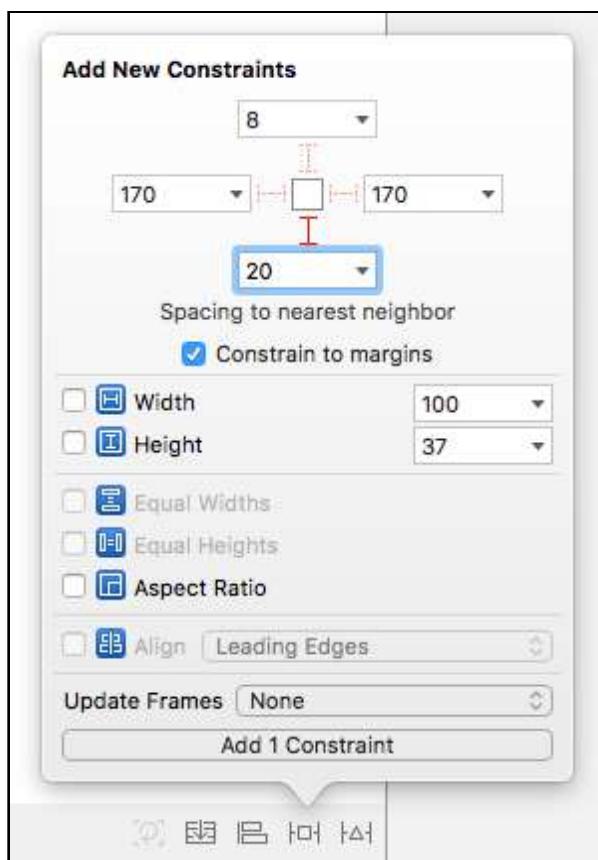
But for its position, there's only a constraint for the X-coordinate (the alignment in the horizontal direction). You also need to add a constraint for the Y-coordinate.

As you've noticed, there are different types of constraints — there are alignment constraints and spacing constraints, like the ones you added via the Add New Constraints button. In this case, you want to add a spacing constraint. [TODO: FPE: I added this last sentence, please verify it's OK --editor]

- With the **Close** button still selected, click on the **Add New Constraints** button.

You want the Close button to always sit at a distance of 20 points from the bottom of the screen.

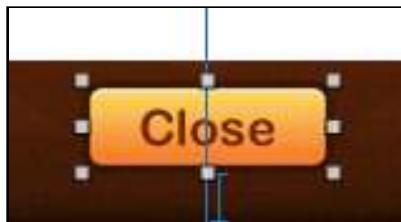
- In the **Add New Constraints** menu, in the **Spacing to nearest neighbor** section, set the bottom spacing to **20** and make sure that you've enabled the I-beam above the text box.



The red I-beams determine the sides that you have pinned down

► Click **Add 1 Constraint** to finish.

The red border will now turn blue, meaning that everything is OK:



The constraints on the Close button are valid

If you see orange bars instead of blue ones at this point, then something's still wrong with your Auto Layout constraints:



The views are not positioned according to the constraints

This happens when the constraints are valid (otherwise some, or all, of the bars or borders would be red) but the view is not in the right place in the scene. The dashed orange box off to the side is where Auto Layout has calculated the view should be, based on the constraints you have given it.

To fix this issue, select the **Close** button again and click the **Update Frames** button at the bottom of the Interface Builder canvas.



The Update Frames button

You can also use the **Editor ▶ Resolve Auto Layout Issues ▶ Update Frames** item from the menu bar.

The Close button should always be perfectly centered now, regardless of the device's screen size.

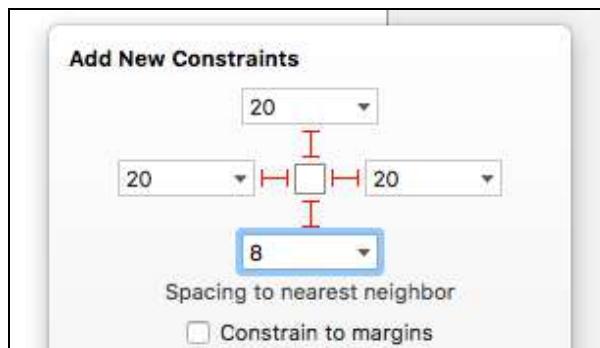
Note: What happens if you don't add any constraints to your views? In that case, Xcode will automatically add constraints when it builds the app. That's why you didn't need to bother with any of this before.

However, these default constraints may not always do what you want. For example, they will not automatically resize your views to accommodate larger (or smaller) screens. If you want that to happen, then it's up to you to add your own constraints. After all, Auto Layout can't read your mind!

As soon as you add just one constraint to a view, Xcode will no longer add any other automatic constraints to that view. From then on, you're responsible for adding enough constraints so that UIKit always knows what the position and size of the view will be.

There's one thing left to fix in the About screen: The web view.

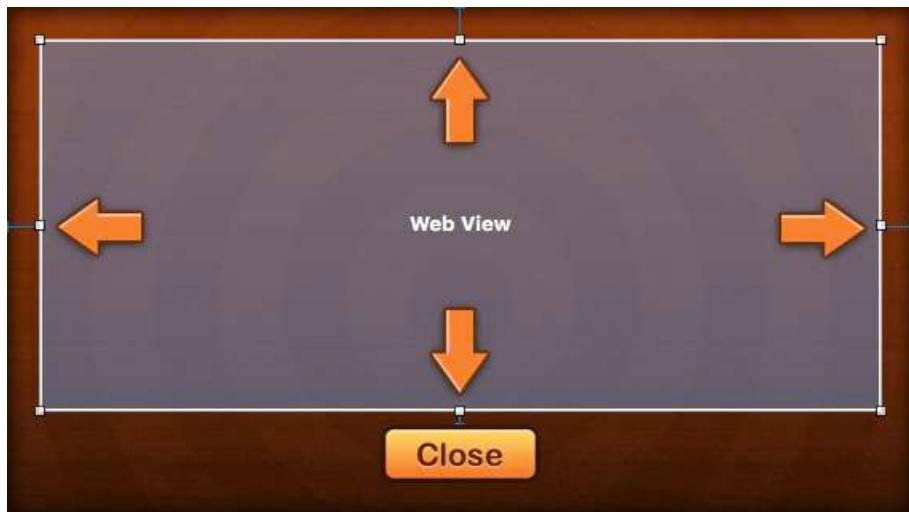
- Select the **Web View** and open the **Add New Constraints** menu. First, make sure you've unchecked **Constrain to margins**. Click all four I-beam icons so they become solid red and then set their spacing to 20 points, except the bottom one which should be 8 points:



Creating the constraints for the web view

- Finish by clicking **Add 4 Constraints**.

There are now four constraints on the web view, indicated by the blue bars on each side:



The four constraints on the web view

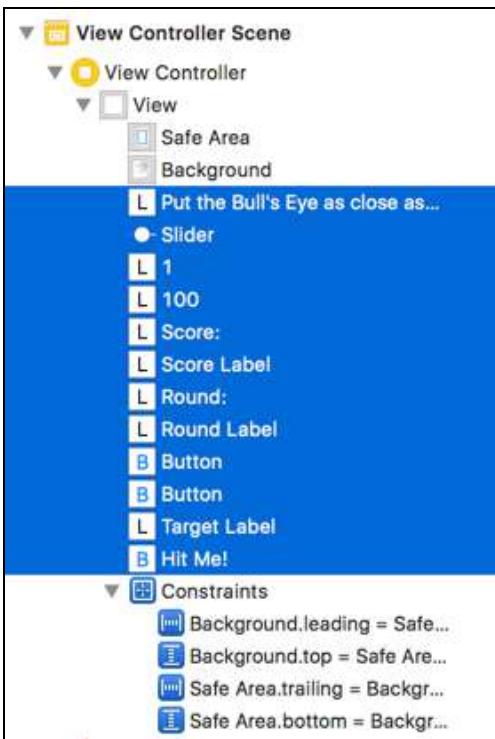
Three of these constraints pin the Web view to the Main view, so that they always resize together, while one connects the Web view to the Close button. This is enough to determine the size and position of the web view in any scenario.

Fixing the rest of the main scene

Back to the main game scene, which still needs some work.

The game now looks a bit lopsided on bigger screens. You'll fix that by placing all the labels, buttons and the slider into a new "Container" view. Using Auto Layout, you'll center that Container view in the screen, regardless of how big the screen is.

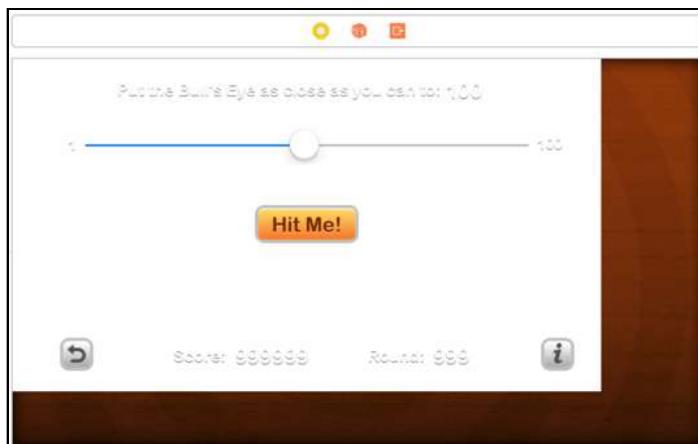
- Select all the labels, the buttons and the slider. You can hold down **Command** and click them individually, but an easier method is to go to the **Document Outline**, click on the first view (for me that's the "Put the Bullseye as close as you can to:" label) and then hold down Shift and click on the last view.



Selecting the views from the Document Outline

You should have selected everything but the background image view.

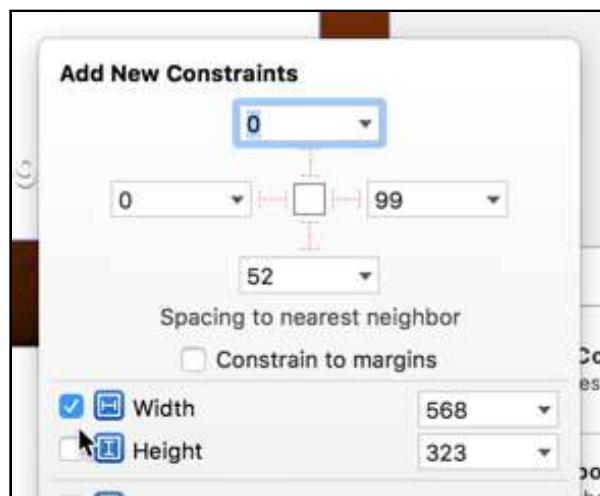
- From Xcode's menu bar, choose **Editor** ▶ **Embed In** ▶ **View**. This places the selected views inside a new container view:



The views are embedded in a new container view

This new view is completely white. You'll want to change this eventually, but for now, it makes it easier to add the constraints.

- Select the newly-added **container view** and open the **Add New Constraints** menu. Check the boxes for **Width** and **Height** to make constraints for them and leave the width and height at the values specified by Interface Builder. Click **Add 2 Constraints** to finish.



Pinning the width and height of the container view

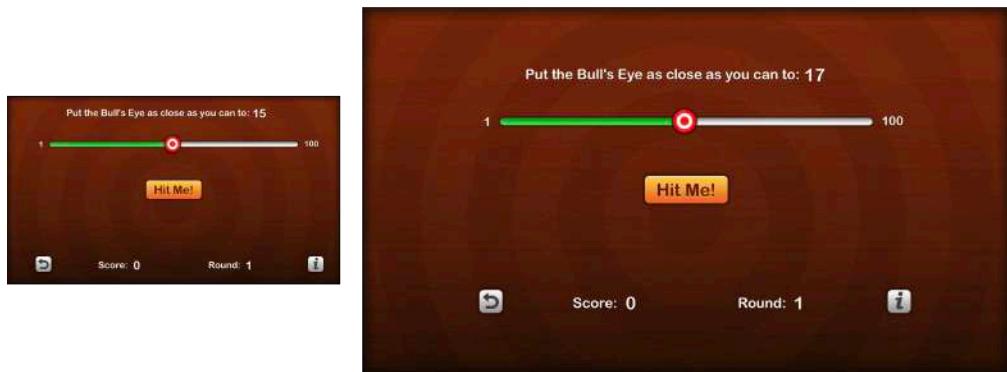
Interface Builder now draws several bars around the view that represent the Width and Height constraints that you just made... but they're red. Don't panic! It only means that there aren't enough constraints yet. No problem, you'll add the missing constraints next.

- With the container view still selected, open the **Align menu**. Check the **Horizontally in Container** and **Vertically in Container** options, then click **Add 2 Constraints**.

All of the Auto Layout bars should be blue now, and the view is perfectly centered.

- Finally, change the **Background** color of the container view to **Clear Color** – in other words, 100% transparent.

You now have a layout that works correctly on any iPhone display! Try it out:



The game running on 4-inch and 5.5-inch iPhones

Auto Layout may take a while to get used to. Adding constraints in order to position UI elements is a little less obvious than just dragging them into place.

But this also buys you a lot of power and flexibility, which you need when you're dealing with devices that have different screen sizes.

You'll learn more about Auto Layout in the other parts of **The iOS Apprentice**.

Exercise: As you try the game on different devices, you might notice something — the controls for the game are always centered on screen, but they do not take up the whole area of the screen on bigger devices!

This is because you set the container view for the controls to be a specific size. If you want the controls to change position and size depending on how much screen space is available, then you have to remove the container view or set it to resize depending on screen size. You then need to set up the Auto Layout constraints for each control separately.

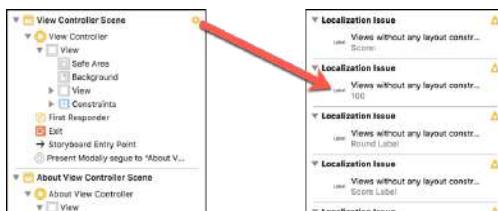
Are you up to the challenge of doing this on your own?

Compiler warning

There's a compiler warning about views without any layout constraints that's been there since almost the first time you created the main game view.

But where are these views without constraints? The Issue navigator certainly does not give you any clue! You have to go to a different screen to find the list of affected views.

If you look at the Document Outline, you'll notice that there's a small yellow circle with an arrow inside it next to the View Controller Scene — an indication that there are some warnings (not errors). If you click this circle, it will bring you to a list of Auto Layout issues in the scene:



List of Auto Layout issues for your scene

You'll notice that all the controls inside the container view have warnings. This is because none of those views have any Auto Layout constraints, so they'll remain at their original positions when the view displays.

So how do you fix these warnings? Simple enough — add some constraints for each view so that Auto Layout can determine the view's size and position when it displays. Of course, you have the necessary knowledge at this point to do that yourself. So I'll leave it to you to tackle as a personal challenge.

If you get stuck, check out the final project provided for this chapter — that should show you how I fixed the issues.

Testing on iPhone X

So it looks as if you got the app working correctly for all devices, right?

Well... not quite!

Try running the app on the iPhone X simulator. You should see something like this:



The game on iPhone X

Whoa! What happened?

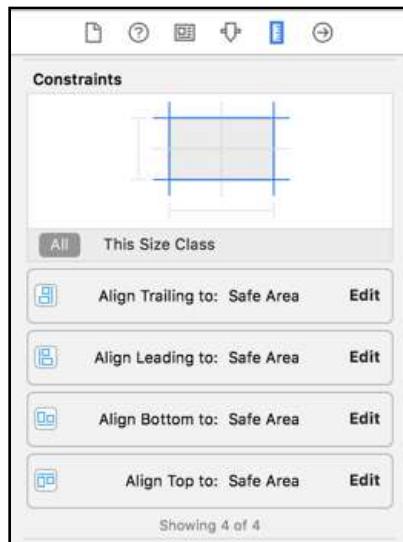
In Xcode 9, Apple introduced a new mechanism to go along with Auto Layout for the iPhone X screen. Since the iPhone X has a notch at the top, you don't want your app to display its content beneath the notch, since that content would not display properly.

So, Xcode 9 has **safe layout guides** that define where it's safe to display content. If you take a look at the Auto Layout constraints for the background image on the main scene (or even the About scene, for that matter), you will notice that the background image aligns to the safe area.

Just to be clear, this is the correct behavior for most apps — you want your content to stick to the safe area. However, in the case of a game like **Bullseye** where you have a custom background, you want the background to cover the entire screen.

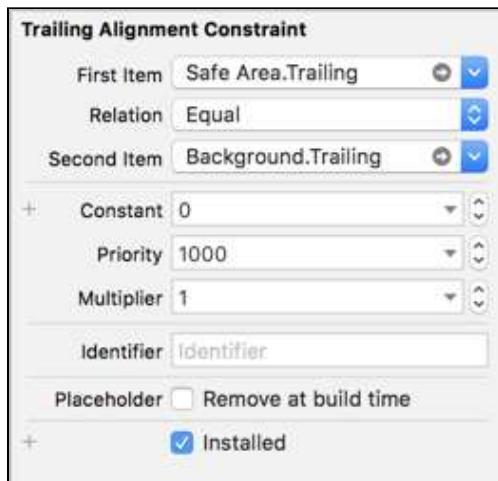
So how do you fix it? Simple enough... just change the image constraints so that they align with the superview, which is the main view for the scene.

- Select the background image in the main scene and switch to the **Size inspector**. It should show the four constraints set on the image:



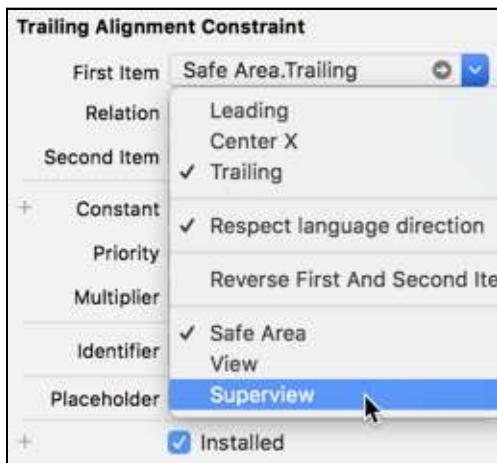
Auto Layout constraints for your image

- Double-click the first one. You should get a detailed editor for that Auto Layout constraint:



Auto Layout constraint editor

- Click the drop-down for **First Item** and change it from **Safe Area** to **Superview**. Also, if the **Constant** field has a non-zero value, change it to 0.



Edit Auto Layout constraints

- Do the same for the other three constraints. Note that for two of the constraints, Safe Area will be the Second Item and not the First Item. So change either one based on which one specifies Safe Area.
- Build and run your app on both the iPhone X simulator and at least one of the other simulators, like the iPhone 8 Plus, to make sure that your changes work correctly on all devices.

Crossfade

There's one final bit of knowledge I want to impart before calling the game complete — Core Animation. This technology makes it easy to create really sweet animations in your apps with just a few lines of code. Adding subtle animations (with the emphasis on subtle!) can make your app a delight to use.

So next, you'll add a simple crossfade after the user presses the **Start Over**, so the transition back to round one won't seem so abrupt.

- In **ViewController.swift**, change `startNewGame()` to:

```
@IBAction func startNewGame() {  
    ...  
    startNewRound()  
    // Add the following lines
```

```
let transition = CATransition()
transition.type = CATransitionType.fade
transition.duration = 1
transition.timingFunction = CAMediaTimingFunction(name:
    CAMediaTimingFunctionName.easeOut)
view.layer.add(transition, forKey: nil)
}
```

I'm not going to go into too much detail here. Suffice it to say you're setting up an animation that crossfades from what is currently on the screen to the changes you're making in `startNewRound()`. You reset the slider to center position and reset the values of the labels.

- Run the app and move the slider so that it's no longer in the center. Press the **Start Over** button and you should see a subtle crossfade animation.



The screen crossfades between the old and new states

UIKit knowledge unlocked!

You're now familiar with SwiftUI and UIKit. Well done!

In the Source Code folder for this book, you can find the complete source code for the UIKit version of the **Bullseye** app. If you're still unclear about something in this chapter, it's a good idea to look at this cleaned-up source code.

If you're feeling exhausted after all that coding, pour yourself a drink and put your feet up for a bit. You've earned it! On the other hand, if you just can't wait to get to grips with more code, go ahead and move on to the next app!



Section 4: My Locations

With this fourth section and the MyLocations app, you get into Swift programming in earnest.

Sure you've already done coding in the previous sections, but this section starts with a good review of all the Swift coding principles you've learned so far and added to it by introducing some new concepts, too. In addition to that, you learn about using GPS coordinates, displaying data on maps, and using the iOS image picker to take photos using your camera or to pick existing images from your photo album. There's a lot of valuable general information on Swift development as well as specific information about building location-aware apps.

This section contains the following chapters:

26. Swift Review: You have made great progress. You have built your first app using UIKit which is some achievement. Whilst we have been writing the apps using Swift, you will need some additional theory to level up your knowledge. In this chapter, we will go into details about some of the Swift language, such as Variables, Constants, Types, Methods & Functions, Loops, and Objects.

27. Get Location Data: Are you ready for the final app challenge? In this chapter, you will commence the final app MyLocations. It's all about using the Core Location framework and displaying using MapKit.

28. Use Location Data: You've learned about getting GPS coordinate information. Now it's time to deal with GPS errors, improving GPS results and testing on real devices to mimic real-world scenarios.



29. Objects vs. Classes: This question will most likely crop up in your next iOS interview. It's time to put the toolbox down and learn some theory. Expect to learn about classes, inheritance, overriding methods and casting an object.

30. The Tag Location Screen: In this chapter, you will be picking up the toolbox again and building our tag location screen. This involves building out some TableViewCells, displaying location info and adding a category picker.

31. Adding Polish: Who doesn't love adding a bit of polish? It's time to start making it look more like an app ready for the App Store. We will improve the experience of the app and add a loading 'HUD'.

32. Saving Locations: At this point, you have an app that obtains GPS coordinates and allows the user to tag the location. We're going to be deep-diving into Core Data, the object persistence framework for iOS apps.

33. The Locations Tab: Now that you can persist the data to Core Data, we're going to explore displaying this data in the TableView. Learn about TableView sections, NSFetchedResults and add functionality to delete tagged locations.

34. Maps: Showing a list of locations is great, but not very visually appealing. In this chapter, you will learn about MapKit, the awesome map view control giving in the iOS toolbox.

35. Image Picker: UIKit comes with a built-in view controller, UIImagePickerController that lets users take new photos or select existing ones. In this chapter explore this controller and how best to display the image on the screen.

36. Polishing the App: You have made it this far! It's time to give MyLocations a complete makeover. Prepare your pixel paintbrush for this chapter and let's get your creative flair at the ready. In this chapter you will cover the map screen improvements by adding icons, polishing the main screen and adding some cool effects to the app.

Chapter 26: Swift Review

Eli Ganim

You have made great progress! You've learned the basics of Swift programming and created two applications from scratch, one of them in SwiftUI and the other in UIKit. You are on the threshold of creating your next app.

A good building needs a good foundation. And in order to strengthen the foundations of your Swift knowledge, you first need some additional theory. There is still a lot more to learn about Swift and object-oriented programming!

In the previous chapters, you saw a fair bit of the Swift programming language already, but not quite everything. Previously, it was good enough if you could more-or-less follow along, but now is the time to fill in the gaps in the theory. So, here's a little refresher on what you've learned so far.

In this chapter, you will cover the following:

- **Variables, constants and types:** the difference between variables and constants, and what a type is.
- **Methods and functions:** what are methods and functions — are they the same thing?
- **Making decisions:** an explanation of the various programming constructs that can be used in the decision making process for your programs.
- **Loops:** how do you loop through a list of items?
- **Objects:** all you ever wanted to know about Objects — what they are, their component parts, how to use them, and how not to abuse them.
- **Protocols:** the nitty, gritty details about protocols.



Variables, constants and types

A **variable** is a temporary container for a specific type of value:

```
var count: Int
var shouldRemind: Bool
var text: String
var list: [ChecklistItem]
```

The **data type**, or just **type**, of a variable determines what kind of values it can contain. Some variables hold simple values such as `Int` or `Bool`, others hold more complex objects such as `String` or `Array`.

The basic types you've used so far are: `Int` for whole numbers, `Float` for numbers with decimals (also known as *floating-point* numbers), and `Bool` for boolean values (`true` or `false`).

There are a few other fundamental types as well:

- `Double`. Similar to a `Float` but with more precision. You will use `Doubles` later on for storing latitude and longitude data.
- `Character`. Holds a single character. A `String` is a collection of `Characters`.
- `UInt`. A variation on `Int` that you may encounter occasionally. The U stands for *unsigned*, meaning the data type can hold positive values only. It's called *unsigned* because it cannot have a negative sign (-) in front of the number. `UInt` can store numbers between 0 and 18 quintillion, but no negative numbers.
- `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`. These are all variations on `Int`. The difference is in how many bytes they have available to store their values. The more bytes, the bigger the values they can store. In practice, you almost always use `Int`, which uses 8 bytes for storage on a 64-bit platform (a fact that you may immediately forget) and can fit positive and negative numbers up to about 19 digits. Those are big numbers!
- `CGFloat`. This isn't really a Swift type but a type defined by the iOS SDK. It's a decimal point number like `Float` and `Double`. For historical reasons, this is used throughout `UIKit` for floating-point values. (The "CG" prefix stands for the **Core Graphics** framework.)

Swift is very strict about types, more so than many other languages. If the type of a variable is `Int`, you cannot put a `Float` value into it. The other way around also won't work: an `Int` won't go into a `Float`.



Even though both types represent numbers of some sort, Swift won't automatically convert between different number types. You always need to convert the values explicitly.

For example:

```
var i = 10
var f: Float
f = i           // error
f = Float(i)   // OK
```

You don't always need to specify the type when you create a new variable. If you give the variable an initial value, Swift uses **type inference** to determine the type:

```
var i = 10          // Int
var d = 3.14        // Double
var b = true        // Bool
var s = "Hello, world" // String
```

The integer value 10, the floating-point value 3.14, the boolean true and the string "Hello, world" are named **literal constants** or just **literals**.

Note that using the value 3.14 in the example above leads Swift to conclude that you want to use a Double here. If you intended to use a Float instead, you'd have to write:

```
var f: Float = 3.14
```

The : Float bit is called a **type annotation**. You use it to override the guess made by Swift's type inference mechanism, since it doesn't always get things right.

Likewise, if you wanted the variable i to be a Double instead of an Int, you'd write:

```
var i: Double = 10
```

Or a little shorter, by giving the value 10 a decimal point:

```
var i = 10.0
```

These simple literals such as 10, 3.14, or "Hello world", are useful only for creating variables of the basic types – Int, Double, String, and so on. To use more complex types, you'll need to **instantiate** an object first.

When you write the following,

```
var item: ChecklistItem
```



it only tells Swift you want to store a `ChecklistItem` object into the `item` variable, but it does not create that `ChecklistItem` object itself. For that you need to write:

```
item = ChecklistItem()
```

This first reserves memory to hold the object's data, followed by a call to `init()` to properly set up the object for use. Reserving memory is also called **allocation**; filling up the object with its initial value(s) is **initialization**.

The whole process is known as **instantiating** the object — you're making an object **instance**. The instance is the block of memory that holds the values of the object's variables (that's why they are called "instance variables," get it?).

Of course, you can combine the above into a single line:

```
var item = ChecklistItem()
```

Here you left out the `: ChecklistItem` type annotation because Swift is smart enough to realize that the type of `item` should be `ChecklistItem`.

However, you can't leave out the `()` parentheses — this is how Swift knows that you want to make a new `ChecklistItem` instance.

Some objects allow you to pass **parameters** to their `init` method. For example:

```
var item = ChecklistItem(text: "Charge my iPhone", checked:  
false)
```

This calls the corresponding `init(text:checked:)` method to prepare the newly allocated `ChecklistItem` object for usage.

You've seen two types of variables: **local variables**, whose existence is limited to the method they are declared in, and **instance variables** (also known as "ivars," or properties) that belong to the object and therefore can be used from within any method in the object.

The lifetime of a variable is called its **scope**. The scope of a local variable is smaller than that of an instance variable. Once the method ends, any local variables are destroyed.

```
class MyObject {  
    var count = 0      // an instance variable  
  
    func myMethod() {  
        var temp: Int   // a local variable  
        temp = count    // OK to use the instance variable here  
    }  
  
    // the local variable "temp" doesn't exist outside the method  
}
```

If you have a local variable with the same name as an instance variable, then it is said to **shadow** (or **hide**) the instance variable. You should avoid these situations as they can lead to subtle bugs where you may not be using the variable that you think you are:

```
class MyObject {  
    var count = 7      // an instance variable  
  
    func myMethod() {  
        var count = 42  // local variable "hides" instance variable  
        print(count)   // prints 42  
    }  
}
```

Some developers place an underscore in front of their instance variable names to avoid this problem: `_count` instead of `count`. An alternative is to use the keyword `self` whenever you want to access an instance variable:

```
func myMethod() {  
    var count = 42  
    print(self.count)  // prints 7  
}
```

Constants

Variables are not the only code elements that can hold values. A variable is a container for a value that is allowed to *change* over the course of the app being run.

For example, in a note-taking app, the user can change the text of the note. So, you'd place that text into a `String` variable. Every time the user edits the text, the variable is updated.



Sometimes, you'll just want to store the result of a calculation or a method call into a temporary container, after which this value will never change. In that case, it is better to make this container a **constant** rather than a variable.

The following values cannot change once they've been set:

```
let pi = 3.141592
let difference = abs(targetValue - currentValue)
let message = "You scored \(points) points"
let image = UIImage(named: "SayCheese")
```

If a constant is local to a method, it's allowed to give the constant a new value the next time the method is called. The value from the previous method invocation is destroyed when that method ends, and the next time the app enters that method you're creating a new constant with a new value (but with the same name). Of course, for the duration of that method call, the constant's value must remain the same.

Tip: My suggestion is to use `let` for everything — that's the right solution 90% of the time. When you get it wrong, the Swift compiler will warn that you're trying to change a constant. Only then should you change it to a `var`. This ensures you're not making things variable that don't need to be.

Value types vs. reference types

When working with basic values such as integers and strings — which are **value types** — a constant created with `let` cannot be changed once it has been given a value:

```
let pi = 3.141592
pi = 3           // not allowed
```

However, with objects that are **reference types**, it is only the reference that is constant. The object itself can still be changed:

```
let item = ChecklistItem()
item.text = "Do the laundry"
item.checked = false
item.dueDate = yesterday
```

But this is not allowed:

```
let anotherItem = ChecklistItem()
item = anotherItem // cannot change the reference
```

So how do you know what is a reference type and what is a value type?

Objects defined as `class` are reference types, while objects defined as `struct` or `enum` are value types. In practice, this means most of the objects from the iOS SDK are reference types but things that are built into the Swift language, such as `Int`, `String`, and `Array`, are value types. (More about this important difference later.)

Collections

A variable stores only a single value. To keep track of multiple objects, you can use a **collection** object. Naturally, I'm talking about arrays (`Array`) and dictionaries (`Dictionary`), both of which you've seen previously.

An **array** stores a list of objects. The objects it contains are ordered sequentially and you retrieve them by index.

```
// An array of ChecklistItem objects:  
var items: Array<ChecklistItem>  
  
// Or, using shorthand notation:  
var items: [ChecklistItem]  
  
// Making an instance of the array:  
items = [ChecklistItem]()  
  
// Accessing an object from the array:  
let item = items[3]
```

You can write an array as `Array<Type>` or `[Type]`. The first one is the official version, the second is “syntactic sugar” that is a bit easier to read. (Unlike other languages, in Swift you don’t write `Type[]`. The type name goes inside the brackets.)

A **dictionary** stores key-value pairs. An object, usually a string, is the key that retrieves another object.

```
// A dictionary that stores (String, Int) pairs, for example a  
// list of people's names and their ages:  
var ages: Dictionary<String, Int>  
  
// Or, using shorthand notation:  
var ages: [String: Int]  
  
// Making an instance of the dictionary:  
ages = [String: Int]()  
  
// Accessing an object from the dictionary:  
var age = dict["Jony Ive"]
```

The notation for retrieving an object from a dictionary looks very similar to reading from an array — both use the [] brackets. For indexing an array, you always use a positive integer, but for a dictionary you typically use a string.

There are other sorts of collections as well, but array and dictionary are the most common ones.

Generics

Array and Dictionary are known as **generics**, meaning that they are independent of the type of thing you want to store inside these collections.

You can have an Array of Int objects, but also an Array of String objects — or an Array of any kind of object, really (even an array of other arrays).

That's why you have to specify the type of object to store inside the array, before you can use it. In other words, you cannot write this:

```
var items: Array // error: should be Array<TypeName>
var items: []     // error: should be [TypeName]
```

There should always be the name of a type inside the [] brackets or following the word Array in < > brackets. (If you're coming from Objective-C, be aware that the < > mean something completely different there.)

For Dictionary, you need to supply two type names: one for the type of the keys and one for the type of the values.

Swift requires that all variables and constants have a value. You can either specify a value when you declare the variable or constant, or by assigning a value inside an init method.

Optionals

Sometimes, it's useful to have a variable that can have no value, in which case you need to declare it as an **optional**:

```
var checklistToDelete: Checklist?
```

You cannot use this variable immediately; you must always first test whether it has a value or not. This is called **unwrapping** the optional:

```
if let checklist = checklistToDelete {
    // "checklist" now contains the real object
```

```
    } else {
        // the optional was nil
    }
```

The age variable from the dictionary example in the previous section is actually an optional, because there is no guarantee that the dictionary contains the key “Jony Ive.” Therefore, the type of age is Int? instead of just Int.

Before you can use a value from a dictionary, you need to unwrap it first using if let:

```
if let age = dict["Jony Ive"] {
    // use the value of age
}
```

If you are 100% sure that the dictionary contains a given key, you can also use **force unwrapping** to read the corresponding value:

```
var age = dict["Jony Ive"]!
```

With the ! you tell Swift, “This value will not be nil. I’ll stake my reputation on it!” Of course, if you’re wrong and the value is nil, the app will crash and your reputation is down the drain. Be careful with force unwrapping!

A slightly safer alternative to force unwrapping is **optional chaining**. For example, the following will crash the app if the navigationController property is nil:

```
navigationController!.delegate = self
```

But this won’t:

```
navigationController?.delegate = self
```

Anything after the ? will simply be ignored if navigationController does not have a value. It’s equivalent to writing:

```
if navigationController != nil {
    navigationController!.delegate = self
}
```

It is also possible to declare an optional using an exclamation point instead of a question mark. This makes it an **implicitly unwrapped optional**:

```
var dataModel: DataModel!
```

Such a value is potentially unsafe because you can use it as a regular variable without having to unwrap it first. If this variable has the value `nil` when you don't expect it — and don't they always — your app will crash.

Optionals exist to guard against such crashes, and using `!` undermines the safety of using optionals.

However, sometimes using implicitly unwrapped optionals is more convenient than using pure optionals. Use them when you cannot give the variable an initial value at the time of declaration, nor in `init()`.

But once you've given the variable a value, you really ought not to make it `nil` again. If the value can become `nil` again, it's better to use a true optional with a question mark.

Methods and functions

You've learned that objects, the basic building blocks of all apps, have both data and functionality. Instance variables and constants provide the data, **methods** provide the functionality.

When you call a method, the app jumps to that section of the code and executes all the statements in the method one-by-one. When the end of the method is reached, the app jumps back to where it left off:

```
let result = performUselessCalculation(314)
print(result)

...
func performUselessCalculation(_ a: Int) -> Int {
    var b = Int(arc4random_uniform(100))
    var c = a / 2
    return (a + b) * c
}
```

Methods often return a value to the caller, usually the result of a computation or looking up something in a collection. The data type of the result value is written after the `->` arrow. In the example above, it is `Int`. If there is no `->` arrow, the method does not return a value (also known as returning `Void`).

Methods are **functions** that belong to an object, but there are also standalone functions such as `print()`.



Functions serve the same purpose as methods – they bundle functionality into small re-usable units – but live outside of any objects. Such functions are also called *free* functions or *global* functions.

These are examples of methods:

```
// Method with no parameters, no return a value.  
override func viewDidLoad()  
  
// Method with one parameter, slider. No return a value.  
// The keyword @IBAction means that this method can be connected  
// to a control in Interface Builder.  
@IBAction func sliderMoved(_ slider: UISlider)  
  
// Method with no parameters, returns an Int value.  
func countUncheckedItems() -> Int  
  
// Method with two parameters, cell and item, no return value.  
// Note that the first parameter has an extra label, for,  
// and the second parameter has an extra label, with.  
func configureCheckmarkFor(for cell: UITableViewCell,  
                           with item: ChecklistItem)  
  
// Method with two parameters, tableView and section.  
// Returns an Int. The _ means the first parameter does not  
// have an external label.  
override func tableView(_ tableView: UITableView,  
                      numberOfRowsInSection section: Int) -> Int  
  
// Method with two parameters, tableView and indexPath.  
// The question mark means it returns an optional IndexPath  
// object (may also return nil).  
override func tableView(_ tableView: UITableView,  
                      willSelectRowAt indexPath: IndexPath) -> IndexPath?
```

To call a method on an object, you write `object.method(parameters)`. For example:

```
// Calling a method on the lists object:  
lists.append(checklist)  
  
// Calling a method with more than one parameter:  
tableView.insertRows(at: indexPaths, with: .fade)
```

You can think of calling a method as *sending a message* from one object to another: “Hey `lists`, I’m sending you the `append` message for this `checklist` object.”

The object whose method you’re calling is known as the *receiver* of the message.

It is very common to call a method from the same object. Here, `loadChecklists()` calls the `sortChecklists()` method. Both are members of the `DataModel` object.



```
class DataModel {
    func loadChecklists() {
        sortChecklists() // this method also lives in DataModel
    }

    func sortChecklists() {
    }
}
```

Sometimes, this is written as:

```
func loadChecklists() {
    self.sortChecklists()
}
```

The `self` keyword makes it clear that the `DataModel` object itself is the receiver of this message.

Note: In this book, the `self` keyword is left out for method calls, because it's not necessary to have it. Objective-C developers are very attached to `self`, so you'll probably see it used a lot in Swift too. It is a topic of heated debate in developer circles, but except for a few specific scenarios, the compiler doesn't really care whether you use `self` or not.

Inside a method, you can also use `self` to get a reference to the object itself:

```
@IBAction func cancel() {
    delegate?.itemDetailViewControllerDidCancel(self)
}
```

Here, `cancel()` sends a reference to the object (i.e. `self`) along to the delegate, so the delegate knows who sent this `itemDetailViewControllerDidCancel()` message.

Also note that the use of **optional chaining** here. The `delegate` property is an optional, so it can be `nil`. Using the question mark before the method call will ensure nothing bad happens if `delegate` is not set.

Parameters

Often methods have one or more **parameters**, so they can work with multiple data items. A method that is limited to a fixed set of data is not very useful or reusable. Consider `sumValuesFromArray()`, a method that has no parameters:

```
class MyObject {
    var numbers = [Int]()

    func sumValuesFromArray() -> Int {
        var total = 0
        for number in numbers {
            total += number
        }
        return total
    }
}
```

Here, `numbers` is an instance variable. The `sumValuesFromArray()` method is tied closely to that instance variable, and is useless without it.

Suppose you add a second array to the app that you also want to apply this calculation to. One approach is to copy-paste the above method and change the name of the variable to that of the new array. That certainly works, but it's not smart programming!

It is better to give the method a parameter that allows you to pass in the array object that you wish to examine. Then, the method becomes independent from any instance variables:

```
func sumValues(from array: [Int]) -> Int {
    var total = 0
    for number in array {
        total += number
    }
    return total
}
```

Now you can call this method with any `[Int]` (or `Array<Int>`) object as its parameter.

This doesn't mean methods should never use instance variables, but if you can make a method more general by giving it a parameter, then that is usually a good idea.

Often methods use two names for their parameters, the **external label** and the **internal label**. For example:

```
func downloadImage(for searchResult: SearchResult,  
                   withTimeout timeout: TimeInterval,  
                   andPlaceOn button: UIButton) {  
    . . .
```

This method has three parameters: `searchResult`, `timeout`, and `button`. Those are the internal parameter names you'd use in the code inside the method.

The external labels become part of the method name. The full name for the method is `downloadImage(for:withTimeout:andPlaceOn:)` — method names in Swift are often quite long!

To call this method, you'd use the external labels:

```
downloadImage(for: result, withTimeout: 10,  
              andPlaceOn: imageButton)
```

Sometimes you'll see a method whose first parameter does not have an external label, but has an `_` underscore instead:

```
override func tableView(_ tableView: UITableView,  
                      numberOfRowsInSection section: Int) -> Int
```

This is often the case with delegate methods. It's a holdover from the Objective-C days, where the label for the first parameter was embedded in the first part of the method name. For example, in Objective-C the `downloadImage()` method example above would be named `downloadImageForSearchResult()`. These kinds of names should become less and less common in the near future.

Swift is pretty flexible with how it lets you name your methods, but it's smart to stick to the established conventions.

Inside a method you can do the following things:

- Create local variables and constants.
- Do basic arithmetic with mathematical operators such as `+`, `-`, `*`, `/`, and `%`.
- Put new values into variables (both local and instance variables).
- Call other methods.
- Make decisions with `if` or `switch` statements.

- Perform repetitions with the `for` or `while` statements.
- Return a value to the caller.

Let's look at the `if` and `for` statements in more detail.

Making decisions

The `if` statement looks like this:

```
if count == 0 {  
    text = "No Items"  
} else if count == 1 {  
    text = "1 Item"  
} else {  
    text = "\(count) Items"  
}
```

The expression after `if` is called the **condition**. If a condition is true then the statements in the following `{ }` block are executed. The `else` section gets performed if none of the conditions are true.

Comparison Operators

You use **comparison operators** to perform comparisons between two values:

- `==` equal to
- `!=` not equal
- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal

```
let a = "Hello, world"  
let b = "Hello," + " world"  
print(a == b)           // prints true
```

When you use the `==` operator, the contents of the objects are compared. The above code only returns true if `a` and `b` have the same value:

This is different from Objective-C, where `==` is only `true` if the two objects are the exact same instance in memory. However, in Swift `==` compares the values of the objects, not whether they actually occupy the same spot in memory. (If you need to do that use `===`, the identity operator.)

Logical Operators

You can use **logical** operators to combine two expressions:

`a && b` is true if both `a` *and* `b` are true

`a || b` is true when either `a` *or* `b` is true (or both)

There is also the logical **not** operator, `!`, that turns `true` into `false`, and `false` into `true`. (Don't confuse this with the `!` that is used with optionals.)

You can group expressions with `()` parentheses:

```
if ((this && that) || (such && so)) && !other {  
    // statements  
}
```

This reads as:

```
if ((this and that) or (such and so)) and not other {  
    // statements  
}
```

Or if you want to see clearly in which order these operations are performed:

```
if (  
    (this and that)  
    or  
    (such and so)  
)  
and  
(not other)
```

Of course, the more complicated you make it, the harder it is to remember exactly what you're doing!

switch statement

Swift has another very powerful construct in the language for making decisions, the `switch` statement:

```
switch condition {  
    case value1:  
        // statements  
  
    case value2:  
        // statements  
  
    case value3:  
        // statements  
  
    default:  
        // statements  
}
```

It works the same way as an `if` statement with a bunch of `else if`s. The following is equivalent:

```
if condition == value1 {  
    // statements  
} else if condition == value2 {  
    // statements  
} else if condition == value3 {  
    // statements  
} else {  
    // statements  
}
```

In such a situation, the `switch` statement would be more convenient to use. Swift's version of `switch` is much more powerful than the one in Objective-C. For example, you can match on ranges and other patterns:

```
switch difference {  
    case 0:  
        title = "Perfect!"  
    case 1..  
        title = "You almost had it!"  
    case 5..  
        title = "Pretty good!"  
    default:  
        title = "Not even close..."  
}
```

The `..<<` is the **half-open range** operator. It creates a range between the two numbers, but the top number is exclusive. So the half-open range `1..<<5` is the same as the **closed range** `1...4`.

You'll see the `switch` statement in action a little later on.

return statement

Note that `if` and `return` can be used to return early from a method:

```
func divide(_ a: Int, by b: Int) -> Int {
    if b == 0 {
        print("You really shouldn't divide by zero")
        return 0
    }
    return a / b
}
```

This can even be done for methods that don't return a value:

```
func performDifficultCalculation(list: [Double]) {
    if list.count < 2 {
        print("Too few items in list")
        return
    }
    // perform the very difficult calculation here
}
```

In this case, `return` simply means: "We're done with the method." Any statements following the `return` are skipped and execution immediately returns to the caller.

You could also have written it like this:

```
func performDifficultCalculation(list: [Double]) {
    if list.count < 2 {
        print("Too few items in list")
    } else {
        // perform the very difficult calculation here
    }
}
```

Which approach you use is up to you. The advantage of an early `return` is that it avoids multiple nested blocks of code with multiple levels of indentation — the code just looks cleaner.

For example, sometimes you see code like this:

```
func someMethod() {  
    if condition1 {  
        if condition2 {  
            if condition3 {  
                // statements  
            } else {  
                // statements  
            }  
        } else {  
            // statements  
        }  
    } else {  
        // statements  
    }  
}
```

This can become very hard to read. You could restructure that kind of code as follows:

```
func someMethod() {  
    if !condition1 {  
        // statements  
        return  
    }  
  
    if !condition2 {  
        // statements  
        return  
    }  
  
    if !condition3 {  
        // statements  
        return  
    }  
  
    // statements  
}
```

Both do exactly the same thing, but the second one is easier to understand. (Note that the conditions now use the `!` operator to invert their meaning.)

Swift even has a dedicated feature, `guard`, to help write this kind of code. It looks like this:

```
func someMethod() {  
    guard condition1 else {  
        // statements  
        return  
    }
```

```
    }
    guard condition2 else {
        // statements
        return
    }
    . . .
```

As you become more experienced, you'll start to develop your own taste for what looks good and what is readable code.

Loops

You've seen the `for...in` statement for looping through an array:

```
for item in items {
    if !item.checked {
        count += 1
    }
}
```

Which can also be written as:

```
for item in items where !item.checked {
    count += 1
}
```

This performs the statements inside the `for...in` block once for each object from the `items` array matching the condition provided by the `where` clause.

Note that the scope of the variable `item` is limited to just this `for` statement. You can't use it outside this statement, so its lifetime is even shorter than a local variable.

Looping through number ranges

Some languages, including Swift 2, have a `for` statement that looks like this:

```
for var i = 0; i < 5; ++i {
    print(i)
}
```

When you run this code, it should print:

```
0  
1  
2  
3  
4
```

However, as of Swift 3.0 this kind of `for` loop was removed from the language. Instead, you can loop over a range. This has the same output as above:

```
for i in 0...4 {    // or 0..  
    print(i)  
}
```

By the way, you can also write this loop as:

```
for i in stride(from: 0, to: 5, by: 1) {  
    print(i)  
}
```

The `stride()` function creates a special object that represents the range 0 to 5 in increments of 1. If you wanted to show just the even numbers, you could change the `by` parameter to 2. You can even use `stride()` to count backwards if you pass the `by` parameter a negative number.

while statement

The `for` statement is not the only way to perform loops. Another very useful looping construct is the `while` statement:

```
while something is true {  
    // statements  
}
```

The `while` loop keeps repeating the statements until its condition becomes false. You can also write it as follows:

```
repeat {  
    // statements  
} while something is true
```

In the latter case, the condition is evaluated after the statements have been executed at least once.



You can rewrite the loop that counts the ChecklistItems as follows using a `while` statement:

```
var count = 0
var i = 0
while i < items.count {
    let item = items[i]
    if !item.checked {
        count += 1
    }
    i += 1
}
```

Most of these looping constructs are really the same, they just look different. Each of them lets you repeat a bunch of statements until some ending condition is met.

Still, using a `while` is slightly more cumbersome than “`for item in items`,” which is why you’ll see `for...in` used most of the time.

There really is no significant difference between using a `for`, `while`, or `repeat...while` loop, except that one may be easier to read than the others, depending on what you’re trying to do.

Note: `items.count` and `count` in this example are two different things with the same name. The first `count` is a property on the `items` array that returns the number of elements in that array; the second `count` is a local variable that contains the number of unchecked to-do items counted so far.

Just like you can prematurely exit from a method using the `return` statement, you can exit a loop at any time using the `break` statement:

```
var found = false
for item in array {
    if item == searchText {
        found = true
        break
    }
}
```

This example loops through the array until it finds an `item` that is equal to the value of `searchText` (presumably both are strings). Then it sets the variable `found` to `true` and jumps out of the loop using `break`. You’ve found what you were looking for, so it makes no sense to look at the other objects in that array — for all you know there could be hundreds of items.

There is also a `continue` statement that is somewhat the opposite of `break`. It doesn't exit the loop but immediately skips to the next iteration. You use `continue` to say, "I'm done with the current item, let's look at the next one."

Loops can often be replaced by *functional programming* constructs such as `map`, `filter`, or `reduce`. These are known as *higher order functions* and they operate on a collection, performing some code for each element, and return a new collection (or single value, in the case of `reduce`) with the results.

For example, using `filter` on an array will return items that satisfy a certain condition. To get a list of all the unchecked `ChecklistItem` objects, you'd write:

```
var uncheckedItems = items.filter { item in !item.checked }
```

That's a lot simpler than writing a loop. Functional programming is an advanced topic so we won't spend too much time on it here.

Objects

Objects are what it's all about. They combine data with functionality into coherent, reusable units — that is, if you write them properly!

The data is made up of the object's instance variables and constants. We often refer to these as the object's **properties**. The functionality is provided by the object's methods.

In your Swift programs you will use existing objects, such as `String`, `Array`, `Date`, `UITableView`, and you'll also make your own.

To define a new object, you need a bit of code that contains a `class` section:

```
class MyObject {
    var text: String
    var count = 0
    let maximum = 100

    init() {
        text = "Hello world"
    }

    func doSomething() {
        // statements
    }
}
```

Inside the brackets for the class, you add properties (the instance variables and constants) and methods.

Properties

There are two types of properties:

- **Stored properties** are the usual instance variables and constants.
- **Computed properties** don't store a value, but perform logic when you read from, or write to, their values.

This is an example of a computed property:

```
var indexOfSelectedChecklist: Int {  
    get {  
        return UserDefaults.standard.integer(  
            forKey: "ChecklistIndex")  
    }  
    set {  
        UserDefaults.standard.set(newValue,  
            forKey: "ChecklistIndex")  
    }  
}
```

The `indexOfSelectedChecklist` property does not store a value like a normal variable would. Instead, every time someone uses this property, it performs the code from the `get` or `set` block.

The alternative would be to write separate `setIndex0fSelectedChecklist()` and `getIndex0fSelectedChecklist()` methods, but that doesn't read as nicely.

If a property name is preceded by the keyword `@IBOutlet`, that means that the property can refer to a user interface element in Interface Builder, such as a label or button. Such properties are usually declared `weak` and `optional`.

Similarly, the keyword `@IBAction` is used for methods that will be performed when the user interacts with the app.

Methods

There are three kinds of methods:

- Instance methods
- Class methods
- Init methods

As mentioned previously, a method is a function that belongs to an object. To call such a method you first need to have an instance of the object:

```
let myInstance = MyObject()    // create the object instance
myInstance.doSomething()      // call the method
```

You can also have **class methods**, which can be used without an object instance. In fact, they are often used as “factory” methods, to create new object instances:

```
class MyObject {
    .

    class func makeObject(text: String) -> MyObject {
        let m = MyObject()
        m.text = text
        return m
    }

    let myInstance = MyObject.makeObject(text: "Hello world")
```

Init methods, or **initializers**, are used during the creation of new object instances. Instead of the above factory method, you might as well use a custom `init` method:

```
class MyObject {
    .

    init(text: String) {
        self.text = text
    }

    let myInstance = MyObject(text: "Hello world")
```

The main purpose of an `init` method is to set up (or, initialize) the object’s properties. Any instance variables or constants that do not have a value yet must be given one in the `init` method.



Swift does not allow variables or constants to have no value (except for optionals), and `init` is your last chance to make this happen.

Objects can have more than one `init` method; which one you use depends on the circumstances.

A `UITableViewController`, for example, can be initialized either with `init?(coder:)` when automatically loaded from a storyboard, with `init(nibName:bundle:)` when manually loaded from a nib file, or with `init(style:)` when constructed without a storyboard or nib — sometimes you use one, sometimes the other. You can also provide a `deinit` method that gets called just before the object is destroyed.

By the way, `class` isn't the only way to define an object in Swift. It also supports other types of objects such as `structs` and `enums`. You'll learn more about these later in the book.

Protocols

Besides objects, you can also define **protocols**. A protocol is simply a list of method names (and possibly, properties):

```
protocol MyProtocol {
    func someMethod(value: Int)
    func anotherMethod() -> String
}
```

A protocol is like a job ad. It lists all the things that a candidate for a certain position in your company should be able to do.

But the ad itself doesn't do the job — it's just words printed in the careers section of the newspaper. So, you need to hire an actual employee who can get the job done. That would be an object.

Objects need to indicate that they conform to a protocol:

```
class MyObject: MyProtocol {
    .
    .
}
```

This object now has to provide an implementation for the methods listed in the protocol. (If not, it's fired!)



From then on, you can refer to this object as a `MyObject` (because that is its class name) but also as a `MyProtocol` object:

```
var m1: MyObject = MyObject()  
var m2: MyProtocol = MyObject()
```

To any part of the code using the `m2` variable, it doesn't matter that the object is really a `MyObject` under the hood. The type of `m2` is `MyProtocol`, not `MyObject`.

All your code sees is that `m2` is *some* object conforming to `MyProtocol`, but it's not important what sort of object that is.

In other words, you don't really care that your employee may also have another job on the side, as long as it doesn't interfere with the duties you've hired him, or her, for.

Protocols are often used to define **delegates**, but they come in handy for other uses as well, as you'll find out later on.

This concludes the quick recap of what you've seen so far of the Swift language. After all that theory, it's time to write some code!

Chapter 27: Get Location Data

Eli Ganim

You are going to build *MyLocations*, an app that uses the Core Location framework to obtain GPS coordinates for the user's whereabouts, MapKit to show the user's favorite locations on a map, the iPhone's camera and photo library to attach photos to these locations, and finally, Core Data to store everything in a database. Phew, that's a lot of stuff!

The finished app looks like this:



The MyLocations app

MyLocations lets you keep a list of spots that you find interesting. Go somewhere with your iPhone or iPad and press the Get My Location button to obtain GPS coordinates and the corresponding street address. Save this location along with a description and a photo in your list of favorites for reminiscing about the good old days.

Think of this app as a “location album” instead of a photo album.

To make the workload easier to handle, you’ll split the project up into smaller chunks:

1. You will first figure out how to obtain GPS coordinates from the Core Location framework and how to convert these coordinates into an address, a process known as **reverse geocoding**. Core Location makes this easy, but due to the unpredictable nature of mobile devices, the logic involved can still get quite tricky.
2. Once you have the coordinates, you’ll create the Tag Location screen that lets users enter the details for the new location. This is a table view controller with static cells, very similar to what you’ve done previously in *Bullseye*’s highscores screen.
3. You’ll store the location data into a Core Data store. For the last app you saved app data into a .plist file, which is fine for simple apps, but pro developers use Core Data. It’s not as scary as it sounds!
4. Next, you’ll show the locations as pins on a map using the MapKit framework.
5. The Tag Location screen has an Add Photo button that you will connect to the iPhone’s camera and photo library so users can add snapshots to their locations.
6. Finally, you’ll make the app look good using custom graphics. You will also add sound effects and some animations to the mix.

Of course, you are not going to do all of that at once. In this chapter, you will do the following:

- **Get GPS Coordinates:** Create a tab bar based app and set up the UI for the first tab.
- **CoreLocation:** Use the CoreLocation framework to get the user’s current location.
- **Display coordinates:** Display location information on screen.

When you're done with this chapter, the app will look like this:



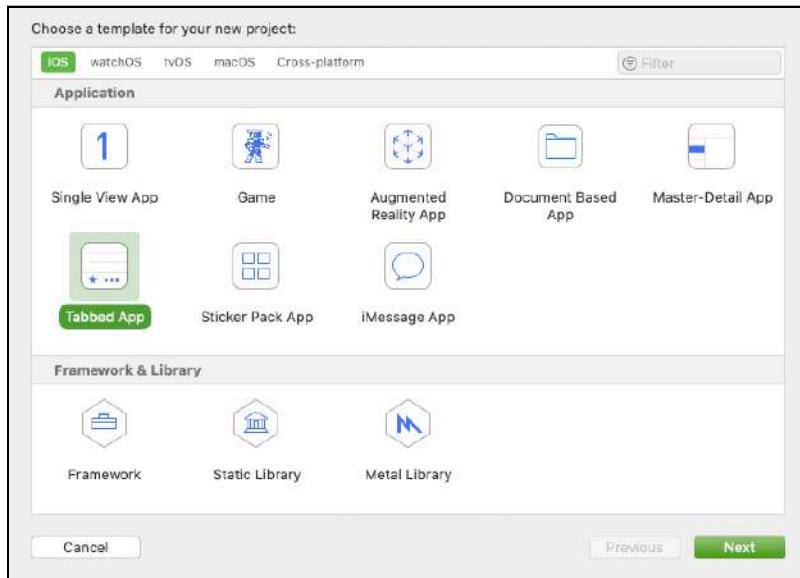
The first screen of the app

Get GPS coordinates

First, you'll create the *MyLocations* project in Xcode and then use the Core Location framework to find the latitude and longitude of the user's location.

Creating the project

- Fire up Xcode and make a new project. Choose the **Tabbed App** template.

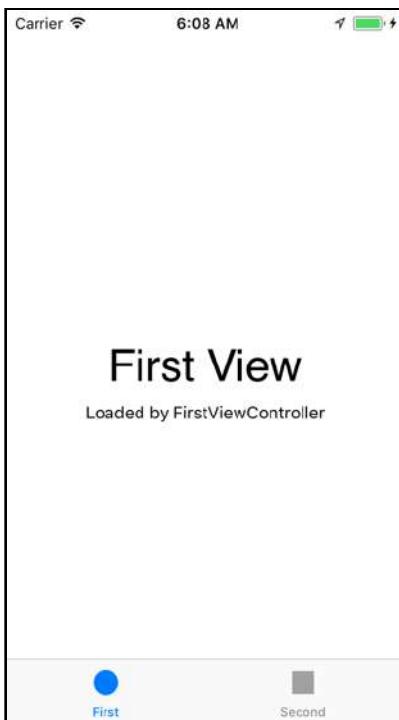


Choosing the Tabbed Application template

- Fill in the options as follows:

- Product Name: **MyLocations**
 - Organization Name: Your name or the name of your company
 - Organization Identifier: Your own identifier in reverse domain notation
 - Language: **Swift**
 - Include Unit Tests and Include UI Tests: unchecked
 - Use SwiftUI: unchecked
- Save the project.

If you run the app, it looks like this:



The app from the Tabbed Application template

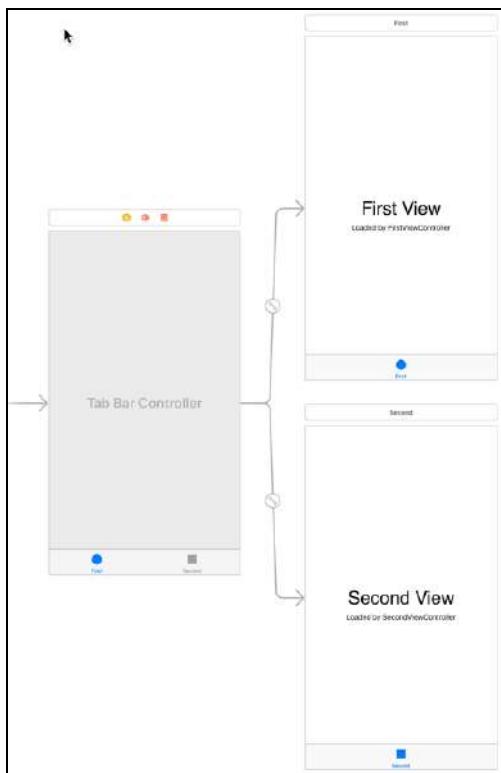
The app has a tab bar along the bottom with two tabs: First and Second.

Even though it doesn't do much yet, the app already employs three view controllers:

1. The *root controller* is a UITabBarController that contains the tab bar and performs the switching between the different screens.
2. A view controller for the First tab.
3. A view controller for the Second tab.

The two tabs each have their own view controller. By default, the Xcode template names them `FirstViewController` and `SecondViewController`.

At this point, the storyboard looks like this:



The storyboard from the Tabbed Application template

It's zoomed out to fit the whole thing on the screen. Storyboards are great, but they sure take up a lot of space!

As before, you'll be editing the storyboard using the iPhone 8 dimensions. Later, if necessary, you'll make some adjustments to get the app to work on other screen sizes as well.

- In the **View as:** pane at the bottom, choose **iPhone 8**.

The first tab

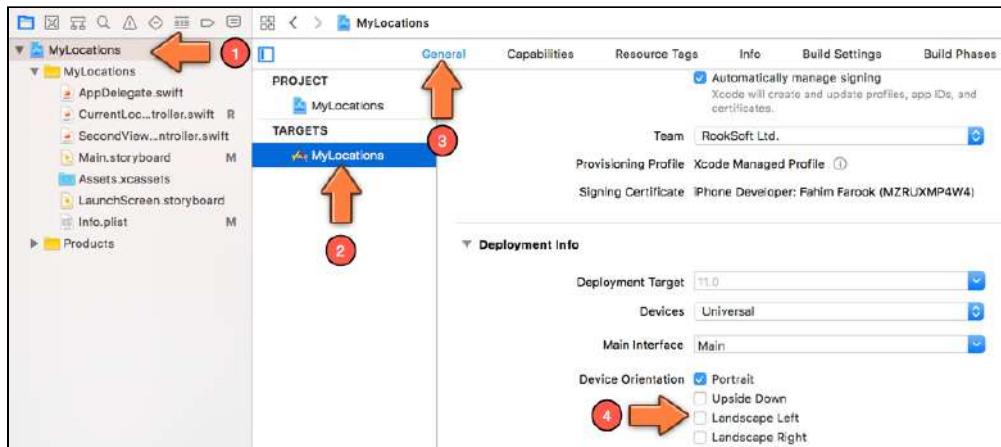
In this chapter, you'll be working with the first tab only. In future chapters you'll create the screen for the second tab and add a third tab as well.

Let's give `FirstViewController` a better name.



Remember the refactoring trick you learned previously? That's what you'll use here since that renames both the file and any references to it anywhere in the project.

- Open **FirstViewController.swift**, hover your mouse cursor over the word **FirstViewController** in the class line, right-click (or Control-click) and select **Refactor > Rename...** from the context menu.
- Change the name to **CurrentLocationViewController**. This changes the file name, the class name and the reference to the class in the storyboard, all at once! Nifty, eh?
- Go to the **Project Settings** screen and de-select the Landscape Left and Landscape Right settings under **Deployment Info – Device Orientation**. Now the app is portrait-only. (You can enable **Upside Down** at the same time if you like, since this would enable both portrait modes on iPad.)



The app only works in portrait

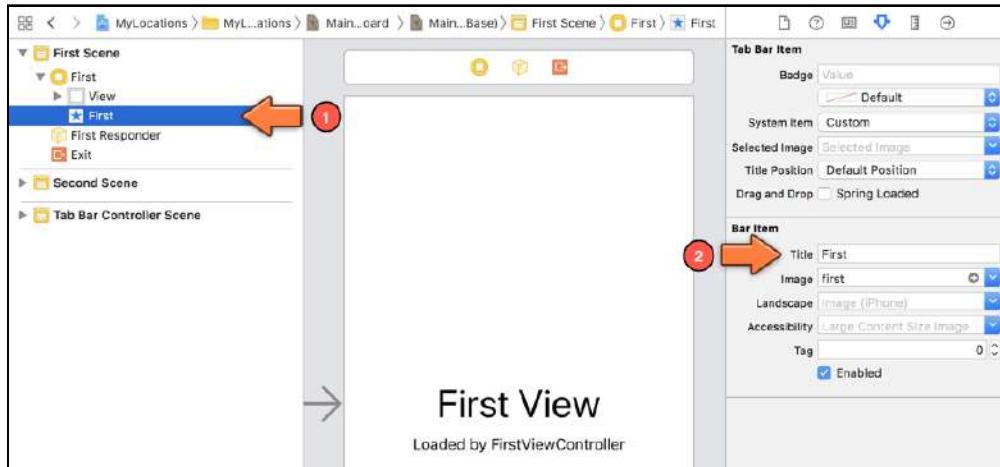
- Run the app again just to make sure everything still works.

Whenever you change how things are hooked up in the storyboard, it's useful to run the app and verify that the change was successful — it's way too easy to forget a step and you want to catch such mistakes right away.

And if you are wondering where you changed things in the storyboard, remember how you renamed the **FirstViewController**? That change modified the storyboard, too.

A view controller that sits inside a navigation controller has a **Navigation Item** object that allows it to configure the navigation bar. Tab bars work the same way. Each view controller that represents a tab has a **Tab Bar Item** object.

- Open the storyboard, select the **Tab Bar Item** object from the **First Scene** (this is the Current Location View Controller) and go to the **Attributes inspector**. Change the **Title** to **Tag**.



Changing the title of the Tab Bar Item

Later on, you'll also set a new image for the Tab Bar Item too; it currently uses the default image from the template.

First tab UI

You will now design the screen for this first tab. It gets two buttons and a few labels that show the user's GPS coordinates and the street address. To save you some time, you'll add all the outlets in one go.

- Add the following to the class in **CurrentLocationViewController.swift**, just after the class definition and before `viewDidLoad()`:

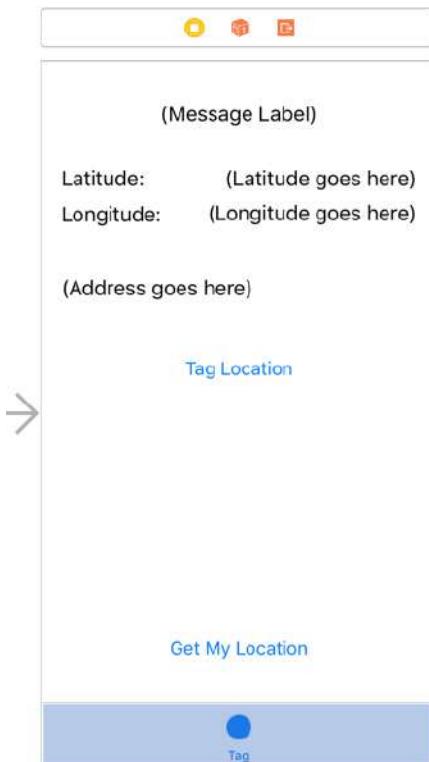
```
@IBOutlet weak var messageLabel: UILabel!
@IBOutlet weak var latitudeLabel: UILabel!
@IBOutlet weak var longitudeLabel: UILabel!
@IBOutlet weak var addressLabel: UILabel!
@IBOutlet weak var tagButton: UIButton!
@IBOutlet weak var getButton: UIButton!
```

- Still in **CurrentLocationViewController.swift**, add this just before the last curly brackets:

```
// MARK:- Actions
@IBAction func getLocation() {
```

```
// do nothing yet  
}
```

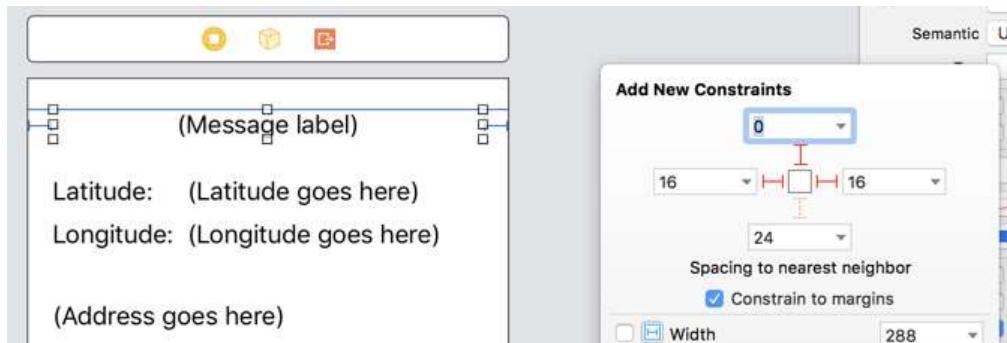
Now open the storyboard, remove the existing labels and design the UI to look something like this — always make use of the positioning guides that Interface Builder provides to place controls since this gives you nice, even spacing:



The design of the Current Location screen

- The **(Message Label)** at the top should span the whole width of the screen. You'll use this label for status messages while the app is obtaining the GPS coordinates. Set the **Alignment** attribute to centered and connect the label to the `messageLabel` outlet.
- Once you've positioned the **(Message Label)**, set its Auto Layout constraints for left, top and right so that it aligns with the Safe Area. In case you're wondering, you don't have to explicitly select the Safe Area.

If you set up the constraints as below, most of the time the constraints should be set correctly for you.



The (Message Label) constraints

- Make the **(Latitude goes here)** and **(Longitude goes here)** labels right-aligned and connect them to the `latitudeLabel` and `longitudeLabel` outlets respectively.
- Set up **left**, **top**, **right** and **bottom** Auto Layout constraints for the **Latitude:** label. You can use your judgement with regards to the top spacing — here 24 points is the value used — or you can use the suggested spacing of 8 points. It's totally up to you, but feels like there should be a bit more spacing between the message and the latitude, longitude grouping.
- Then, set up **left**, **right** and **bottom** constraints for the **Longitude:** label — you don't need a top constraint since the bottom constraint of the **Latitude:** label acts as the top constraint for this one.

Again, you can use your judgement with regards to the bottom spacing since that determines how far away the **(Address goes here)** label is from the **Longitude:** label. It probably should have the same amount of spacing as there was at the top to the **(Message Label)** and so 24 points make sense.

- Add **top**, **right** and **bottom** constraints for **(Latitude goes here)** and **right** and **bottom** constraints for **(Longitude goes here)**.

Do note that as you add constraints, the positions of some of the labels might shift. So you might need to adjust positioning again — for example, position the **(Longitude goes here)** label so that it stretches to the right edge of the screen — to set things up as they originally were.

- You will get some Auto Layout constraint issues at this point. This is due to none of the labels in the latitude and longitude grouping having specific widths or heights. It's hard for Xcode to determine what the actual sizes should be.

We know that the longer of the two left labels is **Longitude**: So let's try setting both labels on the left to be the same size as the **Longitude**: label — Control-drag from the **Longitude**: label to the **Latitude**: label and select **Equal Widths** from the pop-up.

Hmm... that made things worse! Why?

Because you had an existing trailing space from the **Latitude**: label to the **(Latitude goes here)** label and that spacing is now incorrect. Select the **(Latitude goes here)** label, switch to the **Size inspector** and remove the leading constraint to the **Latitude**: label.

- The **(Latitude goes here)** label will resize to fit its contents again. Add a leading constraint between it and the **Latitude**: label, but make the spacing greater than or equal to 8 points to match what's there for the longitude label set.

Why greater than or equal to when all the other constraints are set to equal to? Because if you set it to equal to, you'll get another set of red constraints.

- The **(Address goes here)** label spans the whole width of the screen and should be 50 points high so it can fit two lines of text. Set its **Lines** attribute to **0** (that means it can display a variable number of lines). Connect this label to the `addressLabel` outlet.
- Set **left**, **right** and **bottom** constraints on the **(Address goes here)** label. Make the bottom constraint be 24 points to match the top spacing previously set, or, whatever value you set/like. It's your choice.
- The **Tag Location** button doesn't do anything yet, but should be connected to the `tagButton` outlet.
- Set **left** and **right** constraints of 16 points on the **Tag Location** button so that it stretches from side to side.
- Connect the **Get My Location** button to the `getButton` outlet and its Touch Up Inside event to the `getLocation` action.
- Set **left**, **right** and **bottom** constraints on the **Get My Location** button so that it stretches from side to side and is at least 16 points from the bottom of the screen — you can use your judgement as to the actual positioning you think is good.
- Run the app to see the new design in action.

If you think the positioning of some element is off, feel free to adjust the Auto Layout constraints till the layout looks right. There is no right or wrong here, it's all a matter of how it looks to you.

So far, nothing special. With the exception of the tab bar, this is stuff you've seen and done before. Time to add something new: Let's play with Core Location!

Core Location

Most iOS devices have a way to let you know exactly where you are on the globe, either through communication with GPS satellites, or Wi-Fi and cell tower triangulation. The Core Location framework puts that power in your own hands.

An app can ask Core Location for the user's current latitude and longitude. For devices with a compass, it can also give the heading — you won't be using that for this app. Core Location can also provide continuous location updates while you're on the move.

Get your current location

Getting a location from Core Location is pretty easy, but there are some pitfalls that you need to avoid. Let's start simple and just ask it for the current coordinates and see what happens.

- At the top of **CurrentLocationViewController.swift**, add an import statement:

```
import CoreLocation
```

That is all you have to do to add the Core Location framework to your project.

Core Location, like many other parts of the iOS SDK, works via a delegate, so you should make the view controller conform to the `CLLocationManagerDelegate` protocol.

- Add `CLLocationManagerDelegate` to the view controller's class line:

```
class CurrentLocationViewController: UIViewController,  
    CLLocationManagerDelegate {
```

- Also add a new property:

```
let locationManager = CLLocationManager()
```



The `CLLocationManager` is the object that will give you GPS coordinates. You’re putting the reference to this object in a constant — using `let`, not a variable (`var`). Once you have created the location manager object, the value of `locationManager` will never have to change.

The new `CLLocationManager` object doesn’t give you GPS coordinates right away. To begin receiving coordinates, you have to call its `startUpdatingLocation()` method first.

Unless you’re doing turn-by-turn navigation, you don’t want your app to continuously receive GPS coordinates. That requires a lot of power and will quickly drain the battery. For this app, you only turn on the location manager when you want a location fix and turn it off again when you’ve received a usable location.

You’ll implement that logic in a minute — it’s more complex than you’d think. For now, you’re only interested in receiving something from Core Location, just so you know that it works.

- Change the `getLocation()` action method to the following:

```
@IBAction func getLocation() {
    locationManager.delegate = self
    locationManager.desiredAccuracy =
        kCLLocationAccuracyNearestTenMeters
    locationManager.startUpdatingLocation()
}
```

This method is hooked up to the **Get My Location** button. It tells the location manager that the view controller is its delegate and that you want to receive locations with an accuracy of up to ten meters. Then you start the location manager. From that moment on, the `CLLocationManager` object will send location updates to its delegate, i.e., the view controller.

- Speaking of the delegate, add the following code:

```
// MARK: - CLLocationManagerDelegate
func locationManager(_ manager: CLLocationManager,
    didFailWithError error: Error) {
    print("didFailWithError \(error.localizedDescription)")
}

func locationManager(_ manager: CLLocationManager,
    didUpdateLocations locations: [CLLocation]) {
    let newLocation = locations.last!
    print("didUpdateLocations \(newLocation)")
}
```

These are the delegate methods for the location manager. For the time being, you'll simply output a `print()` message to the Console. Also, do note the `error.localizedDescription` bit which, instead of simply printing out the contents of the `error` variable, outputs a human readable version of the error (if possible) based on the device's current locale (or language setting).

- Run the app in the simulator and press the **Get My Location** button.

Hmm... nothing seems to be happening. That's because you need to ask for permission before accessing location information.

Ask for permission

- Add the following lines to the top of `getLocation()`:

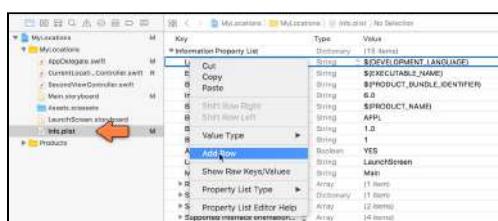
```
let authStatus = CLLocationManager.authorizationStatus()
if authStatus == .notDetermined {
    locationManager.requestWhenInUseAuthorization()
    return
}
```

This checks the current authorization status. If it is `.notDetermined` — meaning that this app has not asked for permission yet — then the app will request “When In Use” authorization. That allows the app to get location updates while it is open and the user is interacting with it.

There is also “Always” authorization, which permits the app to check the user’s location even when it is not active. That’s useful for a navigation app, for example. For most apps, including *MyLocations*, when-in-use is what you want to ask for.

Just adding these lines of code is not enough. You also have to add a special key to the app’s `Info.plist`.

- Open `Info.plist` file. Right-click somewhere inside `Info.plist` and choose **Add Row** from the menu.



Adding a new row to Info.plist

- For the key, type **NSLocationWhenInUseUsageDescription** (or choose **Privacy – Location When In Use Usage Description** from the list).
- Type the following text in the Value column:

This app lets you keep track of interesting places. It needs access to the GPS coordinates for your location.

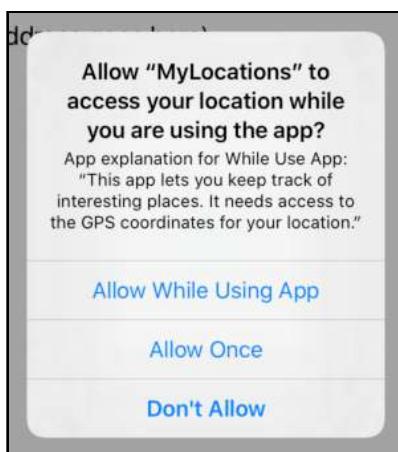
This description tells the user what the app wants to use the location data for.

Key	Type	Value
# Information Property List	Dictionary	{(8 items)}
LocationAuthorization	String	NSLOCATIONWHENINUSE
NSLocationWhenInUseUsageDescription	String	"This app lets you keep track of interesting places. It needs access to the GPS coordinates for your location."
Executable File	String	SHOOTOUT-BUNDLE-IDENTIFIER
Bundle Identifier	String	com.raywenderlich.shootout
CFBundleVersion	String	1.0
CFBundleName	String	Shootout

Adding the new item to Info.plist

- Run the app again and press the **Get My Location** button.

Core Location will pop up the following alert, asking the user for permission:



Users have to allow your app to use their location

If a user denies the request with the **Don't Allow** button, then Core Location will never give your app location coordinates. If the user chooses "Allow Once", then location services will only be available during this session and the app will request access again next time.

- Press the **Don't Allow** button. Now press Get My Location again.

Xcode's debug area should now show the following message (or something similar):

```
didFailWithError The operation couldn't be completed.  
(kCLErrorDomain error 1.)
```

This comes from the `locationManager(_:didFailWithError:)` delegate method. It's telling you that the location manager wasn't able to obtain a location. The reason why is described by an `Error` object, which is the standard object that the iOS SDK uses to convey error information. You'll see it in many other places in the SDK since there are plenty of places where things can go wrong!

This `Error` object has a *domain* and a *code*. The domain in this case is `KCLErrorDomain` meaning the error came from Core Location (CL). The code is 1, also identified by the symbolic name `CLError.denied`, which means the user did not allow the app to obtain location information.

Note: The `k` prefix is often used by the iOS frameworks to signify that a name represents a constant value — maybe whoever came up with this prefix thought it was spelled “konstant.” This is an old convention and you won’t see it used much in new frameworks or in Swift code, but it still pops up here and there.

- Stop the app from within Xcode and run it again.

When you press the Get My Location button, the app does not ask for permission anymore but immediately gives you the same error message.

Let’s make this a bit more user-friendly, because a normal user would never see that `print()` output.

Handle permission errors

- Add the following method to `CurrentLocationViewController.swift`:

```
// MARK:- Helper Methods
func showLocationServicesDeniedAlert() {
    let alert = UIAlertController(
        title: "Location Services Disabled",
        message: "Please enable location services for this app in
Settings.",
        preferredStyle: .alert)

    let okAction = UIAlertAction(title: "OK", style: .default,
                                handler: nil)
    alert.addAction(okAction)

    present(alert, animated: true, completion: nil)
}
```

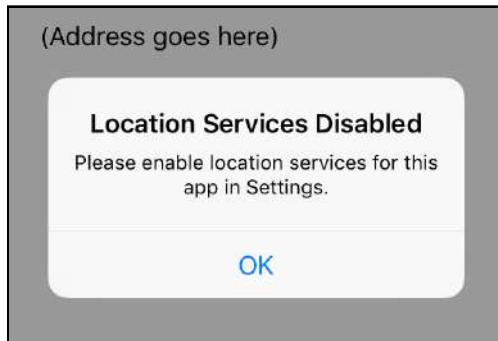
This pops up an alert with a helpful hint. This app is pretty useless without access to the user's location, so it should encourage the user to enable location services. (It's not necessarily the user of the app who has denied access to the location data; a systems administrator or parent could also have restricted location access.)

- To show this alert, add the following lines to `getLocation()`, just before you set the `locationManager`'s delegate:

```
if authStatus == .denied || authStatus == .restricted {  
    showLocationServicesDeniedAlert()  
    return  
}
```

This shows the alert if the authorization status is denied or restricted. Notice the use of `||` here, the “logical or” operator. `showLocationServicesDeniedAlert()` will be called if either of those two conditions is true.

- Try it out. Run the app and tap **Get My Location**. You should now get the Location Services Disabled alert:



The alert that pops up when location services are not available

Fortunately, users can change their minds and enable location services for your app again. This is done from the device's Settings app.

- Open the **Settings** app in the simulator and go to **Privacy ▶ Location Services**.



Location Services in the Settings app

- Click **MyLocations** and then **While Using the App** to enable location services again. Switch back to the app (or run it again from Xcode) and press the **Get My Location** button.

If you try it, the following message will appear in Xcode's debug area:

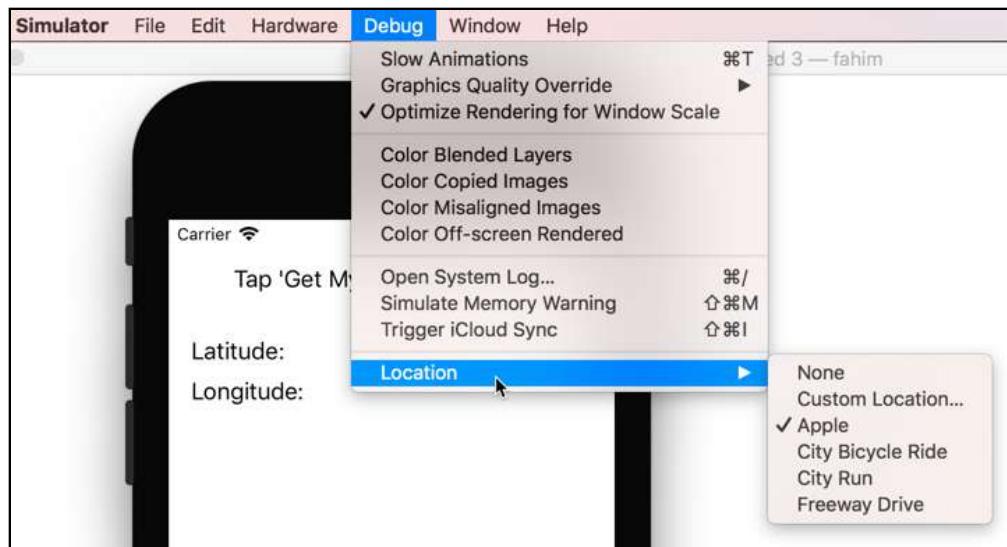
```
didFailWithError The operation couldn't be completed.  
(kCLErrorDomain error 0.)
```

Again there is an error message but with a different code, 0. This is “location unknown” which means Core Location was unable to obtain a location for some reason.

That is not so strange, as you're running this from the simulator, which obviously does not have a real GPS. Your Mac may have a way to obtain location information through Wi-Fi but this is not built into the simulator. Fortunately, there is a way to fake it!

Fake location on the simulator

- With the app running, from the simulator's menu bar at the top of the screen, choose **Debug** ▶ **Location** ▶ **Apple**.



The simulator's Location menu

You should now see messages like these in the debug area:

```
didUpdateLocations <+37.33259552,-122.03031802> +/- 500.00m  
(speed -1.00 mps / course -1.00) @ 6/30/17, 8:19:11 AM Israel  
Daylight Time  
didUpdateLocations <+37.33241211,-122.03050893> +/- 65.00m  
(speed -1.00 mps / course -1.00) @ 6/30/17, 8:19:13 AM Israel  
Daylight Time  
didUpdateLocations <+37.33240901,-122.03048800> +/- 65.00m  
(speed -1.00 mps / course -1.00) @ 6/30/17, 8:19:14 AM Israel  
Daylight Time
```

It keeps going on and on, giving the app a new location every second or so. These particular coordinates point at the Apple headquarters in Cupertino, California.

Look carefully at the coordinates the app is receiving. The first one says “ $+/- 500.00m$,” the second one “ $+/- 65.00m$,” a little further on “ $+/- 50.00m$ ” etc. This number keeps getting smaller and smaller until it stops at about “ $+/- 5.00m$.”

This is the accuracy of the measurement, expressed in meters. What you see is the simulator imitating what happens when you ask for a location on a real device.

If you go out with an iPhone and try to obtain location information, the iPhone uses three different ways to find your coordinates. It has onboard cellular, Wi-Fi and GPS radios that each give it location information at different levels of detail:

- Cell tower triangulation will always work if there is a signal but it’s not very precise.
- Wi-Fi positioning works better, but that is only available if there are known Wi-Fi routers nearby. This system uses a big database that contains the locations of wireless networking equipment.
- The very best results come from the GPS (**Global Positioning System**), but that needs satellite communication and is therefore the slowest of the three. It also won’t work very well indoors.

So, your device has several ways of obtaining location data, ranging from fast but inaccurate (cell towers, Wi-Fi) to accurate but slow (GPS). And none of these are guaranteed to work. Some devices don’t even have a GPS or cellular radio at all and have to rely on just Wi-Fi. Suddenly obtaining a location seems a lot trickier.

Fortunately for us, Core Location does all of the hard work of turning the location readings from its various sources into a useful number. Instead of making you wait for the definitive results from the GPS — which may never come — Core Location sends location data to the app as soon as it gets it, and then follows up with more and more accurate readings.

Exercise: If you have an iPhone, iPod touch or iPad nearby, try the app on your device and see what kind of readings it gives you. If you have more than one device, try the app on all of them and note the differences.

Asynchronous operations

Obtaining a location is an example of an **asynchronous** process.

Sometimes apps need to do things that may take a while. After you start an operation, you have to wait until it gives you the results. If you’re unlucky, those results may never come at all!

In the case of Core Location, it can take a second or two before you get the first location reading and then quite a few seconds more to get coordinates that are accurate enough for your app to use.

Asynchronous means that after you start such an operation, your app will continue on its merry way. The user interface is still responsive, new events are being sent and handled, and the user can still tap on things.

The asynchronous process is said to be operating “in the background.” As soon as the operation is done, the app is notified through a delegate so that it can process the results.

The opposite is **synchronous** (without the a). If you start an operation that is synchronous, the app won’t continue until that operation is done. In effect, the app freezes up.

In the case of `CLLocationManager` that would cause a big problem: your app would be totally unresponsive for the couple of seconds that it takes to get a location fix. Those kinds of “blocking” operations are often a bad experience for the user.

For example, *MyLocations* has a tab bar at the bottom. If the app blocked while getting the location, switching to another tab during that time would have no effect. The user expects to always be able to change tabs, but now it appears that the app is frozen, or worse, has crashed.

The designers of iOS decided that such behavior is unacceptable and therefore operations that take longer than a fraction of a second should be performed in an asynchronous manner.

For the next app, you’ll see more asynchronous processing in action when we talk about network connections and downloading stuff from the Internet.

By the way, iOS has something called a *watchdog timer*. If your app is unresponsive for too long, then under certain circumstances, the watchdog timer will kill your app without mercy — so don’t do anything that freezes your UI!

The take-away is this: any operation that takes long enough to be noticeable by the user should be done asynchronously, in the background.



Displaying coordinates

The `locationManager(_:_didUpdateLocations:)` delegate method gives you an array of `CLLocation` objects that contain the current latitude and longitude coordinates of the user. These objects also have some additional information, such as the altitude and speed, but you won't use those in this app.

You'll take the last `CLLocation` object from the array — because that is the most recent update — and display its coordinates in the labels that you added to the screen earlier.

- Add a new instance variable to `CurrentLocationViewController.swift`:

```
var location: CLLocation?
```

You will store the user's current location in this variable. This needs to be an optional, because it is possible to *not* have a location, for example, when you're stranded out in the Sahara desert somewhere and there are no cell towers or GPS satellites in sight (it happens).

But even when everything works as it should, the value of `location` will still be `nil` until Core Location reports back with a valid `CLLocation` object, which as you've seen, may take a few seconds. So an optional it is.

- Change `locationManager(_:_didUpdateLocations:)` to:

```
func locationManager(_ manager: CLLocationManager,  
didUpdateLocations locations: [CLLocation]) {  
    let newLocation = locations.last!  
    print("didUpdateLocations \(newLocation)")  
  
    location = newLocation // Add this  
    updateLabels() // Add this  
}
```

You store the `CLLocation` object that you get from the location manager into the instance variable and call a new `updateLabels()` method.

Keep the `print()` in there because it's handy for debugging.

- Add the `updateLabels()` method:

```
func updateLabels() {  
    if let location = location {  
        latitudeLabel.text = String(format: "%.8f",  
            location.coordinate.latitude)
```

```
longitudeLabel.text = String(format: "%.8f",
                             location.coordinate.longitude)
tagButton.isHidden = false
messageLabel.text = ""
} else {
    latitudeLabel.text = ""
    longitudeLabel.text = ""
    addressLabel.text = ""
    tagButton.isHidden = true
    messageLabel.text = "Tap 'Get My Location' to Start"
}
}
```

Because the `location` instance variable is an optional, you use the `if let` syntax to unwrap it.

Note the *shadowing* of the original `location` variable by the unwrapped variable. Inside the `if` statement, `location` now refers to an actual `CLLocation` object that is not `nil`.

If there is a valid location object, you convert the latitude and longitude, which are values with type `Double`, into strings and put them into the labels.

You've seen *string interpolation* before to put values into strings, so why doesn't this code simply do the following?

```
latitudeLabel.text = "\(location.coordinate.latitude)"
```

That would certainly work, but it doesn't give you any control over how the latitude value appears. For this app, you want both latitude and longitude to be shown with 8 digits behind the decimal point.

For that sort of control, you need to use a *format string*.

Format strings

Like string interpolation, a format string uses placeholders that will be replaced by the actual value during runtime. These placeholders, or *format specifiers*, can be quite intricate.

To create the text for the latitude label you do this:

```
String(format: "%.8f", location.coordinate.latitude)
```

This creates a new `String` object using the format string `"%.8f"` and the value to replace in that string, `location.coordinate.latitude`.

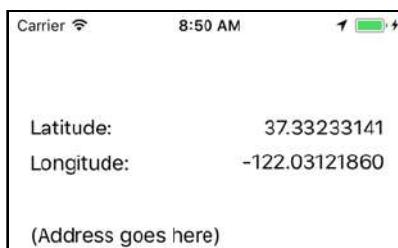


Placeholders always start with a percent (%) sign. Examples of common placeholders are: %d for integer values, %f for floating-point and %@ for objects.

Format strings are very common in Objective-C code, but less so in Swift because string interpolation is much simpler (but less powerful).

The %.8f format specifier does the same thing as %f: it takes a decimal number and puts it in the string. The .8 means that there should always be 8 digits behind the decimal point.

- Run the app, select a location to simulate from the simulator's **Debug** menu and tap the **Get My Location** button. You'll now see the latitude and longitude appear on the screen.



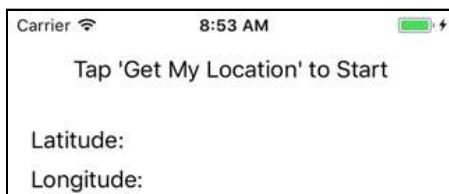
The app shows the GPS coordinates

When the app starts up, it has no location object (location is still nil) and therefore ought to show the "Tap 'Get My Location' to Start" message at the top as a hint to the user. But it doesn't do that yet since the app doesn't call updateLabels() until it receives the first coordinates.

- To fix this, also call updateLabels() from viewDidLoad():

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

- Run the app. Initially, the screen should now say, Tap 'Get My Location' to Start, and the latitude and longitude labels are empty.



What the app looks like when you start it

You can find the project files for this chapter under **27 - Get Location Data** in the Source Code folder.



Chapter 28: Use Location Data

By Eli Ganim

You've learned how to get GPS coordinate information from the device and to display the information on screen.

In this chapter, you will learn the following:

- **Handle GPS errors:** Receiving GPS information is an error-prone process. How do you handle the errors?
- **Improve GPS results:** How to improve the accuracy of the GPS results you receive.
- **Reverse geocoding:** Getting the address for a given set of GPS coordinates.
- **Testing on device:** Testing on device to ensure that your app handles real-world scenarios.
- **Support different screen sizes:** Setting up your UI to work on iOS devices with different screen sizes.



Handling GPS errors

Getting GPS coordinates is error-prone. You may be somewhere where there is no clear line-of-sight to the sky — such as inside or in an area with lots of tall buildings — blocking your GPS signal.

There may not be many Wi-Fi routers around you, or they haven't been catalogued yet, so the Wi-Fi radio isn't much help getting a location fix either.

And of course your cellular signal might be so weak that triangulating your position doesn't offer particularly good results either.

All of that is assuming your device actually has a GPS or cellular radio. I just went out with my iPod touch to capture coordinates and get some pictures for this app. In the city center it was unable to obtain a location fix. My iPhone did better, but it still wasn't ideal.

The moral of this story is that your location-aware apps had better know how to deal with errors and bad readings. There are no guarantees that you'll be able to get a location fix, and if you do, then it might still take a few seconds.

This is where software meets the real world. You should add some error handling code to the app to let users know about problems getting those coordinates.

The error handling code

- Add these two instance variables to **CurrentLocationViewController.swift**:

```
var updatingLocation = false  
var lastLocationError: Error?
```

- Change `locationManager(_:didFailWithError:)` to the following:

```
func locationManager(_ manager: CLLocationManager,  
                     didFailWithError error: Error) {  
    print("didFailWithError \(error.localizedDescription)")  
  
    if (error as NSError).code ==  
        CLError.locationUnknown.rawValue {  
        return  
    }  
    lastLocationError = error  
    stopLocationManager()  
    updateLabels()  
}
```

The location manager may report errors for a variety of scenarios. You can look at the `code` property of the `Error` object to find out what type of error you're dealing with. (You do need to cast to `NSError` first since that is the subclass of `Error` that actually contains the `code` property.)

Some of the possible Core Location errors:

- `CLError.locationUnknown` — the location is currently unknown, but Core Location will keep trying.
- `CLError.denied` — the user denied the app permission to use location services.
- `CLError.network` — there was a network-related error.

There are more (having to do with the compass and geocoding), but you get the point. Lots of reasons for things to go wrong!

Note: These error codes are defined in the `CLError` enumeration. Recall that an enumeration, or `enum`, is a list of values and names for these values.

The error codes used by Core Location have simple integer values. Rather than using the values 0, 1, 2 and so on in your program, Core Location has given them symbolic names using the `CLError` enum. That makes these codes easier to understand and you're less likely to pick the wrong one.

To convert these names back to an integer value you ask for the `rawValue`.

In your updated `locationManager(_:didFailWithError:)`, you do:

```
if (error as NSError).code == CLError.locationUnknown.rawValue {  
    return  
}
```

The `CLError.locationUnknown` error means the location manager was unable to obtain a location right now, but that doesn't mean all is lost. It might just need another second or so to get an uplink to the GPS satellite. In the mean time, it's letting you know that, for now, it could not get any location information.

When you get this error, you will simply keep trying until you do find a location or receive a more serious error.

In the case of a more serious error, you store the error object into a new instance variable, `lastLocationError`:

```
lastLocationError = error
```

That way, you can look up later what kind of error you were dealing with. This comes in useful in `updateLabels()`. You'll be modifying that method shortly to show the error to the user because you don't want to leave them in the dark about such things.

Exercise: Can you explain why `lastLocationError` is an optional?

Answer: When there is no error, `lastLocationError` will not have a value. In other words, it can be `nil`, and variables that can be `nil` must be optionals in Swift.

Finally, the update to `locationManager(_:didFailWithError:)` adds a new method call:

```
stopLocationManager()
```

Stopping location updates

If obtaining a location appears to be impossible for wherever the user currently is on the globe, then you need to tell the location manager to stop. To conserve battery power, the app should power down the iPhone's radios as soon as it doesn't need them anymore.

If this was a turn-by-turn navigation app, you'd keep the location manager running even in the case of a network error, because who knows, a couple of meters ahead you might get a valid location.

For this app, the user will simply have to press the Get My Location button again if they want to try in another spot.

► Add the `stopLocationManager()` method:

```
func stopLocationManager() {
    if updatingLocation {
        locationManager.stopUpdatingLocation()
        locationManager.delegate = nil
        updatingLocation = false
    }
}
```

There's an if statement that checks whether the boolean instance variable `updatingLocation` is true or false. If it is false, then the location manager isn't currently active and there's no need to stop it.

The reason for having this `updatingLocation` variable is that you are going to change the appearance of the Get My Location button and the status message label when the app is trying to obtain a location fix, to let the user know the app is working on it.

- Put some extra code in `updateLabels()` to show the error message:

```
func updateLabels() {
    if let location = location {
        .
        .
        .
        // Remove the following line
        messageLabel.text = "Tap 'Get My Location' to Start"
        // The new code starts here:
        let statusMessage: String
        if let error = lastLocationError as NSError? {
            if error.domain == kCLErrorDomain &&
                error.code == CLError.denied.rawValue {
                statusMessage = "Location Services Disabled"
            } else {
                statusMessage = "Error Getting Location"
            }
        } else if !CLLocationManager.locationServicesEnabled() {
            statusMessage = "Location Services Disabled"
        } else if updatingLocation {
            statusMessage = "Searching..."
        } else {
            statusMessage = "Tap 'Get My Location' to Start"
        }
        messageLabel.text = statusMessage
    }
}
```

The new code determines what to put in the `messageLabel` at the top of the screen. It uses a bunch of if statements to figure out what the current status of the app is. If the location manager gave an error, the label will show an error message.

The first error it checks for is `CLError.denied` in the error domain `kCLErrorDomain`, which means Core Location errors. In that case, the user has not given this app permission to use the location services. That sort of defeats the purpose of this app but it can happen, and you have to check for it anyway.

If the error code is something else, then you simply say "Error Getting Location" as this usually means there was no way of obtaining a location fix.



Even if there was no error, it might still be impossible to get location coordinates if the user disabled Location Services completely on their device (instead of just for this app). You check for that situation with the `locationServicesEnabled()` method of `CLLocationManager`.

Suppose there were no errors and everything works fine, then the status label will say “Searching...” before the first location object has been received.

If your device can obtain the location fix quickly, then this text will be visible only for a fraction of a second, but often, it might take a short while to get that first location fix. No one likes waiting, so it’s nice to let the user know that the app is actively looking up their location. That is what you’re using the `updatingLocation` boolean for.

Note: You put all this logic into a single method because that makes it easy to change the screen when something has changed. Received a location? Simply call `updateLabels()` to refresh the contents of the screen. Received an error? Let `updateLabels()` sort it out...

Starting location updates

- Also add a new `startLocationManager()` method — I suggest you put it right above `stopLocationManager()`, to keep related functionality together:

```
func startLocationManager() {
    if CLLocationManager.locationServicesEnabled() {
        locationManager.delegate = self
        locationManager.desiredAccuracy =
            kCLLocationAccuracyNearestTenMeters
        locationManager.startUpdatingLocation()
        updatingLocation = true
    }
}
```

Starting the location manager used to happen in the `getLocation()` action method. However, because you now have a `stopLocationManager()` method, it makes sense to move the start code into a method of its own, `startLocationManager()`, just to keep things symmetrical.

The only difference from before is that this checks whether the location services are enabled and you set the variable `updatingLocation` to `true` if you did indeed start location updates.



- Change `getLocation()` to:

```
@IBAction func getLocation() {  
    if authStatus == .denied || authStatus == .restricted {  
        // New code below, replacing existing code after this point  
        startLocationManager()  
        updateLabels()  
    }  
}
```

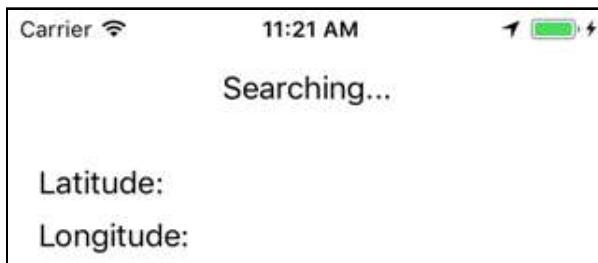
There is one more small change to make. Suppose there was an error and no location could be obtained, but then you walk around for a bit and a valid location comes in. In that case, it's a good idea to remove the old error code.

- At the bottom of `locationManager(_:didUpdateLocations:)`, add the following line just before calling `updateLabels()`:

```
lastLocationError = nil
```

This clears out the old error state. After receiving a valid coordinate, any previous error you may have encountered is no longer applicable.

- Run the app. While the app is waiting for incoming coordinates, the label at the top should say “Searching...” until it finds a valid coordinate or encounters a fatal error.

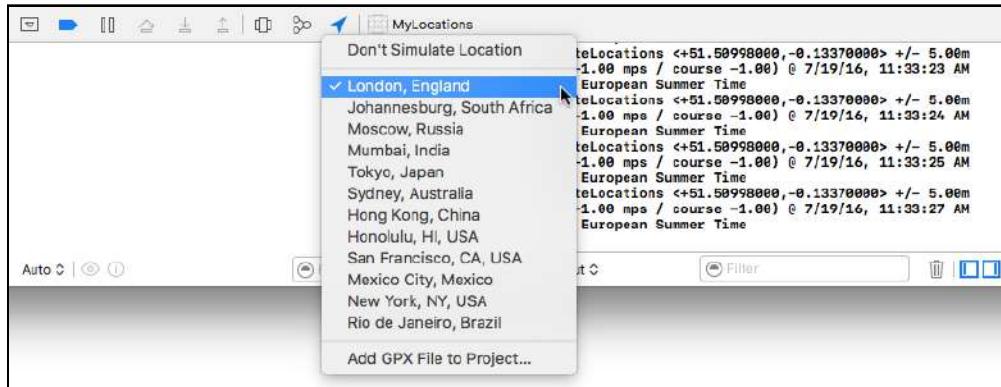


The app is waiting to receive GPS coordinates

Play around with the Simulator’s location settings for a while and see what happens when you choose different locations.

Note that changing the Simulator’s location to None isn’t an error anymore. This still returns the `.locationUnknown` error code but you ignore that because it’s not a fatal error.

Tip: You can also simulate locations from within Xcode. If your app uses Core Location, the bar at the top of the debug area gets an arrow icon. Click on that icon to change the simulated location:



Simulating locations from within the Xcode debugger

Ideally, you should not just test in the Simulator but also on your device, as you're more likely to encounter real errors that way.

Improving GPS results

Cool, you know how to obtain a `CLLocation` object from Core Location and you're able to handle errors. Now what?

Well, here's the thing: You saw in the Simulator that Core Location keeps giving you new location objects over and over, even though the coordinates may not have changed. That's because the user could be on the move, in which case their GPS coordinates *do* change.

However, you're not building a navigation app. So, for *MyLocations* you just want to get a location that is accurate enough and then you can tell the location manager to stop sending updates.

This is important because getting location updates costs a lot of battery power as the device needs to keep its GPS/Wi-Fi/cellular radios powered up for this. This app doesn't need to ask for GPS coordinates all the time, so it should stop when the location is accurate enough.

The problem is that you can't always get the accuracy you want, so you have to detect this. When the last couple of coordinates you received aren't increasing in accuracy then that is probably as good as it's going to get, and you should let the

radio power down.

Getting results for a specific accuracy level

- Change `locationManager(_:didUpdateLocations:)` to the following:

```
func locationManager(_ manager: CLLocationManager,  
didUpdateLocations locations: [CLLocation]) {  
    let newLocation = locations.last!  
    print("didUpdateLocations \(newLocation)")  
  
    // 1  
    if newLocation.timestamp.timeIntervalSinceNow < -5 {  
        return  
    }  
  
    // 2  
    if newLocation.horizontalAccuracy < 0 {  
        return  
    }  
  
    // 3  
    if location == nil || location!.horizontalAccuracy >  
        newLocation.horizontalAccuracy {  
  
        // 4  
        lastLocationError = nil  
        location = newLocation  
  
        // 5  
        if newLocation.horizontalAccuracy <=  
            locationManager.desiredAccuracy {  
            print("*** We're done!")  
            stopLocationManager()  
        }  
        updateLabels()  
    }  
}
```

Let's take these changes one by one:

1. If the time at which the given location object was determined is too long ago — 5 seconds in this case — then this is a *cached* result.

Instead of returning a new location fix, the location manager may initially give you the most recently found location under the assumption that you might not have moved much in the last few seconds — obviously, this does not take into consideration people with jet packs.

You'll simply ignore these cached locations if they are too old.

2. To determine whether new readings are more accurate than previous ones, you'll use the `horizontalAccuracy` property of the location object. However, sometimes locations may have a `horizontalAccuracy` that is less than 0. In which case, these measurements are invalid and you should ignore them.
3. This is where you determine if the new reading is more useful than the previous one. Generally speaking, Core Location starts out with a fairly inaccurate reading and then gives you more and more accurate ones as time passes. However, there are no guarantees — so, you cannot assume that the next reading truly is always more accurate.

Note that a larger accuracy value means *less* accurate — after all, accurate up to 100 meters is worse than accurate up to 10 meters. That's why you check whether the previous reading, `location!.horizontalAccuracy`, is greater than the new reading, `newLocation.horizontalAccuracy`.

You also check for `location == nil`. Recall that `location` is an optional instance variable that stores the `CLLocation` object that you obtained in a previous call to `didUpdateLocations`. If `location` is `nil`, then this is the very first location update you're receiving and in that case you should continue.

So, if this is the very first location reading (`location` is `nil`) or the new location is more accurate than the previous reading, you continue to step 4. Otherwise you ignore this location update.

4. You've seen this part before. It clears out any previous error and stores the new `CLLocation` object into the `location` variable.
5. If the new location's accuracy is equal to or better than the desired accuracy, you can call it a day and stop asking the location manager for updates. When you started the location manager in `startLocationManager()`, you set the desired accuracy to 10 meters (`kCLLocationAccuracyNearestTenMeters`), which is good enough for this app.

Short circuiting

Because `location` is an optional object, you cannot access its properties directly — you first need to unwrap it. You could do that with `if let`, but if you're sure that the optional is not `nil` you can also *force unwrap* it with `!`.

That's what you are doing in this line:

```
if location == nil || location!.horizontalAccuracy >
    newLocation.horizontalAccuracy {
```

You wrote `location!.horizontalAccuracy` with an exclamation point instead of just `location.horizontalAccuracy`.

But what if `location == nil`, won't the force unwrapping fail then? Not in this case, because the force unwrap is never performed.

The `||` operator (logical or) tests whether either of the two conditions is true. If the first one is true (`location` is `nil`), it will not evaluate the second condition. That's called *short circuiting*. There is no need for the app to check the second condition if the first one is already true.

So, the app will only look at `location!.horizontalAccuracy` when `location` is guaranteed to be non-`nil`. Blows your mind, eh?

- Run the app. First set the Simulator's location to None, then press Get My Location. The screen now says "Searching..."
- Switch to location Apple (but don't press Get My Location again). After a brief moment, the screen is updated with GPS coordinates as they come in.

If you check the Xcode Console, you'll get about 10 location updates before it says "**** We're done!" and the location updates stop.

Note: It's possible the above steps won't work for you. If the screen does not say "Searching..." but shows an old set of coordinates instead, then the Simulator is holding on to old location data. This seems to happen when you pick a location from within Xcode (using the arrow in the debug area) instead of the Simulator's Debug menu.

The quickest way to fix this is to quit the Simulator and run the app again — this launches a new Simulator. If you can't get it to work, no worries, it's not that important. Just be aware that the Simulator can be finicky sometimes.

You, as the developer, can tell from the Console when the location updates stop, but obviously, the user won't see this.

The Tag Location button becomes visible as soon as the first location is received so the user can start saving this location to their library right away, but at this point the location may not be accurate enough yet. So it's nice to show the user when the app has found the most accurate location.

Updating the UI

To make this clearer, you are going to toggle the Get My Location button to say “Stop” when the location grabbing is active and switch it back to “Get My Location” when it’s done. That gives a nice visual clue to the user. Later on, you’ll also show an animated activity spinner that makes this even more obvious.

To change the state of the button, you’ll add a `configureGetButton()` method.

- Add the following method to `CurrentLocationViewController.swift`:

```
func configureGetButton() {  
    if updatingLocation {  
        getButton.setTitle("Stop", for: .normal)  
    } else {  
        getButton.setTitle("Get My Location", for: .normal)  
    }  
}
```

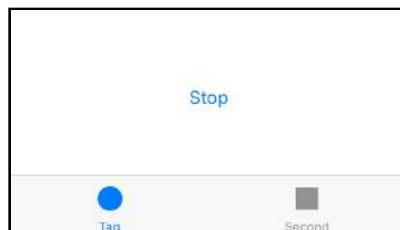
It’s quite simple: if the app is currently updating the location, then the button’s title becomes Stop, otherwise it is Get My Location.

You need to now call `configureGetButton()` from several different places in your code. If you look closely, you’ll notice that wherever you call `updateLabels()`, you also need to call the new method. So might as well call the new method from within `updateLabels()`, right?

- Add a call to `configureGetButton()` at the end of `updateLabels()`:

```
func updateLabels() {  
    // ...  
    configureGetButton()  
}
```

- Run the app again and perform the same test as before. The button changes to Stop when you press it. When there are no more location updates, it switches back.



The stop button

When a button says “Stop,” you naturally expect to be able to press it so you can interrupt the location updates. This is especially so when you’re not getting any coordinates at all. Eventually Core Location may give an error, but as a user, you may not want to wait for that.

Currently, however, pressing Stop doesn’t stop anything. You have to change `getLocation()` for this, as any taps on the button call this method.

- In `getLocation()`, replace the line with the call to `startLocationManager()` with the following:

```
if updatingLocation {  
    stopLocationManager()  
} else {  
    location = nil  
    lastLocationError = nil  
    startLocationManager()  
}
```

Again, you’re using the `updatingLocation` flag to determine what state the app is in.

If the button is pressed while the app is already doing the location fetching, you stop the location manager.

Note that you also clear out the old location and error objects before you start looking for a new location.

- Run the app. Now pressing the Stop button will put an end to the location updates. You should see no more updates in the Console after you press Stop.

Note: If the Stop button doesn’t appear long enough for you to click it, set the location back to None first, tap Get My Location a few times, and then select the Apple location again.

Reverse geocoding

The GPS coordinates you’ve dealt with so far are just numbers. The coordinates 37.33240904, -122.03051218 don’t really mean that much, but the address 1 Infinite Loop in Cupertino, California does.



Using a process known as **reverse geocoding**, you can turn a set of coordinates into a human-readable address. (Regular or “forward” geocoding does the opposite: it turns an address into GPS coordinates. You can do both with the iOS SDK, but for *MyLocations* you only do the reverse one.)

You’ll use the `CLGeocoder` object to turn the location data into a human-readable address and then display that address on screen.

It’s quite easy to do this but there are some rules. You’re not supposed to send out a ton of these reverse geocoding requests at the same time. The process of reverse geocoding takes place on a server hosted by Apple and it costs them bandwidth and processor time to handle these requests. If you flood their servers with requests, Apple won’t be happy.

MyLocations is only supposed to be used occasionally. So theoretically, its users won’t be spamming the Apple servers, but you should still limit the geocoding requests to one at a time, and once for every unique location. After all, it makes no sense to reverse geocode the same set of coordinates over and over.

Reverse geocoding needs an active Internet connection and anything you can do to prevent unnecessary use of the iPhone’s radios is a good thing for your users.

The implementation

- Add the following properties to `CurrentLocationViewController.swift`:

```
let geocoder = CLGeocoder()
var placemark: CLPlacemark?
var performingReverseGeocoding = false
var lastGeocodingError: Error?
```

These mirror what you did for the location manager. `CLGeocoder` is the object that will perform the geocoding and `CLPlacemark` is the object that contains the address results.

The `placemark` variable needs to be an optional because it will have no value when there is no location yet, or when the location doesn’t correspond to a street address — I don’t think it will respond with “Sahara desert, Africa,” but to be fair, I haven’t had the chance to try.

You set `performingReverseGeocoding` to `true` when a geocoding operation is taking place, and `lastGeocodingError` will contain an `Error` object if something went wrong, or `nil` if there is no error.

- You'll put the geocoder to work in `locationManager(didUpdateLocations)`. Add these lines right under the call to `updateLabels()`:

```
if !performingReverseGeocoding {  
    print("/** Going to geocode")  
  
    performingReverseGeocoding = true  
  
    geocoder.reverseGeocodeLocation(newLocation,  
                                    completionHandler: {  
        placemarks, error in  
        if let error = error {  
            print("/** Reverse Geocoding error: \(  
                error.localizedDescription)")  
            return  
        }  
        if let places = placemarks {  
            print("/** Found places: \(places)")  
        }  
    })  
}
```

The app should only perform a single reverse geocoding request at a time. So, first you check whether it is busy by looking at the `performingReverseGeocoding` variable. Then you start the geocoder.

The code looks straightforward enough, right? If you are wondering what the `completionHandler` bit is, harken back to chapter 6 when you used a similar construct to handle a `UIAlertController` action — it's a *closure*.

Closures

Unlike the location manager, `CLGeocoder` does not use a delegate to return results from an operation. Instead, it uses a closure. Closures are an important Swift feature and you can expect to see them all over the place — for Objective-C programmers, a closure is similar to a “block.”

Closures can have parameters too and here, the parameters for the closure are `placemarks` and `error`, both of which are optionals because either one or the other can be `nil` depending on the situation.

So, while all the code inside the closure does print out either the list of places or the error, you do have to unwrap each optional before you do that to be sure that you have a value there.

Unlike the rest of the code in `locationManager(_:didUpdateLocations:)`, the code in the closure is not performed right away. After all, you can only print the



geocoding results once the geocoding completes, and that may be several seconds later.

The closure is kept for later use by the `CLGeocoder` object and is only performed after `CLGeocoder` finds an address or encounters an error.

So why does `CLGeocoder` use a closure instead of a delegate?

The problem with using a delegate to provide feedback is that you need to write one or more separate methods. For example, for `CLLocationManager` there are the `locationManager(_:didUpdateLocations:)` and `locationManager(_:didFailWithError:)` methods.

By creating separate methods, you move the code that deals with the response away from the code that makes the request. With closures, on the other hand, you can put that handling code in the same place. That makes the code more compact and easier to read. Some APIs do both, and you have a choice between using a closure or becoming a delegate.

So when you write,

```
geocoder.reverseGeocodeLocation(newLocation, completionHandler:  
{ placemarks, error in  
    // put your statements here  
})
```

you're telling the `CLGeocoder` object that you want to reverse geocode the location, and that the code in the block following `completionHandler:` should be executed as soon as the geocoding is completed.

The closure itself is:

```
{ placemarks, error in  
    // put your statements here  
}
```

The items before the `in` keyword — `placemarks` and `error` — are the parameters for this closure and they work just like parameters for a method or a function.

When the geocoder finds a result for the location object that you gave it, it invokes the closure and executes the statements within. The `placemarks` parameter will contain an array of `CLPlacemark` objects that describe the address information, and the `error` variable contains an error message in case something went wrong.

Closures are basically the same principle as using delegate methods, except you're not putting the code into a separate method but in a closure.



It's OK if closures have got you scratching your head right now. You'll see them used many more times in the upcoming chapters.

- Run the app and pick a location. As soon as the first location is found, you can see in the Console that the reverse geocoder has kicked in (give it a second or two):

```
didUpdateLocations <+37.33233141,-122.03121860> +/- 379.75m
(speed -1.00 mps / course -1.00) @ 7/1/17, 10:31:15 AM Israel
Daylight Time
*** Going to geocode
*** Found places: [Apple Inc., Apple Inc., 1 Infinite Loop,
Cupertino, CA 95014, United States @
<+37.33233141,-122.03121860> +/- 100.00m, region
CLCircularRegion (identifier:'<+37.33233140,-122.03121860>
radius 141.73', center:<+37.33233140,-122.03121860>,
radius:141.73m)]
```

If you choose the Apple location, you'll see that some location readings are duplicates; the geocoder only does the first of those. Only when the accuracy of the reading improves does the app reverse geocode again. Nice!

Note: Several readers have reported that if you are in China and are trying to reverse geocode an address that is outside of China, you may get an error and placemarks will be `nil` – try a location inside China instead.

Handling reverse geocoding errors

- Replace the contents of the geocoding closure with the following:

```
self.lastGeocodingError = error
if error == nil, let p = placemarks, !p.isEmpty {
    self.placemark = p.last!
} else {
    self.placemark = nil
}

self.performingReverseGeocoding = false
self.updateLabels()
```

Just as with the location manager, you store the error object so you can refer to it later – you do use a different instance variable this time, `lastGeocodingError`.

The next line does something you haven't seen before:

```
if error == nil, let p = placemarks, !p.isEmpty {
```

You know that `if let` is used to unwrap optionals. Here, `placemarks` is an optional, so it needs to be unwrapped before you can use it or you risk crashing the app when `placemarks` is `nil`. The unwrapped `placemarks` array gets the temporary name `p`.

The `!p.isEmpty` bit says that we should only enter this `if` statement if the array of placemark objects is not empty.

You should read this line as:

```
if there's no error and the unwrapped placemarks array is not  
empty {
```

Of course, Swift doesn't speak English, so you have to express this in terms that Swift understands.

You could also have written this as three different, nested `if` statements:

```
if error == nil {  
    if let p = placemarks {  
        if !p.isEmpty {
```

But it's just as easy to combine all of these conditions into a single `if`.

You're doing a bit of **defensive programming** here: you specifically check first whether the array has any objects in it. If there is no error, then it should have at least one object, but you're not going to trust that it always will. Good developers are paranoid!

If all three conditions are met — there is no error, the `placemarks` array is not `nil`, and there is at least one `CLPlacemark` inside this array — then you take the last of those `CLPlacemark` objects:

```
self.placemark = p.last!
```

The `last` property refers to the last item from an array. It's an optional because there is no last item if the array is empty. As an alternative, you can also write `placemarks[placemarks.count - 1]` but that's not as tidy.

Usually there will be only one `CLPlacemark` object in the array, but there is the odd situation where one location coordinate may refer to more than one address. This app can only handle one address at a time. So, you'll just pick the last one, which usually is the only one.

If there was an error during geocoding, you set `self.placemark` to `nil`. Note that you did not do that for the locations. If there was an error there, you kept the previous location object because it may actually be correct (or good enough) and it's better than nothing. But for the address that makes less sense.

You don't want to show an old address, only the address that corresponds to the current location or no address at all.

In mobile development, nothing is guaranteed. You may get coordinates back or you may not, and if you do, they may not be very accurate. The reverse geocoding will probably succeed if there is some type of network connection available, but you also need to be prepared to handle the case where there is none.

And remember, not all GPS coordinates correspond to actual street addresses — there is no corner of 52nd and Broadway in the Sahara desert.

Note: Did you notice that inside the `completionHandler` closure you used `self` to refer to the view controller's properties and methods? This is a Swift requirement.

Closures are said to *capture* all the variables they use and `self` is one of them. You can forget about that immediately, if you like; just know that Swift requires that all captured variables are explicitly mentioned.

As you've seen, outside a closure, you can use `self` to refer to properties and methods, but it's not a requirement. However, you do get a compiler error if you leave out `self` inside a closure. So you don't have much choice there.

Displaying the address

Let's show the address to the user.

► Modify `updateLabels()` like this:

```
func updateLabels() {
    if let location = location {

        // Add this block
        if let placemark = placemark {
            addressLabel.text = String(from: placemark)
        } else if performingReverseGeocoding {
            addressLabel.text = "Searching for Address..."
        } else if lastGeocodingError != nil {
            addressLabel.text = "Error Finding Address"
        }
    }
}
```

```
    } else {
        addressLabel.text = "No Address Found"
    }
    // End new code
} else {
    ...
}
```

Because you only do the address lookup once the app has a valid location, you just have to change the code inside the first `if` branch. If you've found an address, you show that to the user, otherwise you show a status message.

The code to format the `CLPlacemark` object into a string is placed in its own method, just to keep the code readable.

► Add the `string(from)` method:

```
func string(from placemark: CLPlacemark) -> String {
    // 1
    var line1 = ""

    // 2
    if let s = placemark.subThoroughfare {
        line1 += s + " "
    }

    // 3
    if let s = placemark.thoroughfare {
        line1 += s
    }

    // 4
    var line2 = ""

    if let s = placemark.locality {
        line2 += s + " "
    }
    if let s = placemark.administrativeArea {
        line2 += s + " "
    }
    if let s = placemark.postalCode {
        line2 += s
    }

    // 5
    return line1 + "\n" + line2
}
```

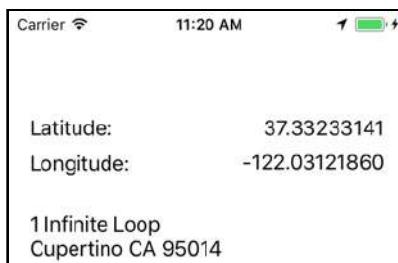
Let's look at this in detail:

1. The address will be two lines of text — create a new string variable for the first line of text.
2. If the placemark has a `subThoroughfare`, add it to the string. This is an optional property, so you unwrap it with `if let` first. Just so you know, `subThoroughfare` is a fancy name for house number.
3. Adding the `thoroughfare` (or street name) is done similarly. Note that you put a space between it and `subThoroughfare` so they don't get glued together.
4. The same logic goes for the second line of text. This adds the locality (the city), administrative area (the state or province), and postal code (or zip code), with spaces between them where appropriate.
5. Finally, the two lines are concatenated (added together) with a newline character in between. The `\n` adds the line break (or newline) to the string.

► In `getLocation()`, clear out the `placemark` and `lastGeocodingError` variables to start with a clean slate. Put this just above the call to `startLocationManager()`:

```
placemark = nil  
lastGeocodingError = nil
```

► Run the app again. Seconds after a location is found, the address label should be filled in as well.



Reverse geocoding finds the address for the GPS coordinates

It's fairly common that street numbers or other details are missing from the address. The `CLPlacemark` object may contain incomplete information, which is why its properties are all optionals. Geocoding is not an exact science!

Exercise: If you pick the City Bicycle Ride or City Run locations from the Simulator's Debug menu, you should see in the Console that the app jumps

through a whole bunch of different coordinates — it simulates someone moving from one place to another. However, the coordinates on the screen and the address label don't change nearly as often. Why is that?

Answer: The logic for *MyLocations* was designed to find the most accurate set of coordinates for a stationary position. You only update the `location` variable when a new set of coordinates comes in that is more accurate than previous readings. Any new readings with a higher — or the same — `horizontalAccuracy` value are simply ignored, regardless of what the actual coordinates are.

With the City Bicycle Ride and City Run options, the app doesn't receive the same coordinates with increasing accuracy but a series of completely different coordinates. That means this app doesn't work very well when you're on the move — unless you press Stop and try again —, but that's also not what it was intended for.

Note: If you're playing with different locations in the Simulator or from the Xcode debugger menu and you get stuck, then the quickest way to get unstuck is to reset the Simulator. Sometimes it just doesn't want to move to a new location even if you tell it to, and then you have to show it who's the boss!

Testing on device

When I first wrote this code, I had only tested it on the Simulator. It worked fine there. Then, I put it on my iPod touch and guess what? Not so good.

The problem with the iPod touch is that it doesn't have GPS, so it relies only on Wi-Fi to determine the location. But Wi-Fi might not be able to give you accuracy up to ten meters; I got +/- 100 meters at best.

Right now, you only stop the location updates when the accuracy of the reading falls within the `desiredAccuracy` setting — something that will never actually happen on my iPod touch.

That goes to show that you can't always rely on the Simulator to test your apps. You need to put them on your device and test them in the wild, especially when using device-dependent functionality like location-based APIs. If you have more than one device, then test on all of them! In order to deal with this situation, you will improve upon the `didUpdateLocations` delegate method.



First fix

- Change locationManager(_:didUpdateLocations:) to:

```
func locationManager(_ manager: CLLocationManager,  
didUpdateLocations locations: [CLLocation]) {  
    . . .  
  
    if newLocation.horizontalAccuracy < 0 {  
        return  
    }  
  
    // New section #1  
    var distance = CLLocationDistance(  
        Double.greatestFiniteMagnitude)  
    if let location = location {  
        distance = newLocation.distance(from: location)  
    }  
    // End of new section #1  
    if location == nil || location!.horizontalAccuracy >  
        newLocation.horizontalAccuracy {  
  
        if newLocation.horizontalAccuracy <=  
            locationManager.desiredAccuracy {  
  
            // New section #2  
            if distance > 0 {  
                performingReverseGeocoding = false  
            }  
            // End of new section #2  
        }  
        updateLabels()  
        if !performingReverseGeocoding {  
            . . .  
  
            // New section #3  
        } else if distance < 1 {  
            let timeInterval = newLocation.timestamp.timeIntervalSince(  
                location!.timestamp)  
            if timeInterval > 10 {  
                print("!!! Force done!")  
                stopLocationManager()  
                updateLabels()  
            }  
            // End of new sectiton #3  
        }  
    }  
}
```



It's a pretty long method now, but only the three highlighted sections were added. This is the first one:

```
var distance = CLLocationDistance(  
    Double.greatestFiniteMagnitude)  
if let location = location {  
    distance = newLocation.distance(from: location)  
}
```

This calculates the distance between the new reading and the previous reading, if there was one. We can use this distance to measure if our location updates are still improving.

If there was no previous reading, then the distance is `Double.greatestFiniteMagnitude`. That is a built-in constant that represents the maximum value that a `Double` value can have. This little trick gives it a gigantic distance if this is the very first reading. You're doing that so any of the following calculations still work even if you weren't able to calculate a true distance yet.

You also add an `if` statement later where you stop the location manager:

```
if distance > 0 {  
    performingReverseGeocoding = false  
}
```

This forces a reverse geocoding for the final location, even if the app is already currently performing another geocoding request.

You absolutely want the address for that final location, as that is the most accurate location you've found. But if some previous location was still being reverse geocoded, this step would normally be skipped.

Simply by setting `performingReverseGeocoding` to `false`, you always force the geocoding to be done for this final coordinate.

(Of course, if `distance` is 0, then this location is the same as the location from a previous reading, and you don't need to reverse geocode it anymore.)

The real improvement is found in the final new section:

```
} else if distance < 1 {  
    let timeInterval = newLocation.timestamp.timeIntervalSince(  
        location!.timestamp)  
    if timeInterval > 10 {  
        print("!!! Force done!")  
        stopLocationManager()  
        updateLabels()
```

```
    }
```

If the coordinate from this reading is not significantly different from the previous reading and it has been more than 10 seconds since you've received that original reading, then it's a good point to hang up your hat and stop.

It's safe to assume you're not going to get a better coordinate than this and you can stop fetching the location.

This is the improvement that was necessary to make my iPod touch stop scanning after some time. It wouldn't give me a location with better accuracy than +/- 100 meters, but it kept repeating the same one over and over.

A time limit of 10 seconds seems to give good results.

Note that you don't just say:

```
} else if distance == 0 {
```

The distance between subsequent readings is never exactly 0. It may be something like 0.0017632. Rather than checking for equals to 0, it's better to check for less than a certain distance, in this case one meter.

By the way, did you notice how you used `location!` to unwrap it before accessing the timestamp property? When the app gets inside this `else-if`, the value of `location` is guaranteed to be non-`nil`, so its safe to force unwrap the optional.

► Run the app and test that everything still works. It may be hard to recreate this situation on the Simulator, but try it on your device inside the house and see what output you see in the Console.

There is another improvement you can make to increase the robustness of this logic, and that is to set a time-out on the whole thing. You can tell iOS to perform a method one minute from now. If by that time the app hasn't found a location yet, you stop the location manager and show an error message.



Second fix

- First add a new instance variable:

```
var timer: Timer?
```

- Then change `startLocationManager()` to:

```
func startLocationManager() {
    if CLLocationManager.locationServicesEnabled() {

        timer = Timer.scheduledTimer(timeInterval: 60, target: self,
                                      selector: #selector(didTimeOut), userInfo: nil,
                                      repeats: false)
    }
}
```

The new lines set up a timer object that sends a `didTimeOut` message to `self` after 60 seconds; `didTimeOut` is the name of a method.

A *selector* is the term that Objective-C uses to describe the name of a method, and the `#selector()` syntax is how you create a selector in Swift.

- Change `stopLocationManager()` to:

```
func stopLocationManager() {
    if updatingLocation {

        if let timer = timer {
            timer.invalidate()
        }
    }
}
```

You have to cancel the timer in case the location manager is stopped before the time-out fires. This happens when an accurate enough location is found within one minute after starting, or when the user taps the Stop button.

- Finally, add the `didTimeOut()` method:

```
@objc func didTimeOut() {
    print("== Time out")
    if location == nil {
        stopLocationManager()
        lastLocationError = NSError(
            domain: "MyLocationsErrorDomain",
            code: 1, userInfo: nil)
        updateLabels()
    }
}
```

```
    }
```

There's something new about this method — there's a new `@objc` attribute before `func` — whatever could it be?

Remember how `#selector` is an Objective-C concept? (How could you forget, it was just a few paragraphs ago, right?) So, when you use `#selector` to identify a method to call, that method has to be accessible not only from Swift, but from Objective-C as well. The `@objc` attribute allows you to identify a method (or class, or property, or even enumeration) as being accessible from Objective-C.

So, that's what you've done for `didTimeOut` — declared it as being accessible from Objective-C.

`didTimeOut()` is always called after one minute, whether you've obtained a valid location or not — unless `stopLocationManager()` cancels the timer first.

If after that one minute there still is no valid location, you stop the location manager, create your own error code, and update the screen.

By creating your own `NSError` object and putting it into the `lastLocationError` instance variable, you don't have to change any of the logic in `updateLabels()`.

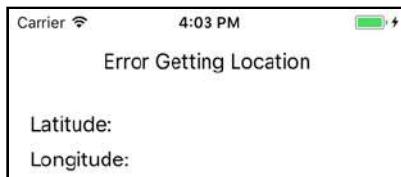
However, you do have to make sure that the error's domain is not `kCLErrorDomain` because this error object does not come from Core Location but from within your own app.

An error domain is simply a string, so `MyLocationsErrorDomain` will do. For the code I picked 1. The value of the code doesn't really matter at this point because you only have one custom error, but you can imagine that when an app becomes bigger, you might need multiple error codes.

Note that you don't always have to use an `NSError` object; there are other ways to let the rest of your code know that an error occurred. In this case `updateLabels()` was already using an `NSError` anyway, so having your own error object just made sense.

- Run the app. Set the Simulator location to None and press **Get My Location**.

After a minute, the debug area should say “*** Time out” and the Stop button reverts to Get My Location. There should also be an error message on the screen:



The error after a time out

Just getting a simple location from Core Location and finding the corresponding street address turned out to be a lot more hassle than it looked. There are many different situations to handle. Nothing is guaranteed, and everything can go wrong — iOS development sometimes requires nerves of steel!

To recap, the app either:

- Finds a location with the desired accuracy,
- Finds a location that is not as accurate as you'd like and you don't get any more accurate readings,
- Doesn't find a location at all,
- Or, takes too long finding a location.

The code now handles all these situations, but I'm sure it's not perfect yet. No doubt the logic could be tweaked more, but it will do for the purposes of this book.

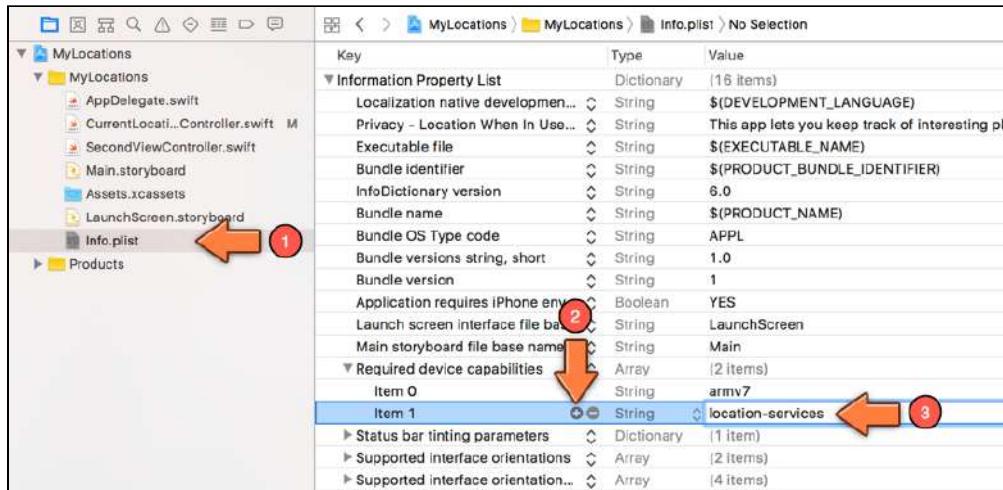
I hope it's clear that if you're releasing a location-based app, you need to do a lot of field testing!

Required device capabilities

The **Info.plist** file has a key, **Required device capabilities**, that lists the hardware that your app needs in order to run. This is the key that the App Store uses to determine whether a user can install your app on their device.

The default value is **armv7**, which is the CPU architecture of the iPhone 3GS and later models. If your app requires additional features, such as Core Location to retrieve the user's location, you should list them here.

► Add a new item with the value **location-services** to **Info.plist**:



Adding location-services to Info.plist

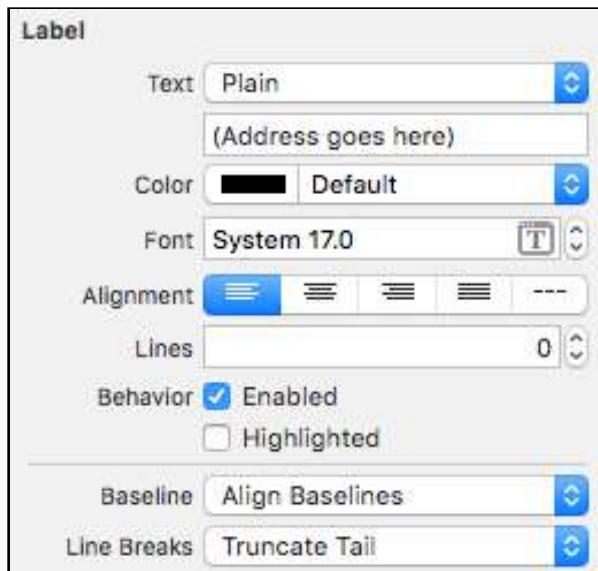
You could also add the item **gps**, in which case the app requires a GPS receiver. But if you did, users won't be able to install the app on an iPod touch or on certain iPads.

For the full list of possible device capabilities, see the *App Programming Guide for iOS* on the Apple Developer website.

P.S. You can now take the `print()` statements out of the app (or simply comment them out). You might want to keep them in there as they're handy for debugging. In an app that you plan to upload to the App Store, you'll definitely want to remove the `print()` statements when development's complete.

Attributes and properties

Most of the attributes in Interface Builder's inspectors correspond directly to properties on the selected object. For example, a `UILabel` has the following attributes:



These are directly related to the following properties:

Text	<code>label.text</code>
Color	<code>label.textColor</code>
Font	<code>label.font</code>
Alignment	<code>label.textAlignment</code>
Lines	<code>label.numberOfLines</code>
Enabled	<code>label.isEnabled</code>
Baseline	<code>label.baselineAdjustment</code>
Line Breaks	<code>label.lineBreakMode</code>

And so on... As you can see, the names may not always be exactly the same ("Lines" and `numberOfLines`) but you can easily figure out which property goes with which attribute.

You can find these properties in the documentation for `UILabel`. From the Xcode Help menu, select **Developer Documentation**. Type “`UILabel`” into the search field to bring up the class reference for `UILabel`:

The screenshot shows the Xcode Help documentation for the `UILabel` class. The search bar at the top contains the text “`UILabel`”. The left sidebar lists various UI components under “Text Views”, with `UILabel` selected. The main content area shows the `UILabel` class definition, its description (a view that displays one or more lines of read-only text), and its properties. The properties listed include `text`, `attributedText`, `font`, `textColor`, `textAlignment`, `lineBreakMode`, `isEnabled`, `adjustsFontSizeToFitWidth`, `allowsDefaultTighteningForTruncation`, `baselineAdjustment`, `minimumScaleFactor`, `numberOfLines`, and `highlightedTextColor`. The right sidebar provides information about the language (Swift and Objective-C), SDKs (iOS 2.0+, tvOS 9.0+), framework (UIKit), and links to “On This Page” (Overview, Topics, Relationships).

The documentation for `UILabel` does not list properties for all of the attributes from the inspectors. For example, in the Attributes inspector there is a section named “View.” The attributes in this section come from `UIView`, which is the base class of `UILabel`. So if you can’t find a property in the `UILabel` class, you may need to check the documentation under the “Inherits From” section (which is under the **Relationships** section) of the documentation.

You can find the project files for this chapter under **28 – Use Location Data** in the Source Code folder.

Chapter 29: Objects vs. Classes

Eli Ganim

Time for something new. Up until now I've been calling almost everything an "object." That's not quite correct though. So, it's time for you to brush up on your programming theory a bit more.

In this chapter, you will learn the following:

- **Classes:** The difference between classes and objects.
- **Inheritance:** What class inheritance is and how it works.
- **Overriding methods:** Overriding methods in sub-classes to provide different functionality.
- **Casts:** Casting an object from a subclass to its superclass — how (and why) you do it.



Classes

If you want to use the proper object-oriented programming vernacular, you have to make a distinction between an object and its **class**.

When you do this:

```
class ChecklistItem: NSObject {  
    . . .  
}
```

You're really defining a class named `ChecklistItem`, not an object. An object is what you get when you **instantiate** a class:

```
let item = ChecklistItem()
```

The `item` variable now contains an object of the class `ChecklistItem`. You can also say: the `item` variable contains an **instance** of the class `ChecklistItem`. The terms object and instance mean the same thing.

In other words, “instance of class `ChecklistItem`” is the **type** of this `item` variable.

The Swift language and the iOS frameworks already come with a lot of types built-in, but you can also add types of your own by making new classes.

Let's use an example to illustrate the difference between a class and an instance / object.

You and I are both hungry, so we decide to eat some ice cream (my favorite subject next to programming!). Ice cream is the class of food that we're going to eat.

The ice cream class looks like this:

```
class IceCream: NSObject {  
    var flavor: String  
    var scoops: Int  
  
    func eatIt() {  
        // code goes in here  
    }  
}
```

You and I go on over to the ice cream stand and ask for two cones:

```
// one for you  
let iceCreamForYou = IceCream()  
iceCreamForYou.flavor = "Strawberry"
```



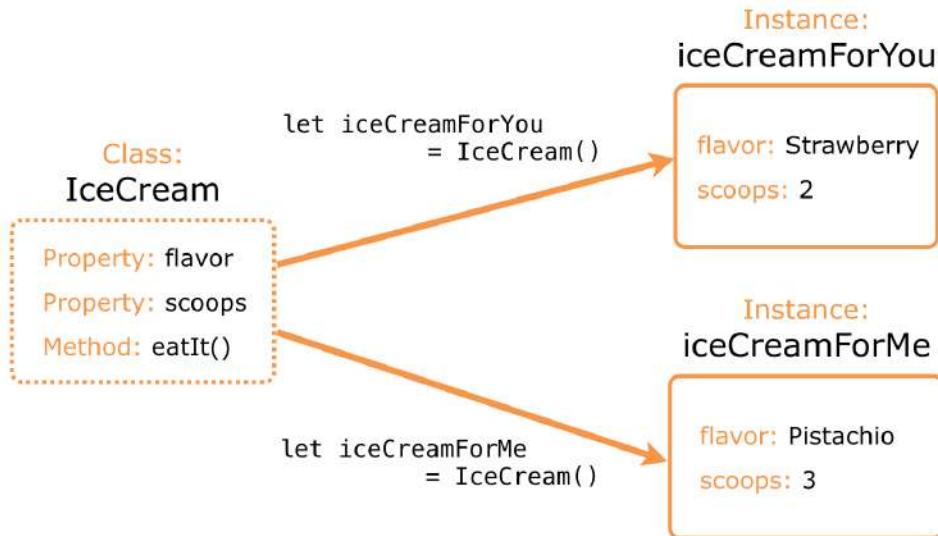
```
iceCreamForYou.scoops = 2

// and one for me
let iceCreamForMe = IceCream()
iceCreamForMe.flavor = "Pistachio"
iceCreamForMe.scoops = 3
```

Yep, I get more scoops, but that's because I'm hungry from all this explaining.

Now the app has two instances of `IceCream`, one for you and one for me. There is just one class that describes what sort of food we're eating — ice cream — but there are two distinct objects. Your object has strawberry flavor, mine pistachio.

The `IceCream` class is like a template that declares: objects of this type have two properties, `flavor` and `scoops`, and a method named `eatIt()`.



The class is a template for making new instances

Any new instance that is made from this template will have those instance variables and methods, but it lives in its own section of computer memory and therefore has its own values.

If you're more into architecture than food, you can also think of a class as a blueprint for a building. It is the design of the building but not the building itself. One blueprint can make many buildings, and you could paint each one — each instance — a different color if you wanted to.

Inheritance

Sorry, this is not where I tell you that you've inherited a fortune. We're talking about **class inheritance** here, one of the main principles of object-oriented programming.

Inheritance is a powerful feature that allows a class to be built on top of another class. The new class takes over all the data and functionality from that other class and adds its own specializations to it.

Take the `IceCream` class from the previous example. It is built on `NSObject`, the fundamental class for iOS frameworks. You can see that in the `class` line that defines `IceCream`:

```
class IceCream: NSObject {
```

This means that `IceCream` is actually the `NSObject` class with a few additions of its own, namely the `flavor` and `scoops` properties and the `eatIt()` method.

`NSObject` is the **base class** for almost all other classes in iOS frameworks. Most objects that you'll encounter are made from a class that either directly inherits from `NSObject`, or from another class that is ultimately based on `NSObject`. You can't escape it!

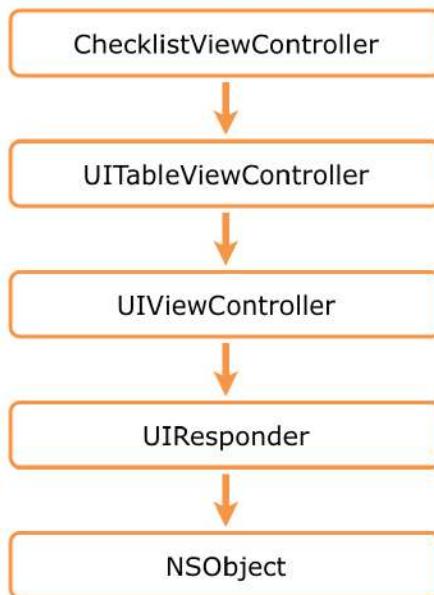
You've also seen class declarations that look like this:

```
class ChecklistViewController: UITableViewController
```

The `ChecklistViewController` class is really a `UITableViewController` class with your own additions. It does everything a `UITableViewController` does, plus whatever new data and functionality you've given it.

This inheritance thing is very handy because `UITableViewController` already does a lot of work for you behind the scenes. It has a table view, it knows how to deal with prototype cells and static cells, and it handles things like scrolling and a ton of other stuff. All you have to do is add your own customizations and you're ready to go.

`UITableViewController` itself is built on top of `UIViewController`, which is built on top of something called `UIResponder`, and ultimately that class is built on `NSObject`. This is called the *inheritance tree*.



All framework classes stand on the shoulders of NSObject

The big idea here is that each object that is higher up performs a more specialized task than the one below it.

`NSObject`, the base class, only provides a few basic functions that are needed by all objects. For example, it contains an `alloc` method that is used to reserve memory space for the object's instance variables, and a basic `init` method.

`UIViewController` is the base class for all view controllers. If you want to make your own view controller, you extend `UIViewController`. To **extend** means that you make a class that inherits from another one. Other commonly used terms are to **derive from** or **to base on** or **to subclass**. These phrases all mean the same thing.

`UIViewController` does way more than you'd think — you really don't want to write all your own screen and view handling code. If you'd had to program each screen totally from scratch, you'd still be working on lesson #1!

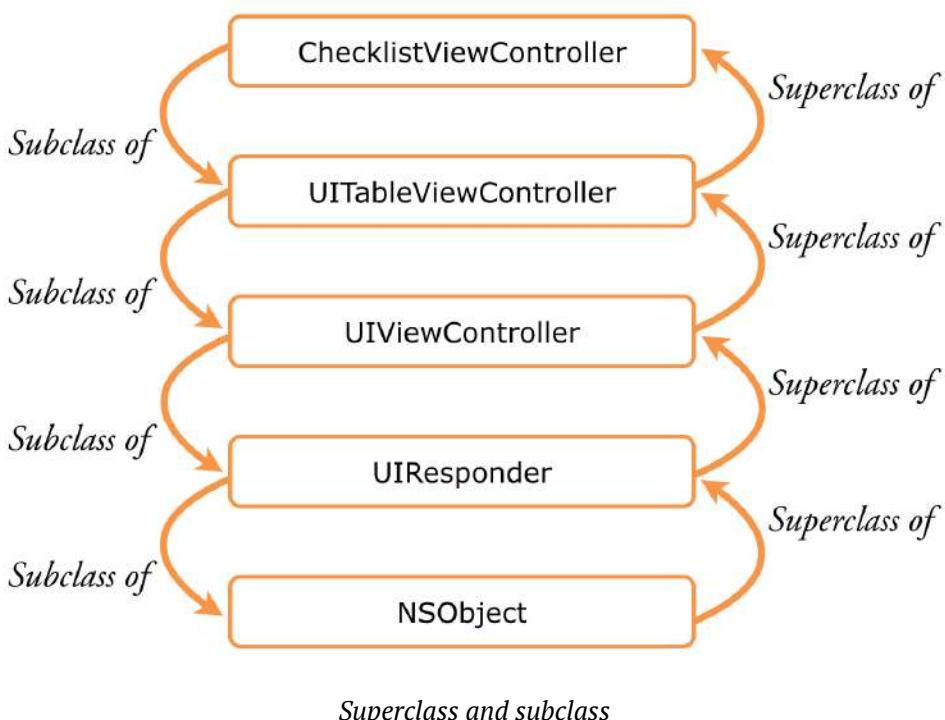
Thank goodness that stuff has been taken care of by very smart people working at Apple and they've bundled it all into `UIViewController`. You simply make a class that inherits from `UIViewController` and you get all that functionality for free. You just add your own data and logic to that class and off you go! If your screen primarily deals with a table view, then you'd subclass `UITableViewController` instead. This class does everything `UIViewController` does — because it inherits from it — but is

more specialized for dealing with table views. You could write all that code by yourself, but why would you, when it's already available in a convenient package? Class inheritance lets you re-use existing code with minimal effort. It can save you a lot of time!

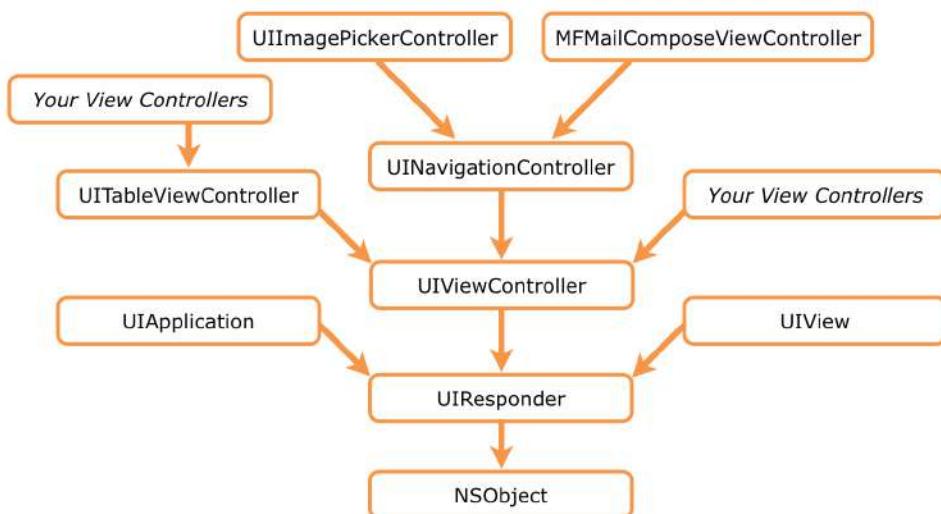
Superclasses and subclasses

When programmers talk about inheritance, they'll often throw around the terms **superclass** and **subclass**.

In the example above, `UITableViewController` is the immediate superclass of `ChecklistViewController`, and conversely `ChecklistViewController` is a subclass of `UITableViewController`. The superclass is the class you derived from (or extended), while a subclass derives from your class.



A class in Swift can have many subclasses but only one immediate superclass. Of course, that superclass can have a superclass of its own. There are many different classes that inherit from `UIViewController`, for example:



A small portion of the UIKit inheritance tree

Because nearly all classes extend from `NSObject`, they form a big hierarchy. It is important that you understand this class hierarchy so you can make your own objects inherit from the proper superclasses.

As you'll see later on, there are many other types of hierarchies in programming. For some reason programmers seem to like hierarchies.

Do note that in Objective-C, all your classes must at least inherit from the `NSObject` class. This is not the case with Swift. You could also have written the `IceCream` class as follows:

```
class IceCream {  
    . . .  
}
```

Now `IceCream` does not have a base class at all. This is fine in pure Swift code, but you might run into troubled waters if you try to use `IceCream` instances in combination with iOS frameworks (which are written in Objective-C). So, sometimes you'll have to use the `NSObject` base class, even if you're writing the app in Swift only.

Inheriting properties (and methods)

Inheriting from a class means your new class gets to use the properties and methods from its superclass. If you create a new base class `Snack`:

```
class Snack {  
    var flavor: String  
    func eatIt() {  
        // code goes in here  
    }  
}
```

And make `IceCream` inherit from that class:

```
class IceCream: Snack {  
    var scoops: Int  
}
```

Then elsewhere in your code you can do:

```
let iceCreamForMe = IceCream()  
iceCreamForMe.flavor = "Chocolate"  
iceCreamForMe.scoops = 1  
iceCreamForMe.eatIt()
```

This works even though `IceCream` did not explicitly declare an `eatIt()` method or `flavor` instance variable. But `Snack` did! Because `IceCream` inherits from `Snack`, it automatically gets the method and instance variable for free.

Overriding methods

In the previous example, `IceCream` could use the `eatIt()` method implementation from `Snack` for free. But that's not the full story! `IceCream` can also provide its own `eatIt()` method if it's important for your app that eating ice cream is different from eating any other kind of snack (for example, you may want to eat it faster, before it melts):

```
class IceCream: Snack {  
    var scoops: Int  
  
    override func eatIt() {  
        // code goes in here  
    }  
}
```

Now, when someone calls `iceCreamForMe.eatIt()`, this new version of the method in the `IceCream` class is invoked. Note that Swift requires you to use the `override` keyword in front of any methods that you provide that already exist in the superclass.

A possible implementation of this overridden version of `eatIt()` could look like this:

```
class IceCream: Snack {
    var scoops: Int
    var isMelted: Bool

    override func eatIt() {
        if isMelted {
            throwAway()
        } else {
            super.eatIt()
        }
    }
}
```

If the ice cream has melted, you want to throw it in the trash. But if it's still edible, you'll call `Snack`'s version of `eatIt()` using `super`.

Just like `self` refers to the current object, the `super` keyword refers to the object's superclass. That is the reason you've been calling `super` in various places in your code, to let any superclasses do their thing.

Something that happens often in iOS frameworks is that methods are used for communicating between a class and its subclasses, so that the subclass can perform specific behavior in certain circumstances. That is what methods such as `viewDidLoad()` and `viewWillAppear(_:)` are for.

These methods are defined and implemented by `UIViewController` but your own view controller subclass can override them.

For example, when its screen is about to become visible, the `UIViewController` class will call `viewWillAppear(true)`. Normally this will invoke the `viewWillAppear(_:)` method from `UIViewController` itself, but if you've provided your own version of this method in your subclass, then yours will be invoked instead.

By overriding `viewWillAppear(_:)`, you get a chance to handle this event before the superclass does:

```
class MyViewController: UIViewController {
    override func viewWillAppear(_ animated: Bool) {
        // do your own stuff before super
```

```
// don't forget to call super!
super.viewDidLoad(animated)

// do your own stuff after super
}
```

That's how you can tap into the power of your superclass. A well-designed superclass provides such "hooks" that allow you to react to certain events.

Don't forget to call `super`'s version of the method, though. If you neglect this, the superclass will not get its own notification and weird things may happen.

You've also seen `override` already in the table view data source methods:

```
override func tableView(_ tableView: UITableView,
                      didSelectRowAt indexPath) {
    ...
}
```

`UITableViewController`, the superclass, already implements these methods. So, if you want to provide your own implementation, you need to override the existing ones.

Note: Inside those table view delegate and data source methods, it's usually not necessary to call `super`. The iOS API documentation can usually tell you whether you need to call `super` or not for an overridden method.

Subclass initialization

When making a subclass, the `init` methods require special care.

If you don't want to change any of the `init` methods from your superclass or add any new `init` methods, then it's easy: You don't have to do anything. The subclass will automatically take over the `init` methods from the superclass.

Most of the time, however, you will want to override an `init` method or add your own. For example, to put values into the subclass's new instance variables. In that case, you may have to override not just that one `init` method but all of them.

In the next app you'll create a class named `GradientView` that extends `UIView`. That app uses `init(frame:)` to create and initialize a `GradientView` object.

`GradientView` overrides this method to set the background color:



```
class GradientView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = UIColor.black
    }
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
    ...
}
```

But because `UIView` also has another `init` method, `init?(coder:)`, `GradientView` needs to implement that method too even if it doesn't do anything but call `super`.

Also note that `init(frame:)` is marked as `override`, but `init?(coder:)` is `required`. The `required` keyword is used to enforce that every subclass always implements this particular `init` method.

Swift wants to make sure that subclasses don't forget to add their own stuff to such required `init` methods, even if the app doesn't actually use that particular `init` method, as in the case of `GradientView` — it can be a bit of an over-concerned parent, that Swift.

The rules for inheritance of `init` methods are somewhat complicated — the official Swift Programming Guide devotes many pages to it — but at least if you make a mistake, Xcode will tell you what's wrong and what you should do to fix it.

Private parts

So... does a subclass get to use all the methods from its superclass? Not quite.

`UIViewController` and other `UIKit` classes have a lot more methods hidden away than you have access to. Often, these secret methods do cool things and it is tempting to use them. But they are not part of the official API, making them off-limits for mere mortals such as you and I.

If you ever hear other developers speak of “private APIs” in hushed tones and down dark alleys, then this is what they are talking about.

It is, in theory, possible to call such hidden methods if you know their names, but this is not recommended. It may even get your app rejected from the App Store, as Apple is known to scan apps for usage of these private APIs.

You're not supposed to use private APIs for two reasons:

1. These APIs may have unexpected side effects and not be as robust as their publicly available relatives.



2. There is no guarantee these methods will exist from one version of iOS to the next. Using them is very risky, as your apps may suddenly stop working.

Sometimes, however, using a private API is the only way to access certain functionality on the device. If so, you're out of luck. Fortunately, for most apps, the official public APIs are more than enough and you won't need to resort to the private stuff.

So how do you mark your own methods as private, I hear you ask? This could get a bit complicated and is probably best left to a more detailed treatment of the subject. But in simple terms, similar to the `@objc` attribute you used in the previous chapter, there are other attributes that you can use to modify the access control level of Swift classes, methods, or properties.

Two of the most common are `public` and `private`. And hopefully, their names alone give you an understanding as to their intent. Since Swift 4.0, `public` is assumed by default. Which is why you have not had to prefix any of your classes or methods with this attribute.

`private` is what you need if you wanted to hide any of your classes, methods, or properties. But a discussion as to how `private` works in terms of what is hidden if you use the attribute and the advantages of doing so, might be a bit too broad a subject for now.

Casts

Often, your code will refer to an instance not by its own class but by one of its superclasses. That probably sounds very weird, so let's look at an example.

`MyLocations` has a `UITabBarController` with three tabs, each of which is represented by a view controller. The view controller for the first tab is `CurrentLocationViewController`. Later on you'll add two others, `LocationsViewController` for the second tab, and `MapViewController` for the third.

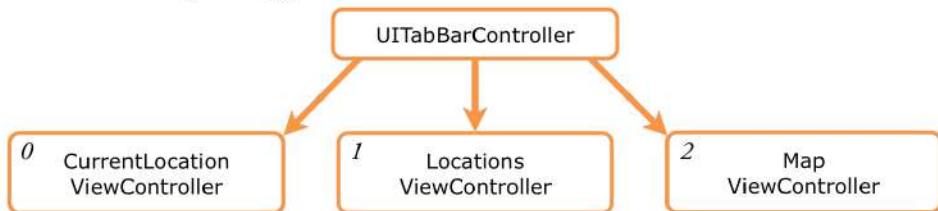
The designers of iOS obviously didn't know anything about those three particular view controllers when they created `UITabBarController`. The only thing the tab bar controller can reliably depend on is that each tab has a view controller that inherits from `UIViewController`.

So, instead of talking to the `CurrentLocationViewController` class, the tab bar controller only sees its superclass part, `UIViewController`.

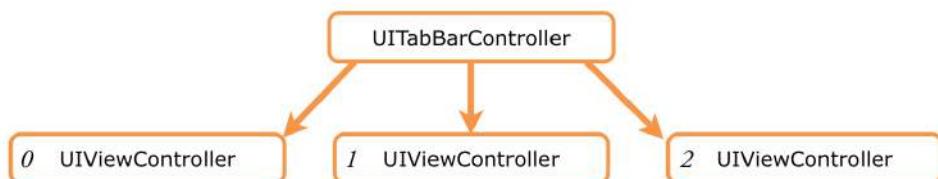


As far as the tab bar controller is concerned, it has three `UIViewController` instances and it doesn't know or care about the additions that you've made to each one.

The structure of the app:



What the tab bar controller sees:



The UITabBarController does not see your subclasses

The same thing goes for `UINavigationController`. To the navigation controller, any new view controllers that get pushed on the navigation stack are all instances of `UIViewController`, nothing more, nothing less.

Sometimes that can be a little annoying. When you ask the navigation controller for one of the view controllers on its stack, it returns a reference to a `UIViewController` instance, even though that is not the full type of that object.

If you want to treat that object as your own view controller subclass instead, you need to **cast** it to the proper type.

Previously you did the following in `prepare(for:sender:)`:

```

let controller = segue.destination as!
ItemDetailViewController
controller.delegate = self
  
```

Here, you wanted to get the segue's destination view controller — which is an instance of `ItemDetailViewController` — and set its `delegate` property.



However, the segue's destination property won't give you an object of type `ItemDetailViewController`. The value it returns is of the plain `UIViewController` type, which naturally doesn't have your `delegate` property.

If you were write the above code without the `as! ItemDetailViewController` bit, like so:

```
let controller = segue.destination
```

Then, Xcode would show an error for the line below it. Swift now infers the type of `controller` to be `UIViewController`, but `UIViewController` does not have a `delegate` property. That property is something you added to the subclass, `ItemDetailViewController`.

You know that `destination` refers to an `ItemDetailViewController`, but Swift doesn't. Even though all `ItemDetailViewController`s are `UIViewController`s, not all `UIViewController`s are `ItemDetailViewController`s!

Just because your friend Chuck has no hair, that doesn't mean all bald guys are named Chuck. Or, that all guys named Chuck have no hair, either!

To solve this problem, you have to cast the object to the proper type. You, as the developer, know this particular object is an `ItemDetailViewController`, so you use the `as!` cast operator to tell the compiler, "I want to treat this object as an `ItemDetailViewController`."

With the cast, the code looks like this:

```
let controller = segue.destination as! ItemDetailViewController
```

Now, you can treat the value from `controller` as an `ItemDetailViewController` object. But... the compiler can't check whether the thing you're casting really is that kind of object. So, if you're wrong and it's not, your app will most likely crash.

Casts can fail for other reasons, too. For example, the value that you're trying to cast may actually be `nil`. If that's a possibility, it's a good idea to use the `as?` operator to make it an optional cast. You must also store the result of the cast into an optional value or use `if let` to safely unwrap it.

Note that a cast doesn't magically convert one type to another. You can't cast an `Int` to a `String`, for example. You only use a cast to make a type more specific, and the two types have to be compatible for this to work.

Casting is very common in Swift programs because of the Objective-C heritage of the iOS frameworks. You'll be doing a lot of it!



To summarize, there are three kinds of casts you can perform:

1. **as?** for casts that are allowed to fail. This would happen if the object is `nil` or doesn't have a type that is compatible with the one you're trying to cast to. It will try to cast to the new type and if it fails, then no biggie. This cast returns an optional that you can unwrap with `if let`.
2. **as!** for casts between a class and one of its subclasses. This is also known as a *downcast*. As with implicitly unwrapped optionals, this cast is potentially unsafe and you should only use `as!` when you are certain it cannot possibly go wrong. You often need to use this cast when dealing with objects coming from UIKit and other iOS frameworks. Better get used to all those exclamation marks!
3. **as** for casts that can never possibly fail. Swift can sometimes guarantee that a type cast will always work, for example between `NSString` and `String`. In that case you can leave off the `?` or the `!` and just write `as`.

It can sometimes be confusing to decide which of these three cast operators you need. If so, just type “as” and Xcode will suggest the correct variant. You can rely on Xcode.



30

Chapter 30: The Tag Location Screen

Eli Ganim

There is a big button on the main screen of the app that says **Tag Location**. It only becomes active when GPS coordinates have been captured, and you use it to add a description and a photo to that location.

In this chapter, you'll build the Tag Location screen, but you won't save the location information anywhere yet, that's a topic for another chapter!

This chapter covers the following:

- **The Screen:** What the finished screen looks like and what it will do.
- **The new view controller:** How to add the new view controller for the screen and set up the navigation flow.
- **Make the cells:** Create the table view cells for displaying information.
- **Display location info:** Display location info on screen via the new view.
- **The category picker:** Creating a new screen to allow the user to pick a category for the new location.



The screen

The Tag Location screen is a regular table view controller with static cells. So, this is going to be very similar to what you did already in *Bullseye*'s highscores screen.

The finished Tag Location screen will look like this:



The Tag Location screen

The description cell (the empty area above the Category cell) at the top contains a UITextView for text. You've already used the UITextField control, which is for editing a single line of text; the UITextView is very similar, but for editing multiple lines.

Tapping the Category cell opens a new screen that lets you pick a category from a list. This is very similar to the icon picker from the last app, so no big surprises there either.

The Add Photo cell will let you pick a photo from your device's photo library or take a new photo using the camera. You'll skip this feature for now and build that later on. Let's not get ahead of ourselves and try too much at once!

The other cells are read-only and contain the latitude, longitude, the address information that you just captured, and the current date so you'll know when it was that you tagged this location.

Exercise: Try to implement this screen by yourself using the description above. You don't have to make the Category and Add Photo buttons work yet. Yikes, that seems like a big job! It sure is, but you should be able to pull this off. This screen doesn't do anything you haven't done previously. So if you feel brave, go ahead!

The new view controller

- Add a new file to the project using the **Swift File** template. Name the file **LocationDetailsViewController**.

You know what's next: create outlets and connect them to the controls on the storyboard. In the interest of saving time, I'll just give you the code that you're going to end up with.

- Replace the contents of **LocationDetailsViewController.swift** with the following:

```
import UIKit

class LocationDetailsViewController: UITableViewController {
    @IBOutlet weak var descriptionTextView: UITextView!
    @IBOutlet weak var categoryLabel: UILabel!
    @IBOutlet weak var latitudeLabel: UILabel!
    @IBOutlet weak var longitudeLabel: UILabel!
    @IBOutlet weak var addressLabel: UILabel!
    @IBOutlet weak var dateLabel: UILabel!

    // MARK:- Actions
    @IBAction func done() {
        navigationController?.popViewControllerAnimated(true)
    }

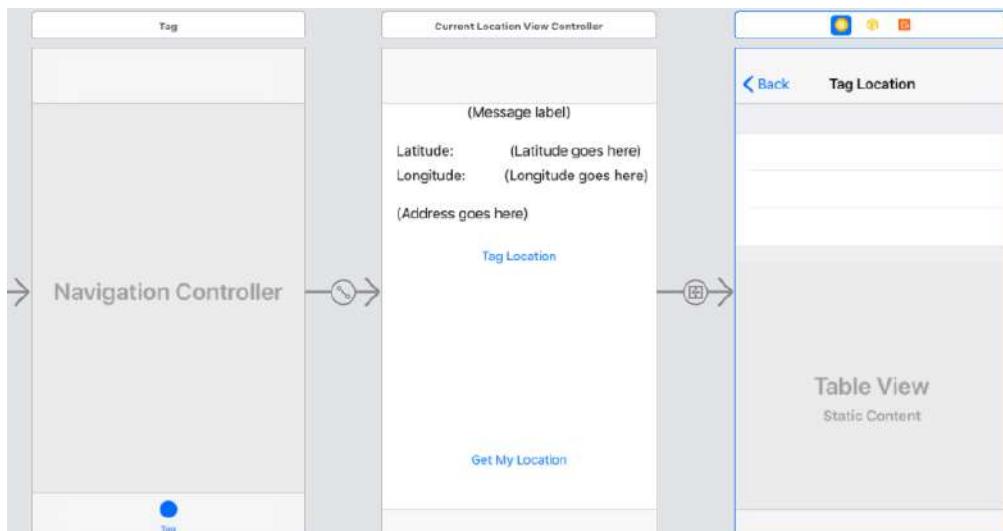
    @IBAction func cancel() {
        navigationController?.popViewControllerAnimated(true)
    }
}
```

Nothing special here, just a bunch of outlet properties and two action methods that both go back to the previous view in the navigation stack.



- In the storyboard, select the Current Location View Controller (the Tag Scene), and choose **Editor** ▶ **Embed In** ▶ **Navigation Controller** from Xcode's menu bar to put it inside a new navigation controller. (This sets up all the views on that particular tab of the tab view controller to be part of a navigation stack.)
- Drag a new **Table View Controller** on to the canvas and put it next to the Tag Scene.
- In the **Identity inspector**, change the **Class** attribute of the table view controller to **LocationDetailsViewController** to link it with the source code file you just created.
- **Control-drag** from the **Tag Location** button on the Tag Scene to the new view controller and create a **Show segue**. Give the segue the identifier **TagLocation**.
- Add a Navigation Item to the Location Details View Controller, and change the title to **Tag Location**.
- Switch the table content to **Static Cells** and its style to **Grouped**.

The storyboard should now look like this:



The Tag Location screen in the storyboard

Hiding the navigation bar

You'll notice that the Tag Scene (the Current Location View Controller) now has an empty navigation bar area. This is because it is now embedded in a Navigation Controller. You can either set the title (and/or make it a large title), or, you can hide the navigation bar altogether for the first view.

For this particular app design, having no titles would look the best. So, you now have to hide the navigation bar at runtime for only the Tag Scene. How do you do it?

Simple enough. It's just a code change!

- Switch to **CurrentLocationViewController.swift** and add a new `viewWillAppear` implementation:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    navigationController?.isNavigationBarHidden = true
}
```

All you do is ask the navigation controller to hide the navigation bar when this particular view is about to appear. Simple as that!

- Run the app and make sure the Tag Location button works.

Do you notice an issue when you switch to the Location Details View Controller via the Tag Location button?

The navigation bar on the new screen is hidden as well! Can you guess why this is?

Yep, it's because you hid the navigation controller's navigation bar in the previous screen. That setting is not a per-screen setting. It affects the navigation bar for the navigation controller from that point onwards for all views displayed by the navigation controller.

So how do you fix it? Simple enough, ask the navigation controller to start showing the navigation bar as soon as you exit the view where you hide the navigation bar. And there is a handy `viewWillDisappear` method that you can override in `UIViewController` that's just the place for this kind of code.

- Add the following method to **CurrentLocationViewController.swift**:

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    navigationController?.isNavigationBarHidden = false
}
```



You simply reverse what you did previously in `viewWillAppear` by asking the navigation controller to show the navigation bar each time the current view is about to disappear from view — usually, either because another view appeared on top of it, or because this view was dismissed in order to go back to a previous view.

- Run the app again and make sure that the navigation flow (and the showing/hiding of the navigation bar) works correctly.

Adding navigation buttons

Of course, the new screen won't do anything useful yet. Let's add some buttons.

- Drag a **Bar Button Item** on to the left slot (where the Back button currently is) of the navigation bar. Make it a **Cancel** button and connect it to the **cancel** action. If you're using the Connections inspector, the thing that you're supposed to connect is the Bar Button Item's "selector," under Sent Actions.

Note: A navigation bar usually has left and right navigation item positions where you can drag either bar button items or views on to. If you are unable to drag an item on to the left/right positions of a navigation bar and the scene has a navigation bar, it is possible that the scene is missing a Navigation Item. Then, you have to first drag a Navigation Item on to the scene.

- Also drag a **Bar Button Item** on to the right slot. Set both the **Style** and **System Item** attributes to **Done**, and connect it to the **done** action.
- Run the app again and make sure you can close the Tag Location screen from both buttons after you've opened it.

Making the cells

There will be three sections in this table view:

1. The description text view and the category cell. These can be changed by the user.
2. The photo. Initially this cell says Add Photo but once the user has picked a photo, you'll display the actual photo inside the cell. It's good to have that in a section of its own.

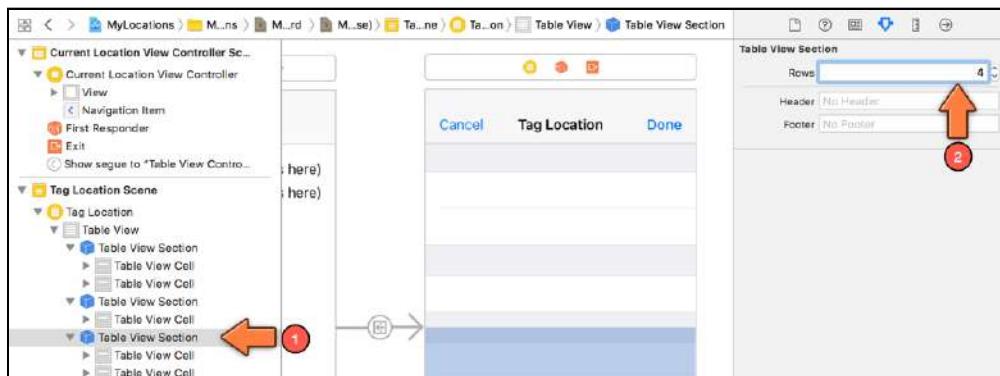


3. The latitude, longitude, address, and date rows. These are read-only information.

- Open the storyboard. Select the table view and go to the **Attributes inspector**. Change the **Sections** field from 1 to 3.

When you do this, the contents of the first section are automatically copied to the new sections. That isn't quite what you want. So, you'll have to remove some rows here and there. The first section will have 2 rows, the middle section will have just 1 row, and the last section will have 4 rows.

- Select one cell in the first section and delete it. If it won't delete, make sure you selected the whole Table View Cell and not its Content View. The Document Outline can be very useful here.
- Delete two cells from the middle section.
- Select the last Table View Section object — use the Document Outline for easy selection — and in the **Attributes inspector** set its **Rows** to 4.



Adding a row to a table view section

Alternatively, you can drag a new Table View Cell from the Object Library on to the section.

The right detail cells

The second row from the first section, and the first, second and fourth rows in the last section will all use a standard cell style.

- Select these cells – you can select multiple items via the Document Outline by Command-clicking – and set their **Style** attribute to **Right Detail**.



The cells with the Right Detail style

The labels in these standard cell styles are regular `UILabels`. So, you can select them and change their properties.

- Change the titles for the labels on the left, from top to bottom to: **Category**, **Latitude**, **Longitude**, and **Date**.

(If Xcode moves the label when you type into it or cuts off the text, then change the cell style to Left Detail and back again to Right Detail. That seems to fix it.)

- Drag a new **Label** into the cell in the middle section (the one that's still empty). You cannot use a standard cell style for this cell. So, you'll design it yourself. Name this label **Add Photo**. (Later on you'll also add an image view to this cell.)

- Make sure the font of the label is **System**, size **17**, so it's the same size as the

labels from the Right Detail cell style. If necessary, use **Editor ➤ Size to Fit Content** to resize the label to its optimal size.

- Add a **left** Auto Layout Constraint — with a value of **0**, and have **Constrain to margins** checked — and also add a constraint to center **Vertically in Container**.

This will add some of the Auto Layout constraints you need to position the label, but not all of them. You will notice that you have a warning still at this point — this is due to the label not having a right constraint. Since we'll be adding an image to this cell later and that would require changes to the right constraint, we will live with the warning for the time being...

The table should now look like this:

Category	Detail
Add Photo	
Latitude	Detail
Longitude	Detail
Date	Detail

The labels in the Tag Location screen

Note: You're going to make a bunch of changes that are the same for each cell. For some of these, it is easier if you select all the cells at once and then change the setting. That will save you some time.

Unfortunately, some menu items and options are grayed out when you have a multiple selection, so you'll still have to change some of the settings for each cell individually.

Tappable cells

Only the Category and Add Photo cells should handle taps, so you have to set the cell selection color to None on the other cells.

- Select all the cells except Category and Add Photo. In the **Attributes inspector**, set **Selection** to **None**.
- Select the Category and Add Photo cells and set **Accessory** to **Disclosure Indicator**.



Category and Add Photo now have a disclosure indicator

The address cell

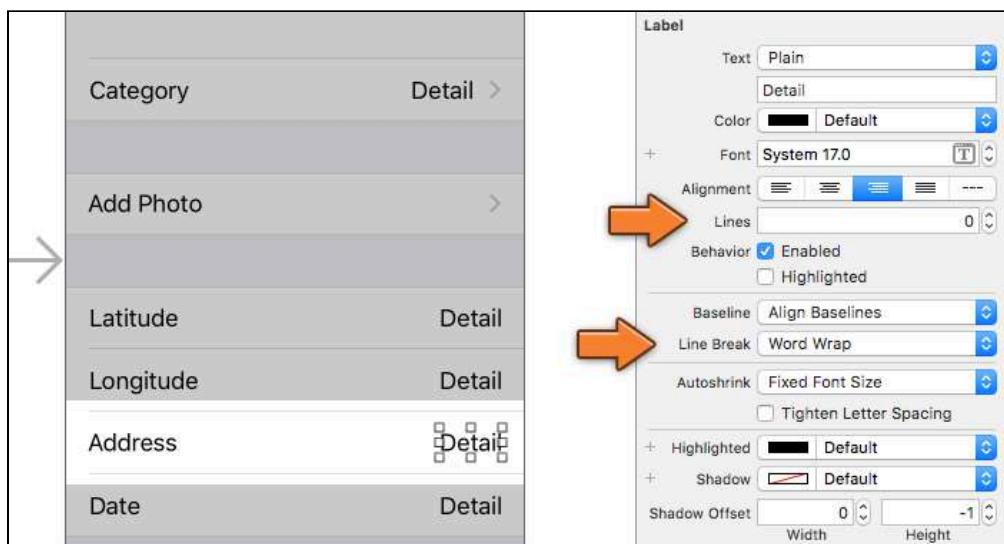
The empty cell in the last section is for the Address label. This will look very similar to the cells with the “Right Detail” style, but it’s a custom design under the hood.

- Drag a new **Label** into that cell and set its title to **Address**.
- Add a **left** Auto Layout constraint (of **0**) to the label and also center **Vertically in Container**.
- Drag another **Label** into the same cell and title it **Detail**.
- Add a **right** Auto Layout constraint (of **16**) to the label and again, center **Vertically in Container**.
- Control-drag from the Address label to the Detail label and select **Horizontal Spacing** from the pop up. This will set up the current spacing between the two items as the default spacing. You don’t want that since you want the Detail label to display an address and so it should have room to breath.
- Select the Address label, switch to the Size inspector, select the trailing space constraint and edit the constraint so that the **Constant** is **>= 8** (instead of **=**). Note that you have to change the operator as well as the numeric constant value.

- Make sure the font of both labels is **System**, size **17**.
- Change the **Alignment** of the address detail label to right-aligned.

The detail label is special. Most likely the street address will be too long to fit in that small space. So, you'll configure this label to have a variable number of lines. This requires a bit of programming in the view controller to make it work, but you also have to set up this label's attributes properly.

- In the **Attributes inspector** for the address detail label, set **Lines** to **0** and **Line Break** to **Word Wrap**. When the number of lines is 0, the label will resize vertically to fit all the text that you put into it, which is exactly what you need.

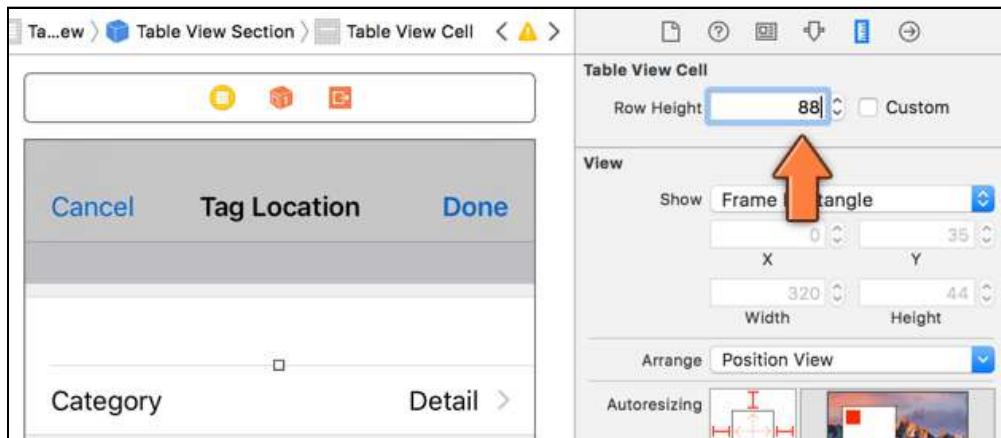


The address detail label can have multiple lines

The description cell

So far, you've left the cell at the top empty. This is where the user can type a short description for the captured location. Currently, there is not much room to type anything. So first, you'll make the cell larger.

- Click on the top cell to select it, then go into the **Size inspector** and type **88** into the **Row Height** field.



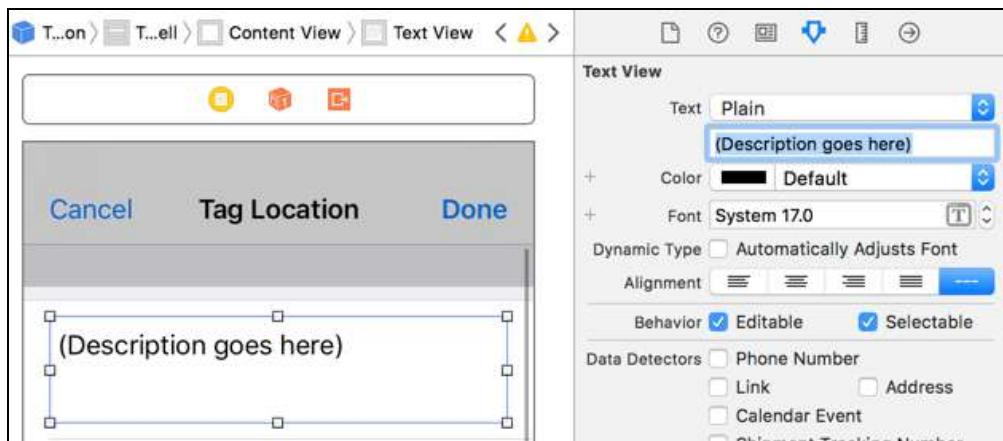
Changing the height of a row

You can also drag the cell to this new height by the sizing handle at its bottom, but I prefer to simply type in the new value.

The reason to use 88 is that quite a few iOS screen elements have a size of 44 points. The navigation bar is 44 points high, regular table view cells are 44 points high, and so on. Choosing 44 or a multiple of it keeps the UI looking balanced.

- Drag a **Text View** into the cell and add Auto Layout constraints for **left: 16, top: 10, right: 16, and bottom: 10**, with **Constrain to margins** unchecked.
- By default, Interface Builder puts a whole bunch of Latin placeholder text (Lorem ipsum dolor, etc) into the text view. Replace that text with **(Description goes here)**. The user will never see that text, but it's handy to remind yourself what this view is for.

- Set the font to **System**, size **17**.

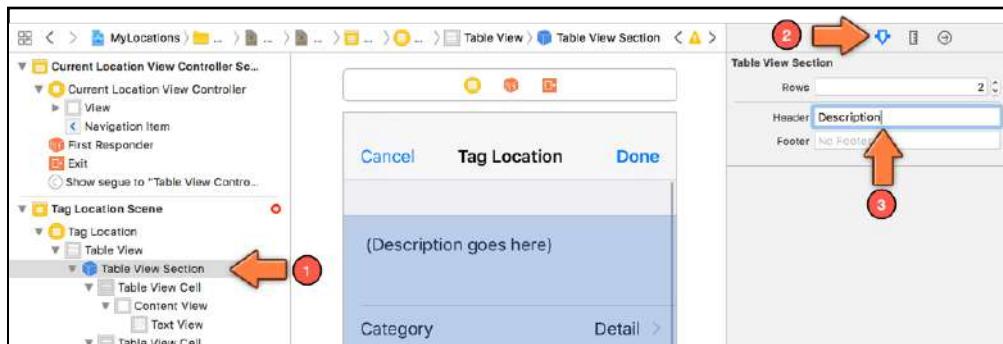


The attributes for the text view

One more thing to do, and then the layout is complete. Because the top cell doesn't have a label to describe what it does — and the text view will initially be empty as well — the user may not know what it is for.

There really isn't any room to add a label in front of the text view, as you've done for the other rows. So, let's add a header to the section. Table view sections can have a header and footer, and these can either be text or complete views with controls of their own.

- Select the top-most Table View Section and in its **Attributes inspector** type **Description** into the **Header** field:



Giving the section a header

That's the layout done. The Tag Location screen should look like this in the storyboard:



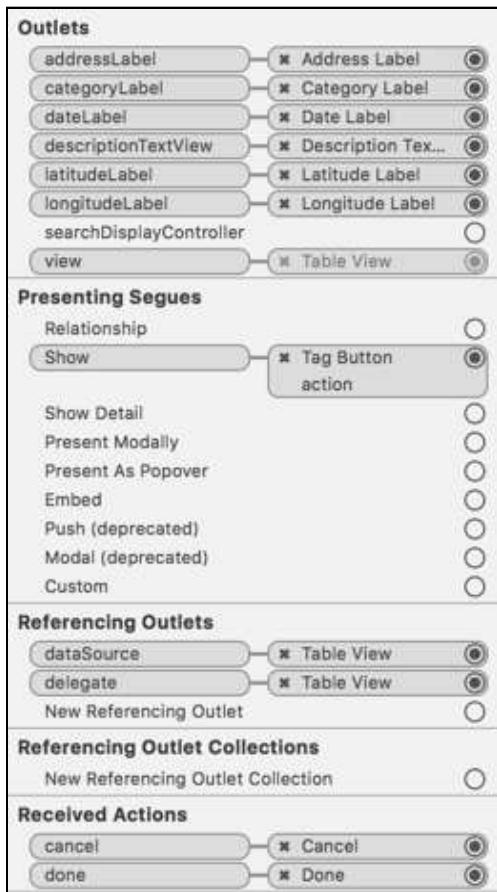
The finished design of the Tag Location screen

Now you can actually make the screen do stuff.

Connecting outlets

- ▶ Connect the Detail labels and the text view to their respective outlets. It should be obvious which one goes where. (Tip: Control-drag from the round yellow icon that represents the view controller to each of the labels. That's the quickest way.)

If you look at the **Connections inspector** for this view controller, you should see the following:



The connections of the Location Details View Controller

- Run the app to test whether everything works.

Of course, the screen still says “Detail” in the labels instead of the location’s actual coordinates and address because you haven’t passed in any data yet. Time to fix that, you reckon?

Displaying location info

- Add two new properties to **LocationDetailsViewController.swift**:

```
var coordinate = CLLocationCoordinate2D(latitude: 0,  
                                         longitude: 0)  
var placemark: CLPlacemark?
```

You've seen the `CLPlacemark` class before. It contains the address information — street name, city name, and so on — that you've obtained through reverse geocoding. This is an optional because there is no guarantee that the geocoder finds an address for the given coordinates.

`CLLocationCoordinate2D` is new. This contains the latitude and longitude from the `CLLocation` object that you received from the location manager. You only need the latitude and longitude, so there's no point in sending along the entire `CLLocation` object. The coordinate is not an optional, so you must give it an initial value.

Exercise: Why is coordinate not an optional?

Answer: You cannot tap the Tag Location button unless GPS coordinates have been found. So, you'll never open the `LocationDetailsViewController` without a valid set of coordinates.

During the segue from the Current Location screen to the Tag Location screen you will fill in these two properties, and then the Tag Location screen can put these values into its labels.

Xcode isn't happy with the two lines you just added. It complains about "Use of unresolved identifier `CLLocationCoordinate2D`" and "`CLPlacemark`." That means Xcode does not know anything about these types yet.

That's because they are part of the Core Location framework — and before you can use anything from a framework, you first need to import it.

- Add the following import to the file:

```
import CoreLocation
```

Now Xcode's error messages should disappear after a second or two. If they don't, use `⌘+B` to build the app again.



Structs

Unlike the objects you've seen before, `CLLocationCoordinate2D` is not a class, instead, it is a **struct** (short for structure).

Structs are like classes, but a little less powerful. They can have properties and methods, but unlike classes, they cannot inherit from one another.

The definition for `CLLocationCoordinate2D` is as follows:

```
struct CLLocationCoordinate2D {  
    var latitude: CLLocationDegrees  
    var longitude: CLLocationDegrees  
}
```

This struct has two fields, `latitude` and `longitude`. Both these fields have the data type `CLLocationDegrees`, which is a synonym for `Double`:

```
typealias CLLocationDegrees = Double
```

As you probably remember from before, the `Double` type is one of the primitive types built into Swift. It's like a `Float` but with higher precision.

Don't let these synonyms confuse you; `CLLocationCoordinate2D` is basically this:

```
struct CLLocationCoordinate2D {  
    var latitude: Double  
    var longitude: Double  
}
```

The reason the designers of Core Location used `CLLocationDegrees` instead of `Double` is that "CL Location Degrees" tells you what this type is intended for: it stores the degrees of a location from the Core Location framework.

Underneath the hood it's a `Double`, but as a user of Core Location all you need to care about when you want to store latitude or longitude is that you can use the `CLLocationDegrees` type. The name of the type adds meaning.

UIKit and other iOS frameworks also use structs regularly. Common examples are `CGPoint` and `CGRect`. In fact, `Array` and `Dictionary` are also structs.

Structs are more lightweight than classes. If you just need to pass around a set of values it's often easier to bundle them into a struct and pass that struct around, and that is exactly what Core Location does with coordinates.

Pass data to the details view

Back to the new properties that you just added to `LocationDetailsViewController`. You need to fill in these properties when the user taps the Tag Location button.

- Switch to `CurrentLocationViewController.swift` and add the following code:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "TagLocation" {
        let controller = segue.destination
            as! LocationDetailsViewController
        controller.coordinate = location!.coordinate
        controller.placemark = placemark
    }
}
```

You've seen how this works before. You use some casting magic to obtain the proper destination view controller and then set its properties. Now when the segue is performed, the coordinate and address are passed on to the Tag Location screen.

Because `location` is an optional, you need to unwrap it before you can access its `coordinate` property. It's perfectly safe to force unwrap at this point because the Tag Location button that triggers the segue won't be visible unless a location is found. At this point, `location` will never be `nil`.

The `placemark` variable is also an optional, but so is the `placemark` property on `LocationDetailsViewController`, so you don't need to do anything special here. You can always assign the value of one optional to another optional without problems.

Now that you have the values, you need to display them in the Tag Location screen.

Display information on the Tag Location screen

`viewDidLoad()` is a good place to display the passed in values on screen.

- Add the following code to `LocationDetailsViewController.swift`:

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

    descriptionTextView.text = ""
    categoryLabel.text = ""
```

```
latitudeLabel.text = String(format: "%.8f",
                             coordinate.latitude)
longitudeLabel.text = String(format: "%.8f",
                             coordinate.longitude)

if let placemark = placemark {
    addressLabel.text = string(from: placemark)
} else {
    addressLabel.text = "No Address Found"
}

dateLabel.text = format(date: Date())
}
```

This simply sets a value for every label. It uses two helper methods that you haven't defined yet: `string(from:)` to format the `CLPlacemark` object into a string, and `format(date:)` to do the same for a `Date` object.

► Add the `string(from:)` method:

```
// MARK:- Helper Methods
func string(from placemark: CLPlacemark) -> String {
    var text = ""

    if let s = placemark.subThoroughfare {
        text += s + " "
    }
    if let s = placemark.thoroughfare {
        text += s + ", "
    }
    if let s = placemark.locality {
        text += s + ", "
    }
    if let s = placemark.administrativeArea {
        text += s + " "
    }
    if let s = placemark.postalCode {
        text += s + ", "
    }
    if let s = placemark.country {
        text += s
    }
    return text
}
```

This is fairly straightforward. It is similar to how you formatted the placemark on the main screen, except that you also include the country here.



Note: You might have noticed the // MARK comments all over the previous sections of code in this chapter. You already know what the // MARK comment does. So, I'm not going to explain that again.

You can feel free to leave the comments out when you type in your own code, but it's recommended to organize code into identifiable sections as seen above so that you can navigate the code easily. It's totally up to you whether you use this, create an organization style of your own, or use no organization at all...

Date formatting

To format the date, you'll use a `NSDateFormatter` object. You've seen this class at work in the previous app. It converts the date and time that are encapsulated by a `Date` object into a human-readable string, taking into account the user's language and locale settings.

For *Checklists* you created a new instance of `NSDateFormatter` every time you wanted to convert a `Date` to a string. Unfortunately, `NSDateFormatter` is a relatively expensive object to create. In other words, it takes a while to initialize this object. If you do that many times over, then it may slow down your app (and drain the phone's battery faster).

It is better to create `NSDateFormatter` just once and then re-use that same object over and over. The trick is that you won't create the `NSDateFormatter` object until the app actually needs it. This principle is called **lazy loading** and it's a very important pattern for iOS apps — the work that you don't do won't cost any battery power.

In addition, you'll only ever create one instance of `NSDateFormatter`. The next time you need to use `NSDateFormatter` you won't make a new instance but re-use the existing one.

To pull this off you'll use a *private global* constant. That's a constant that lives outside of the `LocationDetailsViewController` class (global) but it is only visible inside the `LocationDetailsViewController.swift` file (private).

► Add the following to the top of `LocationDetailsViewController.swift`, in between the `import` and `class` lines:

```
private let dateFormatter: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateStyle = .medium
```

```
    formatter.timeStyle = .short
    return formatter
}()
```

What is going on here? You’re creating a new constant named `dateFormatter` of type `NSDateFormatter`, that much should be obvious. This constant is `private` so it cannot be used outside of this Swift file. (Remember the discussion about `private` and `public` attributes in the previous chapter?)

You’re also giving `dateFormatter` an initial value, but what follows the `=` is not an ordinary value — it looks like a bunch of source code in between `{ }` brackets. That looks like a closure, doesn’t it? That’s because it *is* a closure.

Normally, you’d create a new object like this:

```
private let dateFormatter = DateFormatter()
```

But to initialize the date formatter it’s not enough to just make an instance of `NSDateFormatter`, you also want to set the `dateStyle` and `timeStyle` properties of this instance.

To create the object and set its properties in one go, you can use a closure:

```
private let dateFormatter: DateFormatter = {
    // the code that sets up the DateFormatter object
    return formatter
}()
```

The closure contains the code that creates and initializes the new `NSDateFormatter` object, and then returns it. This returned value is what gets put into `dateFormatter`.

The trick to making this work is the `()` at the end. Closures are like functions, and to perform the code inside the closure you call it just like you’d call a function.

Note: If you leave out the `()`, Swift thinks you’re assigning the closure itself to `dateFormatter` — in other words, `dateFormatter` will contain a block of code, not an actual `NSDateFormatter` object. That’s not what you want.

Instead, you want to assign the *result* of that closure to `dateFormatter`. To make that happen, you use the `()` to perform or **evaluate** the closure — this runs the code inside the closure and returns a `NSDateFormatter` object.



Using a closure to create and configure an object all at once is a nifty trick; you can expect to see this often in Swift programs.

In Swift, globals are always created in a lazy fashion, which means the code that creates and sets up this `NSDateFormatter` object isn't performed until the very first time the `dateFormatter` global is used in the app.

That happens inside the new `format(date:)` method.

- Add the new method — this code goes inside the `class` (and would generally be put in the helper methods section created in the previous // MARK comment, for organizational purposes):

```
func format(date: Date) -> String {  
    return dateFormatter.string(from: date)  
}
```

How simple is that? It just asks the `NSDateFormatter` to turn the `Date` into a `String` and returns that.

Exercise: How can you verify that the date formatter is really only created once?

Answer: Add a `print()` just before the `return formatter` line in the closure. This `print()` text should appear only once in the Xcode Console.

- Run the app. Choose the Apple location from the Simulator's Debug menu. Wait until the street address is visible and then press the Tag Location button.

The coordinates, address and date are all filled in:

Latitude	37.33233141
Longitude	-122.03121860
Addr...	Infinite Loop, Cupertino, CA 95014, United States
Date	Jul 1, 2018 at 8:50 AM

The Address label doesn't fit well

The address seems to be having some trouble fitting in!



Content Compression Resistance

You earlier configured the label to fit multiple lines of text, but the problem is that the two labels in the address row don't know how to get along with each other — the detail label is too full of itself and encroaches on the space of the Address label.

The solution is simple enough — Content Compression Resistance. Quite a mouthful, and not very illuminating, right?

Let me try to shed some light.

- Select the **Address** label, switch to the **Size inspector** and scroll to the bottom. You should see a section named **Content Compression Resistance Priority**.

This section determines how easily the selected control allows other controls to push it (and its content) out of the way to present their own content. The higher the priority, the less likely this control is to be pushed out of the way. All controls have a horizontal and vertical content compression resistance value set and this is by default set to 750. All we need to do is increase the Address label's vertical content resistance priority so that it doesn't get pushed around.

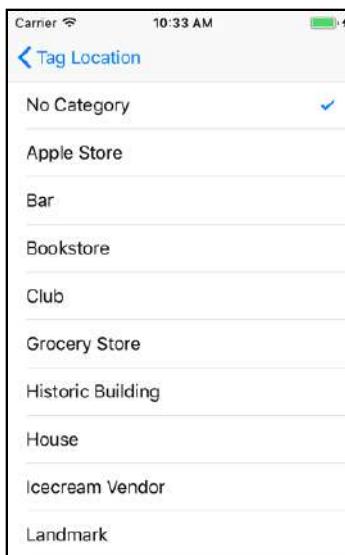
- Change the **Horizontal** value to **751**.
- Run the app. Now the reverse geocoded address should completely fit in the Address cell (even on larger screens). Try it out with a few different locations.

Latitude	37.33233141
Longitude	-122.03121860
Address	Infinite Loop, Cupertino, CA 95014, United States
Date	Jul 1, 2018 at 9:04 AM

The label is not cut off by the address

The category picker

When the user taps the Category cell, the app should show a list of category names:



The category picker

The view controller class

This is a new screen, so you need a new view controller. The way this works is very similar to the icon picker from *Checklists*. I'm just going to give you the source code and tell you how to hook it up.

- Add a new file to the project named **CategoryPickerController.swift**.
- Replace the contents of **CategoryPickerController.swift** with:

```
import UIKit

class CategoryPickerController: UITableViewController {
    var selectedCategoryName = ""

    let categories = [
        "No Category",
        "Apple Store",
        "Bar",
        "Bookstore",
        "Club",
        "Grocery Store",
        "Historic Building",
        "House",
        "Icecream Vendor",
        "Landmark"
    ]
}
```

```
"Historic Building",
"House",
"Icecream Vendor",
"Landmark",
"Park"]
```

```
var selectedIndexPath = IndexPath()
```

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

    for i in 0..

```
// MARK:- Table View Delegates
```



```
override func tableView(_ tableView: UITableView,
 numberOfRowsInSection section: Int) -> Int {
 return categories.count
}
```



```
override func tableView(_ tableView: UITableView,
 cellForRowAt indexPath: IndexPath) ->
 UITableViewCell {
 let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
 for: indexPath)

 let categoryName = categories[indexPath.row]
 cell.textLabel!.text = categoryName

 if categoryName == selectedCategoryName {
 cell.accessoryType = .checkmark
 } else {
 cell.accessoryType = .none
 }
 return cell
}
```



```
override func tableView(_ tableView: UITableView,
 didSelectRowAt indexPath: IndexPath) {
 if indexPath.row != selectedIndexPath.row {
 if let newCell = tableView.cellForRow(at: indexPath) {
 newCell.accessoryType = .checkmark
 }
 if let oldCell = tableView.cellForRow(
 at: selectedIndexPath) {
 oldCell.accessoryType = .none
 }
 }
}
```


```

```
    selectedIndexPath = indexPath
}
}
```

There's nothing special going on here. This is a table view controller that shows a list of category names. The table gets its rows from the `categories` array.

The only thing worth noting is the `selectedIndexPath` instance variable. When the screen opens, it shows a checkmark next to the currently selected category. This comes from the `selectedCategoryName` property, which is filled in when you segue to this screen.

When the user taps a row, you want to remove the checkmark from the previously selected row and put it in the new row.

In order to be able to do that, you need to know which row is the currently selected one. You can't use `selectedCategoryName` for this because that is a string, not a row number. Therefore, you first need to find the row number — or index-path — for the selected category name.

That happens in `viewDidLoad()`. You loop through the array of categories and compare the name of each category to `selectedCategoryName`. If they match, you create an index-path object and store it in the `selectedIndexPath` variable. Once a match is found, you can break out of the loop because there's no point in looping through the rest of the categories.

Now that you know the row number, you can remove the checkmark for this row in `tableView(_:didSelectRowAt:)` when another row gets tapped.

It's a bit of work for such a small feature, but in a good app it's the details that matter.

There are several different ways of looping through the contents of an array.

You've already seen `for...in`, which is used as follows:

```
for category in categories {
```

This puts the name of each category into a temporary constant named `category`.



However, in order to make the index-path object, you don't want the name of the category but the index of that category in the array. So you'll have to loop in a slightly different fashion:

```
for i in 0..    let category = categories[i]  
    ...  
}
```

Thanks to the half-open range operator `..<>`, `i` is a number that increments from 0 to `categories.count - 1`. This is a very common pattern for looping through an array if you want to have the index as well.

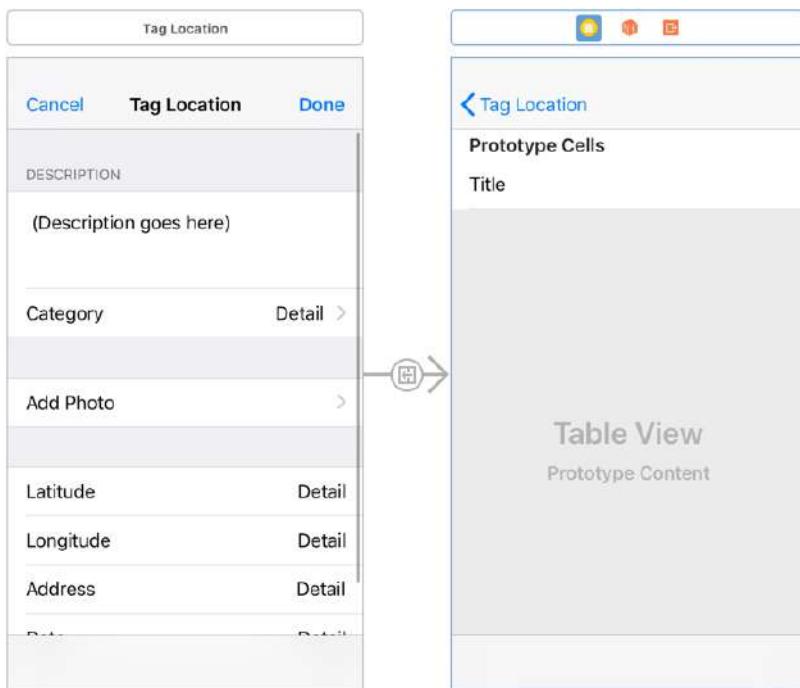
Another way to do this is to use the `enumerated()` method, for which you'll see an example when you get to the next app. As a quick preview, this is how you'd use it:

```
for (i, category) in categories.enumerated() {  
    ...  
}
```

The storyboard scene

- Open the storyboard and drag a new **Table View Controller** on to the canvas. Set its **Class** in the **Identity inspector** to **CategoryPickerController**.
- Change the **Style** of the prototype cell to **Basic**, and give it the re-use identifier **Cell**.
- **Control-drag** from the Category cell on the Location Details View Controller to this new view controller and choose **Selection Segue – Show**.
- Give the segue the identifier **PickCategory**.

The Category Picker View Controller now has a navigation bar at the top. You could change its title to “Choose Category,” but Apple recommends that you do not give view controllers a title if their purpose is obvious. This helps to keep the navigation bar uncluttered.



The category picker in the storyboard

That's enough for the storyboard. Now all that remains is to handle the segue.

The Segue

- Switch back to **LocationDetailsViewController.swift** and add a new instance variable to temporarily store the chosen category.

```
var categoryName = "No Category"
```

Initially you set the category name to “No Category,” which is the category at the top of the list in the category picker.

- Change `viewDidLoad()` to put `categoryName` into the label:

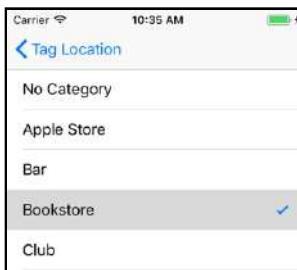
```
override func viewDidLoad() {  
    . . .  
    categoryLabel.text = categoryName // change this line  
    . . .
```

- Finally, add the segue handling code:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                     sender: Any?) {
    if segue.identifier == "PickCategory" {
        let controller = segue.destination as!
            CategoryPickerController
        controller.selectedCategoryName = categoryName
    }
}
```

This simply sets the `selectedCategoryName` property of the category picker. And with that, the app has categories.

- Run the app and play with the category picker.



Selecting a new category

Hmm, it doesn't seem to work very well. You can choose a category, but the screen doesn't close when you tap a row. When you press the back button, the category you picked isn't shown on the parent screen.

Exercise: Which piece of the puzzle is missing?

Answer: The `CategoryPickerController` currently does not have a way to communicate back to the `LocationDetailsViewController` about the user selection.

At this point you might be thinking, “Of course, dummy! You forgot to give the category picker a delegate protocol. That’s why it cannot send any messages to the other view controller.” (If so, awesome! You’re getting the hang of this.)

A delegate protocol is a fine solution indeed, but there’s a handy storyboarding feature that can accomplish the same thing with less work: **unwind segues**.

The unwind segue

In case you were wondering what the orange “Exit” icons in the storyboard are for, you now have your answer: unwind segues.



The Exit icon

Where a regular segue is used to open a new screen, an unwind segue closes the active screen. Sounds simple enough. However, making unwind segues is not very intuitive.

The orange Exit icons don’t appear to do anything. Try Control-dragging from the prototype cell to the Exit icon, for example. It won’t let you make a connection.

First, you have to add a special type of action method to the *destination* of the unwind segue.

► In **LocationDetailsViewController.swift**, add the following method:

```
@IBAction func categoryPickerDidPickCategory(  
    _ segue: UIStoryboardSegue) {  
    let controller = segue.source as! CategoryPickerController  
    categoryName = controller.selectedCategoryName  
    categoryLabel.text = categoryName  
}
```

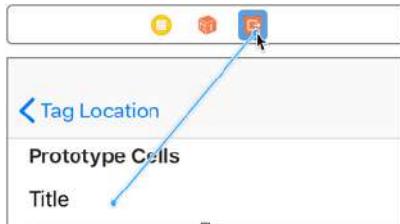
You can see that this is an action method because it has the `@IBAction` annotation. What’s different from a regular action method is the parameter, a `UIStoryboardSegue` object.

Normally, if an action method has a parameter, it points to the control that triggered the action, such as a button or slider. But in order to make an unwind segue, you need to define an action method that takes a `UIStoryboardSegue` parameter.

What happens inside the method is pretty straightforward. You look at the view controller that sent the segue (the source), which of course is the `CategoryPickerController`, and then read the value of its `selectedCategoryName` property. That property contains the category that the user picked.

Now, to use this new method in the storyboard...

- Open the storyboard. **Control-drag** from the prototype cell in the Category Picker scene to the Exit button. This time it allows you to make a connection:



Control-dragging to the Exit icon to make an unwind segue

From the pop-up choose **Selection Segue — categoryPickerDidPickCategory:**, the name of the unwind action method you just added.



The pop-up lists the unwind action methods

If Interface Builder doesn't let you make a connection, then make sure you're really Control-dragging from the Cell, not from its Content View or the label.

Now when you tap a cell in the category picker, the screen closes and this new method is called.

- Run the app to try it out.

That was easy! Well, not quite. Unfortunately, the chosen category is ignored...

That's because `categoryPickerDidPickCategory()` looks at the `selectedCategoryName` property, but that property isn't set anywhere in your code yet.

You need some kind of mechanism that is invoked when the unwind segue is triggered, at which point you can fill in the `selectedCategoryName` based on the row that was tapped.

What might such a mechanism be called? `prepare(for:sender:)`, of course! This works for segues in both directions.

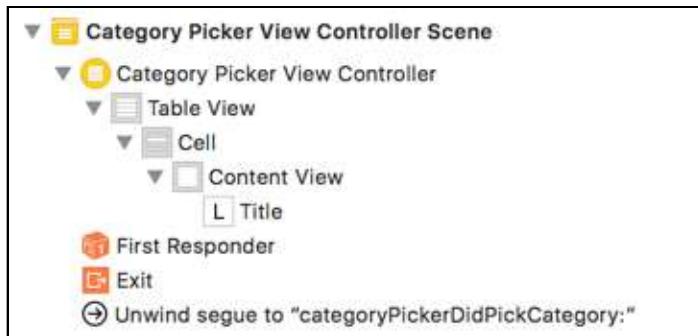
- Add the following method to **CategoryPickerController.swift**:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "PickedCategory" {
        let cell = sender as! UITableViewCell
        if let indexPath = tableView.indexPath(for: cell) {
            selectedCategoryName = categories[indexPath.row]
        }
    }
}
```

This looks at the selected index-path and puts the corresponding category name into the `selectedCategoryName` property.

This logic assumes the unwind segue is named “PickedCategory,” so you still have to set an identifier on the unwind segue.

Unfortunately, there is no visual representation of that unwind segue in the storyboard. There is no nice, big arrow that you can click on. To select the unwind segue you have to locate it in the Document Outline:



You can find unwind segues in the Document Outline

- Select the unwind segue and go to the **Attributes inspector**. Give it the identifier **PickedCategory**.

- Run the app. Now the category picker should work properly. As soon as you tap the name of a category, the screen closes and the new category name is displayed.

Unwind segues are pretty cool and are often easier than using a delegate protocol, especially for simple picker screens such as this one.

You can find the project files for this chapter under **30 - Tag Location Screen** in the Source Code folder.

Chapter 31: Adding Polish

Eli Ganim

Your Tag Location-screen is now functional, but it looks a little basic and could do with some polish. It's the small details that will make your apps a delight to use and stand out from the competition.

In this chapter, you will learn the following:

- How to improve the user experience by adding tiny tweaks to your app which gives it some polish.
- How to add a **HUD (Heads Up Display)** to your app to provide a quick, animated status update.
- How to continue the navigation flow after displaying the HUD.



Improving the user experience

Take a look at the design of the cell with the Description text view:

There is a margin between the text view and the cell border. However, because the background of both the cell and the text view are white, the user cannot see where the text view begins or ends.

It is possible to tap on the cell but be just outside the text view area. That is annoying when you want to start typing: You think that you're tapping in the text view, but the keyboard doesn't appear.

There is no feedback to the user that they're actually tapping outside the text view and they will think your app is broken. In my opinion, deservedly so.

Keyboard activation for cells

You'll have to make the app a little more forgiving. When the user taps anywhere inside that first cell, the text view should activate, even if the tap wasn't on the text view itself.

- Add the following table view delegate methods to **LocationDetailsViewController.swift**:

```
// MARK:- Table View Delegates
override func tableView(_ tableView: UITableView,
    willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    if indexPath.section == 0 || indexPath.section == 1 {
        return indexPath
    } else {
        return nil
    }
}

override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {
    if indexPath.section == 0 && indexPath.row == 0 {
        descriptionTextView.becomeFirstResponder()
    }
}
```

The `tableView(_:willSelectRowAt:)` method limits taps to just the cells from the first two sections. Recall that `||` means *or*. So, if the section number equals 0 *or* when it equals 1, you accept the tap on the cell. The third section only has read-only labels — it doesn't need to allow taps.

The `tableView(_:didSelectRowAt:)` method handles the actual taps on the rows. You don't need to respond to taps on the Category or Add Photo rows as these cells are connected to segues.

But if the user taps on the first row of the first section — the row with the description text view — then you will give the input focus to the text view. Here you use `&&`, meaning *and*, to make sure that the tap is in the first section *and* also on the first row of that section.

► Try it out. Run the app and click or tap somewhere along the edges of the first cell. Any tap inside that first cell should now make the text view active and bring up the keyboard. Remember that on the simulator, you may need to press `⌘+K` to make the keyboard visible.

Anything you can do to make screens less frustrating to use is worth putting in the effort!

Speaking of the text view, once you've activated it, there's no way to get rid of the keyboard! And because the keyboard takes up half of the screen, that can be a bit annoying.

Deactivating the keyboard

It would be nice if the keyboard disappeared after you tapped anywhere else on the screen. As it happens, that is not so hard to implement.

- Add the following to the end of `viewDidLoad()` in `LocationDetailsViewController.swift`:

```
// Hide keyboard
let gestureRecognizer = UITapGestureRecognizer(target: self,
                                             action: #selector(hideKeyboard))
gestureRecognizer.cancelsTouchesInView = false
tableView.addGestureRecognizer(gestureRecognizer)
```

A **gesture recognizer** is a very handy object that can recognize touch-based actions like taps, swipes, pans and pinches. You simply create the gesture recognizer object, give it a method to call when that particular gesture has been observed to take place and add the recognizer object to a view.

You're using a `UITapGestureRecognizer`, which as the name implies, recognizes simple taps.

Notice the `#selector()` keyword again:

```
    . . . target: self, action: #selector(hideKeyboard)) . . .
```

You use this syntax to tell the `UITapGestureRecognizer` that it should call the method named by `#selector()` whenever the gesture happens.

This pattern is known as **target-action** and you've already used it whenever you've connected `UIButtons`, `UIBarButtonItem`s and other controls to action methods.

The “target” is the object receiving the message, which is often `self`, and “action” is the message to send.

Here, you've chosen the message `hideKeyboard` to be sent when a tap is recognized anywhere in the table view. So, you have to implement the method and respond to that message. Also, remember that selectors have their roots in Objective-C. Therefore, any method which is called via a selector has to be accessible from Objective-C.

- Add the `hideKeyboard()` method to `LocationDetailsViewController.swift`:

```
@objc func hideKeyboard(_ gestureRecognizer:
                         UIGestureRecognizer) {
    let point = gestureRecognizer.location(in: tableView)
    let indexPath = tableView.indexPathForRow(at: point)

    if indexPath != nil && indexPath!.section == 0
        && indexPath!.row == 0 {
        return
    }
```

```
    descriptionTextView.resignFirstResponder()  
}
```

Whenever the user taps somewhere in the table view, the gesture recognizer calls this method. Conveniently, it also passes a reference to itself as a parameter, which lets you ask `gestureRecognizer` where the tap happened.

The `gestureRecognizer.location(in:)` method returns a `CGPoint` value indicating the tap position. `CGPoint` is a common struct that you see all the time in UIKit. It contains two fields, `x` and `y`, that describe a position on-screen.

Using this `CGPoint`, you ask the table view which index-path is currently displayed at that position. This is important because you obviously don't want to hide the keyboard if the user tapped in the row with the text view! If the user tapped anywhere else, you hide the keyboard.

Exercise: Does the logic in the `if` statement make sense to you? Explain how this works.

Answer: It is possible that the user tapped inside the table view, but not on a cell. For example, somewhere in between two sections or on the section header. In that case, `indexPath` will be `nil`, making this an optional of type `IndexPath?`. To use an optional, you need to unwrap it somehow, either with `if let` or with `!`.

You only want to hide the keyboard if the index-path for the tap is not section 0, row 0, which is the cell with the text view. If the user did tap that particular cell, you bail out of `hideKeyboard()` with the `return` statement before the code reaches the call to `resignFirstResponder()`.

Note: You don't want to force unwrap an optional if there's a chance it might be `nil` or you risk crashing the app. Force unwrapping `indexPath!.section` and `indexPath!.row` may look dangerous here, but it is guaranteed to work thanks to the **short-circuiting** behavior of the `&&` operator.

If `indexPath` equals `nil`, then everything after the first `&&` is simply ignored. The condition can never become true anymore if one of the terms is false. So, when the app gets to look at `indexPath!.section`, you know that the value of `indexPath` is not `nil` at that point.

An alternative way to write this logic is:

```
if indexPath == nil ||  
    !(indexPath!.section == 0 && indexPath!.row == 0) {  
    descriptionTextView.resignFirstResponder()  
}
```

Can you wrap your head around that? Here, the `if` statement checks for the exact opposite. The `&&` and `||` operators are each other's opposite in Boolean logic and you can often flip the meaning of a condition around by turning `&&` into `||` by introducing the `!` not operator. You don't need to worry about this so early on in your programming career, but at some point, you'll have to learn these rules of Boolean logic. They can be mind-benders!

Of course, you can also use `if let` to safely unwrap `indexPath`. So a third — but more verbose — way to write the `if` statement is as follows:

```
if let indexPath = indexPath {  
    if indexPath.section != 0 && indexPath.row != 0 {  
        descriptionTextView.resignFirstResponder()  
    }  
} else {  
    descriptionTextView.resignFirstResponder()  
}
```

This gives you a brief glimpse of the various ways you can write the conditions in `if` statements. There's often more than one way to do something in Swift. So, choose whatever approach you find easiest to understand.

► Run the app. Tap in the text view to bring up the keyboard. If the keyboard doesn't come up, press `⌘+K`. Tap anywhere else in the table view to hide the keyboard again.

The table view can also automatically dismiss the keyboard when the user starts scrolling. You can enable this in the storyboard.

► Open the storyboard and select the table view in the Tag Location scene. In the **Attributes inspector** change the **Keyboard** option to **Dismiss on drag**. Now, scrolling should also hide the keyboard.

If this doesn't work for you on the simulator, try it on a real device. The keyboard in the simulator can be a bit wonky.

► Also, try the **Dismiss interactively** option. Which one do you like best?

The HUD

There is one more improvement to make to this screen, just to add a little spice. When you tap the **Done** button to close the screen, the app will show a quick animation to let you know it successfully saved the location:

This type of overlay graphic is often called a **HUD**, for **Heads-Up Display**. Apps aren't quite fighter jets, but HUDs are often used to display a progress bar or spinner while files are downloading or another long-lasting task is taking place.

You'll show your own HUD view for a brief second before the screen closes. It adds an extra bit of liveliness to the app. If you're wondering how you can display anything on top of a table, this HUD is simply a `UIView` subclass. You can add views on top of other views. In fact, that's what you've been doing all along.

The labels are views that are added on top of the cells, which are also views. The cells themselves are added on top of the table view, and the table view, in turn, is added on top of the navigation controller's content view.

So far, when you've made your own objects, they have always been view controllers or data model objects, but it's also possible to make your own views.

Often, using the standard buttons and labels is sufficient. But when you want to do something that is not available as a standard view, you can always make your own. You either subclass `UIView` or `UIControl` and do your own drawing. That's what you're going to do for the HUD view as well.

Creating the HUD view

- Add a new file to the project using the **Swift File** template. Name it **HudView**.

Let's build a minimal version of this class just so that you can get something on the screen. When that works, you'll make it look fancy.

- Replace the contents of **HudView.swift** with the following:

```
import UIKit

class HudView: UIView {
    var text = ""

    class func hud(inView view: UIView,
                   animated: Bool) -> HudView {
        let hudView = HudView(frame: view.bounds)
        hudView.isOpaque = false

        view.addSubview(hudView)
        view.isUserInteractionEnabled = false

        hudView.backgroundColor = UIColor(red: 1, green: 0, blue: 0,
                                         alpha: 0.5)
        return hudView
    }
}
```

The `hud(inView, animated)` method is known as a **convenience constructor**. It creates and returns a new `HudView` instance.



Normally, you would create a new HudView object by writing:

```
let hudView = HudView()
```

But using the convenience constructor you'd write:

```
let hudView = HudView.hud(inView: parentView, animated: true)
```

A convenience constructor is generally a **class method**, i.e. a method that works on the class as a whole and not on any particular instance. You can tell because its declaration begins with `class func` instead of just `func`.

When you call `HudView.hud(inView: parentView, animated: true)` you don't have an instance of `HudView` yet. The whole purpose of this method is to create an instance of the HUD view for you — so that you don't have to do that yourself — and to place it on top of another view.

You can see that making an instance is actually the first thing this method does:

```
class func hud(inView view: UIView,
               animated: Bool) -> HudView {
    let hudView = HudView(frame: view.bounds)
    .
    .
    return hudView
}
```

It calls `HudView()`, or actually `HudView(frame:)`, which is an `init` method inherited from `UIView`. At the end of the method, the new instance is returned to the caller.

So why use this convenience constructor? As the name implies, for convenience.

Since there are several steps to set up the view, putting them in the convenience constructor frees you from having to worry about any of that.

One of these additional steps is that this method adds the new `HudView` object as a subview on top of the “parent” view object. This is the navigation controller's view, so the HUD will cover the entire screen.

It also sets the parent view's `isUserInteractionEnabled` property to `false`. While the HUD is showing, you don't want the user to interact with the screen anymore. The user has already tapped the **Done** button and the screen is in the process of closing.

Most users will leave the screen alone at this point, but there's always some joker who wants to try and break things. By setting `isUserInteractionEnabled` to `false`, the view swallows any touches and all the underlying views become unresponsive.



Just for testing, you set the background color of the HUD to 50% transparent red. That way you can see if it covers the entire screen.

Using the HUD view

Let's add the code to call this funky new HUD so that you can see it in action.

- ▶ Change the `done()` method in `LocationDetailsViewController.swift` to:

```
@IBAction func done() {
    let hudView = HudView.hud(inView: navigationController!.view,
                             animated: true)
    hudView.text = "Tagged"
}
```

This creates a `HudView` object and adds it to the navigation controller's view with an animation. You also set the `text` property on the new object.

Previously, `done()` sent you back to the previous view controller. For testing purposes, you're not going to do that anymore. You want to have enough time to see what the `HudView` looks like as you build it step-by-step. If you immediately close the screen after showing the HUD, it will be hard to see what's going on — unless you can slow down time somehow... You'll put back the code that closes the screen later.

- ▶ Run the app. When you press the **Done** button, the screen will look like this:

The app is now totally unresponsive because user interaction is disabled.

When you're working with views, it's a good idea to set the background color to a bright color such as red or blue, so you can see exactly how big a given view is.

Did you, upon looking at the HUD activation code, think: "Hey, how come we are using the navigation controller's view instead of the view from `LocationDetailsViewController`?" If you did, good on you! It shows that you are starting to understand the composition of view controllers and views and thinking about how they work.

The answer is simple enough to figure out. Just try it and see what happens. Change the `HudView` creation line in `done()` to the following:

```
let hudView = HudView.hud(inView: view, animated: true)
```

Here, instead of the navigation controller's content view, you use the current view controller's view as the parent for the HUD.



- Run the app and try the **Done** button. You should get a screen like this:

Do you see what happened?

The HUD now only covers the screen area for the `LocationDetailsViewController`'s view — it does not cover the navigation bar. And you know what that means, right? The user can tap on the **Cancel** or **Done** buttons and have them respond — even if the rest of the screen has user interactions disabled. That can be a problem in certain situations.

Revert your code back to using the navigation controller's view before you forget.

Let's get the HUD view to actually display something on-screen instead of the red background.

Drawing the HUD view

- Remove the `backgroundColor` line from the `hud(inView:animated:)` method.
► Add the following method to `HudView.swift`:

```
override func draw(_ rect: CGRect) {
    let boxWidth: CGFloat = 96
    let boxHeight: CGFloat = 96

    let boxRect = CGRect(
        x: round((bounds.size.width - boxWidth) / 2),
        y: round((bounds.size.height - boxHeight) / 2),
        width: boxWidth,
        height: boxHeight)

    let roundedRect = UIBezierPath(roundedRect: boxRect,
                                    cornerRadius: 10)
    UIColor(white: 0.3, alpha: 0.8).setFill()
    roundedRect.fill()
}
```

The `draw()` method is invoked whenever UIKit wants your view to redraw itself.

Recall that everything in iOS is event-driven. The view doesn't draw anything on-screen unless UIKit asks it to draw itself. That means you should never call `draw()` yourself.

Instead, if you want a view to redraw, you should send it the `setNeedsDisplay()` message. UIKit will then trigger a `draw()` when it is ready to perform the drawing. This may seem strange if you're coming from another platform. You may be used to redrawing the screen whenever you feel like it. On iOS, however, UIKit is in charge of

who gets to draw when.



The above code draws a filled rectangle with rounded corners in the center of the screen. The rectangle is 96 by 96 points (so it's really a square):

```
let boxWidth: CGFloat = 96
let boxHeight: CGFloat = 96
```

This declares two constants you'll be using in the calculations that follow. You're using constants because it's clearer to refer to the symbolic name `boxWidth` than the number 96. That number doesn't mean much by itself, but "box width" is a pretty clear description of its purpose.

Additionally, if you were to later decide to change the size of the HUD box, you only have one place in your code where you need to change the width or the height, instead of going through all of your code trying to figure out where else you had the width or the height value as a number.

Note that you force the type of these constants to be `CGFloat`, which is the type used by UIKit to represent decimal numbers. When working with UIKit or Core Graphics (CG, get it?) you use `CGFloat` instead of the regular `Float` or `Double`.

```
let boxRect = CGRect(
    x: round((bounds.size.width - boxWidth) / 2),
    y: round((bounds.size.height - boxHeight) / 2),
    width: boxWidth,
    height: boxHeight)
```

There's `CGRect` again, the struct that represents a rectangle. You use it to calculate the position for the HUD. The HUD rectangle should be centered horizontally and vertically on the screen. The size of the screen is given by `bounds.size`. This is the size of `HudView` itself, which spans the entire screen.

The above calculation uses the `round()` function to make sure the rectangle doesn't end up on fractional pixel boundaries because that makes the image look fuzzy.

```
let roundedRect = UIBezierPath(roundedRect: boxRect,
    cornerRadius: 10)
UIColor(white: 0.3, alpha: 0.8).setFill()
roundedRect.fill()
```

`UIBezierPath` is a very handy object for drawing rectangles with rounded corners. You just tell it how large the rectangle is and how round the corners should be. Then you fill the rectangle with an 80% opaque dark gray color.

- Run the app. The result should look like this:

There are two more things to add to the HUD, a checkmark and a text label. The checkmark is an image.

Displaying the HUD checkmark

- The Resources folder for the book has two files in the **Hud Images** folder: **Checkmark@2x.png** and **Checkmark@3x.png**. Add these files to the asset catalog, **Assets.xcassets**.

You can do this with the + button or simply drag them from Finder to the Xcode window with the asset catalog open.

- Add the following code to the end of `draw()`:

```
// Draw checkmark
if let image = UIImage(named: "Checkmark") {
    let imagePoint = CGPoint(
        x: center.x - round(image.size.width / 2),
        y: center.y - round(image.size.height / 2) - boxHeight / 8)
    image.draw(at: imagePoint)
}
```

This loads the checkmark image into a `UIImage` object. Then it calculates the position for that image based on the center coordinate of the HUD view (`center`) and the dimensions of the image (`image.size`).

Finally, it draws the image at that position.

- Run the app to see the HUD view with the image:

Note: If you don't see the checkmark when you run the app and, if you did change the `done()` method to use the view controller's view instead of the navigation controller's content view earlier, make sure that you reverted the code back.

The position calculations are based on the HUD view stretching up to the navigation bar and, if the view size is different, the checkmark will be placed a little above the rounded square. Since the background is mostly white outside the square and the checkmark is white, too, you might not even notice it when it is drawn outside the rounded square.

Failable initializers

To create the `UIImage`, you used `if let` to unwrap the resulting object. That's because `UIImage(named:)` is a *failable* initializer.

It is possible that loading the image fails. This could be for one of many different reasons such as there being no image with the specified name, or the file not containing a valid image. You can't fool `UIImage` into loading something that isn't an image!

That's why `UIImage`'s `init(named:)` method is really defined as `init?(named:)`. The question mark indicates that this method returns an optional. If there was a problem loading the image, it returns `nil` instead of a brand new `UIImage` object.

You'll see these failable initializers throughout the iOS frameworks. One that you have encountered before is `init?(coder:)`. Whenever it is possible that creating a new object will fail, the responsible `init` method will return an optional that you need to unwrap before you can use it.

Displaying the HUD text

Usually, to display text in your own view, you'd add a `UILabel` object as a subview and let `UILabel` do all of the hard work. However, for a view as simple as this, you can also do your own text drawing.

- Add the following code to the end of `draw()` to complete the method:

```
// Draw the text
let attribs = [
```

```
NSAttributedString.Key.font: UIFont.systemFont(ofSize: 16),  
NSAttributedString.Key.foregroundColor: UIColor.white ]  
  
let textSize = text.size(withAttributes: attrs)  
  
let textPoint = CGPoint(  
    x: center.x - round(textSize.width / 2),  
    y: center.y - round(textSize.height / 2) + boxHeight / 4)  
  
text.draw(at: textPoint, withAttributes: attrs)
```

When drawing text, you first need to know how big the text is so you can figure out where to position it. `String` has a bunch of handy methods for doing both.

First, set up a dictionary of attributes for the text that you want to draw, such as the font to be used, the text color, etc. Here, you'll use a white system font of size 16.

You use these attributes and the string from the `text` property to calculate how wide and tall the text will be.

The result ends up in the `textSize` constant, which is of type `CGSize`. As you'll notice, `CGPoint`, `CGSize` and `CGRect` are types you use a lot when making your own views.

Finally, you calculate where to draw the text (`textPoint`), and then draw it. Quite simple, really.

- Run the app to try it out. Lookin' good!
- Make sure to test the HUD on different Simulators. No matter the device dimensions, the HUD should always appear centered on the screen.

OK, you've now got a rounded box with a checkmark, but it's still far from spectacular. Time to liven it up a little with some animation!

Adding some animation

You've already seen a bit about animations before — they're really easy to add.

- Add the following method to **HudView.swift**:

```
// MARK:- Public methods
func show(animated: Bool) {
    if animated {
        // 1
        alpha = 0
        transform = CGAffineTransform(scaleX: 1.3, y: 1.3)
        // 2
        UIView.animate(withDuration: 0.3, animations: {
            // 3
            self.alpha = 1
            self.transform = CGAffineTransform.identity
        })
    }
}
```

For the *Bull's Eye* app, you made a crossfade animation using the Core Animation framework. `UIView`, however, has its own animation mechanism. It still uses Core Animation behind the scenes, but it's a little more convenient to use.

The standard steps for doing `UIView`-based animations are as follows:

1. Set up the initial state of the view before the animation starts. Here, you set `alpha` to 0, making the view fully transparent. You also set the `transform` to a scale factor of 1.3. We're not going to go into depth on transforms here but this means the view is initially scaled up to be larger than it normally would be.
2. Call `UIView.animate(withDuration:animations:)` to set up an animation. You pass the method a closure that describes what happens as part of the animation. Recall that a closure is a piece of inline code that is not executed right away. UIKit will animate the properties that you change inside the closure from their initial state to the final state.
3. Inside the closure, set up the state of the view as it should be after the animation completes. You set `alpha` to 1, which means the `HudView` is now fully opaque. You also set the `transform` to the “identity” transform, restoring the scale back to normal. Because this code is part of a closure, you need to use `self` to refer to the `HudView` instance and its properties. That's the rule for closures.

The HUD view will quickly fade in as it goes from fully transparent to fully opaque, and it will scale down from 1.3 times its original size to its regular width and height.

This is only a simple animation but it looks quite smart.

- Change the `hud(inView:animated:)` method to call `show(animated:)` just before it returns:

```
class func hud(inView view: UIView, animated: Bool) -> HudView {  
    . . .  
    hudView.show(animated: animated) // Add this  
    return hudView  
}
```

- Run the app and marvel at the magic of `UIView` animation.

Improving the animation

You can actually do one better. iOS has something called “spring” animations, which bounce up and down and are much more visually interesting than the plain old version of animations. Using them is very simple.



- Replace the `UIView.animate(withDuration:animations:)` code in `show(animated:)` with the following:

```
UIView.animate(withDuration: 0.3, delay: 0,
    usingSpringWithDamping: 0.7, initialSpringVelocity: 0.5,
    options: [], animations: {
    self.alpha = 1
    self.transform = CGAffineTransform.identity
}, completion: nil)
```

The code in the closure is still the same: It sets alpha to 1 and restores the identity transform. However, this new animation method has a lot more options. Feel free to play with these options to see what they do.

- Run the app and watch it bounce. Actually, the effect is very subtle, but subtle is good when it comes to user interfaces. You don't want your users to get seasick from using the app!

Handling the navigation

Back to `LocationDetailsViewController`. You still need to close the screen when the user taps `Done`.

There's a challenge here: You don't want to dismiss the screen right away. It won't look very good if the screen closes before the HUD is finished animating. You didn't spend all that time writing `HudView` for nothing — you want to give your users a chance to see it.

You are going to use the `Grand Central Dispatch` framework, or `GCD` here. `GCD` is a very handy but somewhat low-level library for handling asynchronous tasks. Telling the app to wait a few seconds before executing some code is a perfect example of an `async` task.

- Add these lines to the bottom of the `done()` action method:

```
let delayInSeconds = 0.6
DispatchQueue.main.asyncAfter(deadline: .now() + delayInSeconds,
    execute: {
    self.navigationController?.popViewController(animated: true)
})
```

Believe it or not, these mysterious incantations tell the app to close the Tag Location-screen after 0.6 seconds.

The magic happens in `DispatchQueue.main.asyncAfter()`. This function takes a closure as its final parameter. Inside that closure, you tell the navigation controller to go back to the previous view controller in the navigation stack. This doesn't happen right away, though. That's the exciting thing about closures: Even though this code sits side-by-side with all of the other code in the method, everything inside the closure is ignored for now and kept for a later time.

`DispatchQueue.main.asyncAfter()` uses the time given by `.now() + delayInSeconds` to schedule the closure for some point in the future. Until then, the app just sits there twiddling its thumbs. By the way, `.now()` is a shortcut for `DispatchTime.now()`. Swift's type inference already knows that the type of the `when:` parameter is always a `DispatchTime` object, so you don't have to mention `DispatchTime` explicitly.

After 0.6 seconds, the code from the closure runs and the screen closes.

Note: It takes time finding a suitable value. The HUD view takes 0.3 seconds to fully fade in and then you wait another 0.3 seconds before the screen disappears. You don't want to close the screen too quickly or the effect from showing the HUD is lost, but it shouldn't take too long either, or it will annoy the user. Animations are cool but they shouldn't make the app more frustrating to use!

- ▶ Run the app. Press the Done button and watch how the screen disappears. This looks pretty smooth!

But wait... the HUD never goes away after the Tag Location-screen closes! It still is there after you navigate back to the parent view. This is not good...

Exercise: Can you explain why this happens?

The reason is simple. You added the HUD to the navigation controller's content view, not the Tag Location-screen's view. So, even though you've dismissed the Tag Location-screen, you still have the HUD displaying because the navigation controller itself is still in existence.

So what do you think you should do to hide the HUD? Remove it from view, of course!

- Add the following method to **HudView.swift**:

```
func hide() {
    superview?.isUserInteractionEnabled = true
    removeFromSuperview()
}
```

This method is rather simple. Remember how you disabled user-interactions when showing the HUD? You first re-enable user-interactions and then remove the HudView instance from its parent view. The only new thing might be `superview` and that's a reference to a view's parent view — all `UIView` objects and sub-classes of `UIView` have a `superview` property which identifies the view's parent.

Of course, if you wanted, you could have made the method a bit more complex and interesting by adding some animation to the removal of the HUD. Basically, you'd set up the animation to reverse what you did when you showed the view. It is left for you as an exercise!

Now, you need to call this new method to hide the HUD before you exit the Tag Location-screen.

- Modify the `DispatchQueue.main.asyncAfter` closure for `done()` in **LocationDetailsViewController.swift**:

```
DispatchQueue.main.asyncAfter(deadline: .now() + delayInSeconds,
                               execute: {
    hudView.hide() // Add this line
    self.navigationController?.popViewController(animated: true)
})
```

- Run the app. Press the **Done** button and check if the HUD disappears when the Tag Location-screen goes away.

Cleaning up the code

GCD code can be a bit messy. So let's clean up the code and make it easier to understand.

- Add a new file to the project using the **Swift File** template. Name the file **Functions.swift**.
- Replace the contents of the new file with:

```
import Foundation

func afterDelay(_ seconds: Double, run: @escaping () -> Void) {
```



```
DispatchQueue.main.asyncAfter(deadline: .now() + seconds,  
                           execute: run)  
}
```

That looks very much like the code you just added to `done()`, except it now lives in its own function: `afterDelay()`. This is a **free function**, not a method inside an object. So, it can be used from anywhere in your code.

Take a good look at `afterDelay()`'s second parameter, the one named `run`. Its type is `() -> Void`. That's not some weird emoticon, it is Swift notation for a parameter that takes a closure with no arguments and no return value.

The type for a closure generally looks like this:

```
(parameter list) -> return type
```

In this case, both the parameter list and the return value are empty, `()` and `Void`. This can also be written as `Void -> Void`, or even `() -> ()`, but `() -> Void` is preferred because it looks like a function declaration.

So, whenever you see a `->` in the type annotation for a parameter, you know that parameter is a closure.

`afterDelay()` simply passes this closure along to `DispatchQueue.main.asyncAfter()`.

The annotation `@escaping` is necessary for closures that are not performed immediately. This is so that Swift knows that it should hold on to this closure for a while.

You may be wondering why you're going through all this trouble. No fear! The reason will become apparent after you've made the following change.

► Go back to **LocationDetailsViewController.swift** and change `done()` as follows:

```
@IBAction func done() {  
    let hudView = HudView.hud(inView: navigationController!.view,  
                             animated: true)  
    hudView.text = "Tagged"  
    afterDelay(0.6, run: {  
        hudView.hide()  
        self.navigationController?.popViewController(animated: true)  
    })  
}
```

Now that's the power of Swift! It only takes one look at this code to immediately understand what it does. After a delay, some code is executed.



By moving the nasty GCD stuff into a new function, `afterDelay()`, you have added a new level of **abstraction** to your code that makes it much easier to follow. Writing good programs is all about finding the right abstractions.

Note: Because the code referring to the navigation controller sits in a closure, it needs to use `self`. Inside closures, you always need to use `self` explicitly. But, you didn't need to use `self` for the line referring to the `hudView` since it is a local variable that would be in existence only within the scope of the `done()` method.

You can make the code even more concise. Change the code to:

```
afterDelay(0.6) {  
    hudView.hide()  
    self.navigationController?.popViewController(animated: true)  
}
```

Now the closure sits *outside* of the call to `afterDelay()`.

Swift has a handy rule that says you can put a closure outside a function call if it's the last parameter of the function. This is known as **trailing closure syntax**. You will usually see closures being used in this manner because it reads (and looks) better.

► Run the app again to make sure the timing still works. Boo-yah!

You can find the project files for this chapter under **31 – Adding Polish** in the Source Code folder.

32

Chapter 32: Saving Locations

By Eli Ganim

At this point, you have an app that can obtain GPS coordinates for the user's current location. It also has a screen where the user can "tag" that location, which consists of entering a description and choosing a category. Later on, you'll also allow the user to pick a photo.

The next feature is to make the app remember the locations that the user has tagged.

This chapter covers the following:

- **Core Data overview:** A brief overview of what Core Data is and how it works.
- **Add Core Data:** Add the Core Data framework to the app and use it.
- **The data store:** Initializing the data store used by Core Data.
- **Pass the context:** How to pass the context object used to access Core Data between view controllers.
- **Browse the data:** Looking through the saved data.
- **Save the locations:** Saving entered location information using Core Data.
- **Handle Core Data errors:** Handling Core Data errors when there's an issue with saving.



Core Data overview

You have to persist the data for these captured locations somehow — they need to be remembered even when the app terminates.

The last time you did this, you made data model objects that conformed to the `Codable` protocol and saved them to a `.plist` file. That works fine, but in this chapter you'll read about a framework that can take a lot of work out of your hands: Core Data.

Core Data is an object persistence framework for iOS apps. If you've looked at Core Data before, you may have found the official documentation a little daunting, but the principle is quite simple.

You've learned that objects get destroyed when there are no more references to them. In addition, all objects get destroyed when the app terminates.

With Core Data, you can designate some objects as being persistent so they will always be saved to a **data store**. Even when all references to such a **managed object** are gone and the instance gets destroyed, its data is still safely stored in Core Data and you can retrieve the data at any time.

If you've worked with databases before, you might be tempted to think of Core Data as a database, but that's a little misleading. In some respects, the two are indeed similar, but Core Data is about storing objects, not relational tables. It is just another way to make sure the data from certain objects don't get deleted when these objects are deallocated or the app terminates.

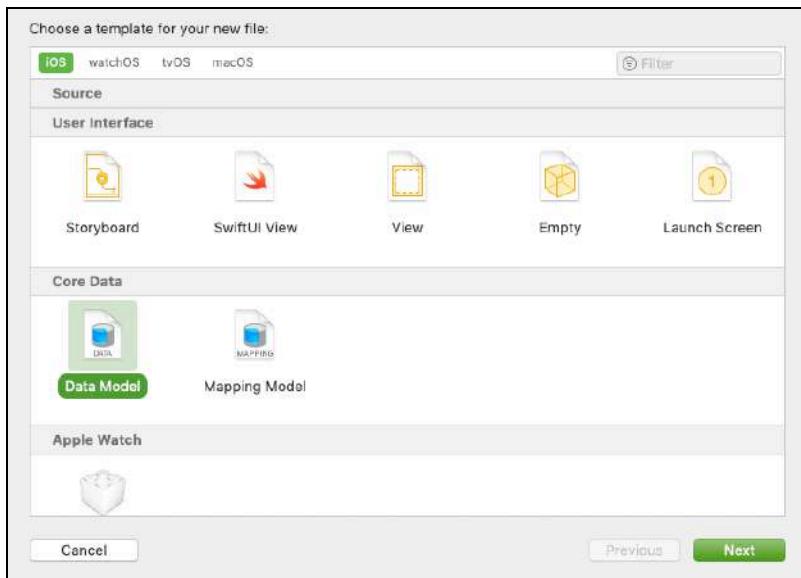
Adding Core Data

Core Data requires the use of a data model. This is a special file that you add to your project to describe the objects that you want to persist. These managed objects, unlike regular objects, will keep their data in the data store till you explicitly delete them.

Creating the data model

- Add a new file to the project. Choose the **Data Model** template under the **Core Data** section (scroll down in the template chooser):



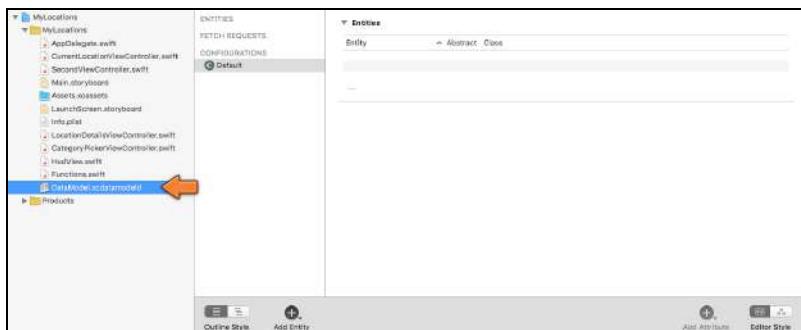


Adding a Data Model file to the project

► Save it as **DataModel**.

This will add a new file to the project, **DataModel.xcdatamodeld**.

► Click **DataModel.xcdatamodeld** in the Project navigator to open the Data Model editor:



The empty data model

For each object that you want Core Data to manage, you have to add an **entity**.

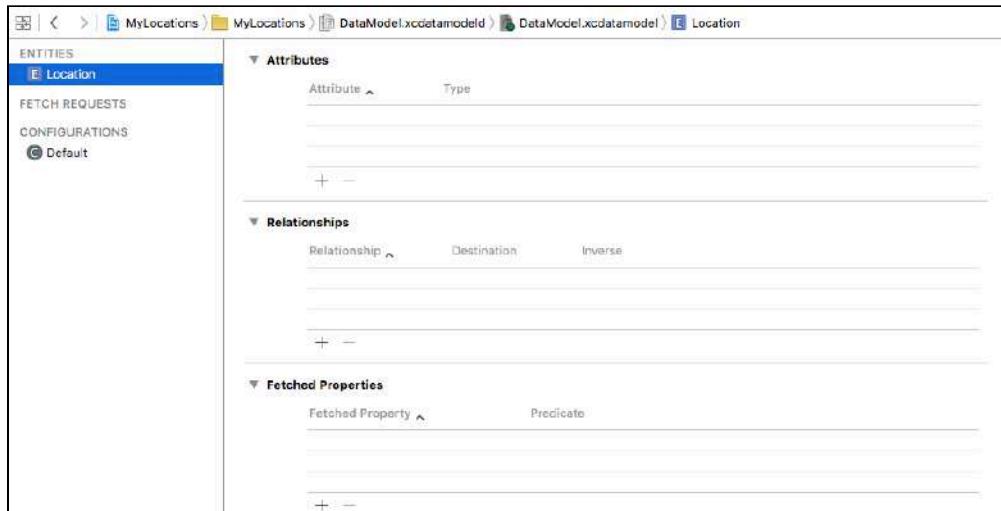
An entity describes which data fields your objects will have. In a sense, it serves the same purpose as a class, but specifically for Core Data's data store — if you've worked with SQL databases before, you can think of an entity as a table.

This app will have one entity, Location, which stores all the properties for a location that the user tagged. Each Location will keep track of the following data:

- Latitude and longitude
- Placemark (the street address)
- The date when the location was tagged
- The user’s description
- Category

These are the items from the Tag Location screen, except for the photo. Photos can potentially be very big and can take up several megabytes of storage space. Even though the Core Data store can handle big “blobs” of data, it is usually better to store photos as separate files in the app’s Documents directory. More about that later.

► Click the **Add Entity** button at the bottom of the data model editor. This adds a new entity under the ENTITIES heading. Name it **Location** — you can rename the entity by clicking its name or from the Data Model inspector pane on the right.



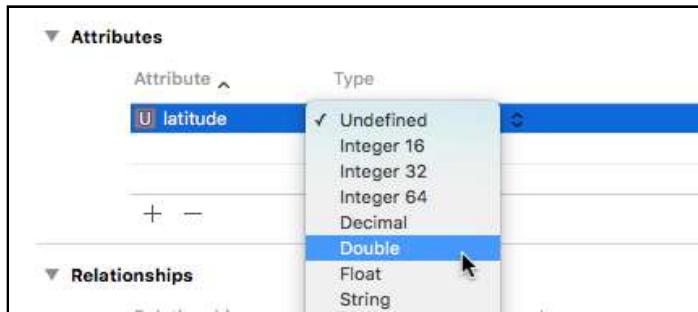
The new Location entity

The entity detail pane in the center shows three sections: Attributes, Relationships and Fetched Properties. The Attributes are the entity’s data fields.

This app only has one entity, but generally, apps will have many entities that are all related to each other somehow. With Relationships and Fetched Properties, you can tell Core Data how your objects depend on each other.

For this app, you will only use the Attributes section.

- Click the **Add Attribute** button at the bottom of the editor, or the small + button below the Attributes section. Name the new attribute **latitude** and set its **Type** to **Double**:



Choosing the attribute type

Attributes are basically the same as properties, and therefore they have a type. You've seen earlier that the latitude and longitude coordinates really have the data type **Double**. So, that's what you're choosing for the attribute as well.

Note: Don't let the change in terminology scare you. Just think:

entity = object (or class)

attribute = property

If you're wondering where you'll define methods in Core Data, then the answer is: you don't. Core Data is only for storing the data portion of objects. That is what an entity describes: the data of an object, and optionally, how that object relates to other objects if you use Relationships and Fetched Properties.

In a short while, you're going to define your own **Location** class by creating a Swift file, just as you've been doing all along. Because it describes a managed object, this class will be associated with the **Location** entity in the data model. But it's still a regular class, so you can add your own methods to it.

- Add the rest of the attributes for the **Location** entity:

- longitude, type Double
- date, type Date
- locationDescription, type String

- category, type String
- placemark, type Transformable

The data model should look like this when you're done:

ENTITIES		▼ Attributes	
	E Location	Attribute ^	Type
FETCH REQUESTS		S category	String
CONFIGURATIONS		D date	Date
C Default		N latitude	Double
		S locationDescription	String
		N longitude	Double
		T placemark	Transformable
+		-	

All the attributes of the Location entity

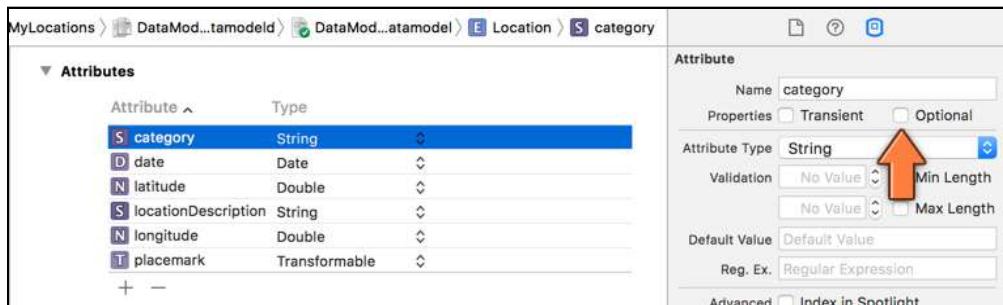
Why didn't you just call the description value "description" instead of "locationDescription"? As it turns out, `description` is the name of a method from `NSObject`. If you try to name an attribute "description," then it will cause a naming conflict with the `NSObject` method since Core Data managed objects are derived from `NSObject`. Xcode will give you an error message if you try to do this.

The type of the placemark attribute is Transformable. Core Data only supports a limited number of data types right out the box, such as `String`, `Double`, and `Date`. The placemark is a `CLPlacemark` object and is not in the list of supported data types.

Fortunately, Core Data has a provision for handling arbitrary data types. Any class that conforms to the `NSCoding` protocol can be stored in a Transformable attribute without additional work. Fortunately for us, `CLPlacemark` does conform to `NSCoding`, so you can store it in Core Data with no trouble. (And in case you are wondering, `NSCoding` is the Objective-C equivalent of the Swift `Codable` protocol — it allows classes to encode and decode themselves if they support it.)

By default, entity attributes are optional, meaning they can be `nil`. In our app, the only thing that can be `nil` is the placemark, in case reverse geocoding failed. It's a good idea to embed this constraint in the data model.

- Select the category attribute. In the inspectors panel, switch to the Data Model inspector (third tab). Uncheck the Optional setting:



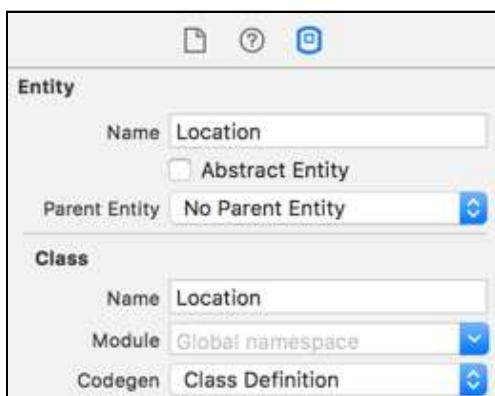
Making the category attribute non-optional

- Repeat this for the other attributes, except for placemark. (Tip: you can select multiple attributes at the same time, either by Command+clicking to select individually, or Shift+Clicking to select a range.)
- Press ⌘+S to save your changes. Xcode is supposed to do this automatically, but I've found the data model editor to be a little unreliable at times. Better safe than sorry!

You're done with the data model, but there's one more thing to do.

Generating the code

- Click on the Location entity to select it and go to the Data Model inspector.



The Data Model inspector

The **Class > Name** field says “Location.” When you retrieve a Location entity from Core Data, it gives you an instance of the Location class which is derived from



`NSManagedObject`.`NSManagedObject` is the base class for all objects that are managed by Core Data. Regular objects inherit from `NSObject`, but Core Data objects extend `NSManagedObject`.

Because using `NSManagedObject` directly is a bit limiting, Xcode helpfully sets you up to use your own `Location` class instead. You're not required to make your own classes for your entities, but it does make Core Data easier to use. So now when you retrieve a `Location` entity from the data store, Core Data doesn't give you an `NSManagedObject` but an instance of your own `Location` class.

Note also that the **Class > Codegen** dropdown is set to "Class Definition". Xcode will automatically generate the code for your entity's class with this setting so that you don't have to do any extra work. However, it is useful to understand how to make your own `NSManagedObject` subclass rather than relying on Xcode magic. So, for this app, you'll write the code yourself.

- In the inspector, change **Codegen** to **Manual/None**.

Even though you won't be using automatic class generation, Xcode can still lend a helping hand.

- From the menu bar, choose **Editor > Create NSManagedObject Subclass...**

The assistant will now ask you for the data models and entities you wish to create classes for.

- Select **DataModel** and click **Next**. In the next step, make sure **Location** is selected and click **Next** again.



Select the *Location* entity

- Choose a location to save the source files — in your case, the folder for your project. Press **Create** to finish.

This adds two new files to the project. The first one is named **Location+CoreDataClass.swift** and looks something like this:

```
import Foundation
import CoreData

@objc(Location)
public class Location: NSManagedObject {

}
```

As you can see in the `class` line, the `Location` class extends `NSManagedObject` instead of the regular `NSObject`.

You already know what the `public` and `@objc` attributes are for since you've encountered them before, but what does the `(Location)` bit do?

That is actually a part of the `@objc` attribute. The Swift compiler uses a mechanism called *name mangling* to rename methods internally so that they can be identified uniquely. Afterall, if you have two methods named `copyFiles` in the same project, how does the compiler know which one a particular bit of code refers to? It has to have a way to identify each method uniquely so that all method calls are resolved correctly.

Name mangling works fine if your project has only Swift code. But since you can combine Swift and Objective-C code in the same project, sometimes you run into trouble in such "hybrid" projects because Objective-C is not able to identify a Swift class correctly due to name mangling. This happens often when working with archived data since the archived data saves the class name and you run into issues when Objective-C can't reconcile the name it receives with a known class.

This is where the `@objc(Location)` (or similar) notation comes into play. The part inside the brackets, in this case `Location`, tells the compiler that that is the name Objective-C code will use to refer to this particular class.

You shouldn't have to worry about the above notation at all in this book since you'll be working with Swift code only. However, it's always a good idea to know things such as this for when you are a full-blown developer since you most likely will encounter a "hybrid" project at some point.

The second file that got created is **Location+CoreDataProperties.swift**:

```
import Foundation
import CoreData

extension Location {
```

```
@nonobjc public class func fetchRequest() ->
    NSFetchedResultsController<Location> {
    return NSFetchedResultsController<Location>(entityName: "Location");
}

@NSManaged public var latitude: Double
@NSManaged public var longitude: Double
@NSManaged public var date: NSDate?
@NSManaged public var locationDescription: String?
@NSManaged public var category: String?
@NSManaged public var placemark: NSManagedObject?
```

In this file, Xcode has created properties for the attributes that you specified in the Data Model editor. But what is this extension thing?

With an *extension* you can add additional functionality to an existing object without having to change the original source code for that object. This even works when you don't actually have the source code for those objects. Later on you'll see an example of how you can use an extension to add new methods to objects from iOS frameworks.

Here, the extension is used for another purpose. If you change your Core Data model at some later time and you want to automatically update the code to match those changes, then you can choose **Create NSManagedObject Subclass** again and Xcode will only overwrite what is in **Location+CoreDataProperties.swift** but not anything you added to **Location+CoreDataClass.swift**.

So, it's not a good idea to make changes to **Location+CoreDataProperties.swift** if you plan on overwriting this file later. Unfortunately, Xcode made a few small boos-boos in the types of the properties, so you'll have to make some changes to this file anyway.

The first thing to fix is the `placemark` variable. Because you made `placemark` a Transformable attribute, Xcode doesn't really know what kind of object this will be. So, it chose the generic type `NSManagedObject`. But you know it's going to be a `CLPlacemark` object. So, you can make things easier for yourself by changing it.

► First import Core Location into **Location+CoreDataProperties.swift**:

```
import CoreLocation
```

► Then change the `placemark` property to:

```
@NSManaged public var placemark: CLPlacemark?
```

You’re adding a question mark too, because `placemark` is optional.

- Finally, remove the question marks behind the `category` and `locationDescription` properties. Earlier you told Core Data these attributes were not optionals. So, they don’t need the question mark.

Because this is a *managed* object, and the data lives inside a data store, Swift will handle `Location`’s variables in a special way. The `@NSManaged` keyword tells the compiler that these properties will be resolved at runtime by Core Data. When you put a new value into one of these properties, Core Data will place that value into the data store for safekeeping, instead of in a regular instance variable.

And if you are wondering, the `@nonobjc` attribute is the reverse of the `@objc` attribute — it makes a class, method, or property not available to Objective-C. Since this came by way of generated boilerplate code, don’t worry too much about why you’d want to do that in this particular case.

This concludes the definition of the data model for *MyLocations*. Now you have to hook it up to a data store.

The data store

On iOS, Core Data stores all of its data into an SQLite — pronounced “SQL light” — database. It’s OK if you have no idea what SQLite is. You’ll take a peek into that database later, but you don’t really need to know what goes on inside the data store in order to use Core Data.

However, you do need to initialize this data store when the app starts. The code for that is the same for just about any app that uses Core Data and it goes in the app delegate class.

As you learned previously, the *app delegate* is the object that gets notifications that concern the application as a whole. This is where iOS notifies the app that it has started up, for example.

You’re going to make a few changes to the project’s `AppDelegate` class.

- Open `AppDelegate.swift` and import the Core Data framework at the very top:

```
import CoreData
```



- Add the following code inside the `AppDelegate` class (usually at the top where you define properties):

```
lazy var persistentContainer: NSPersistentContainer = {
    let container = NSPersistentContainer(name: "DataModel")
    container.loadPersistentStores { (storeDescription, error)
        in
            if let error = error {
                fatalError("Could not load data store: \(error)")
            }
        return container
    }()
}
```

This is the code you need to load the data model that you've defined earlier, and to connect it to an SQLite data store.

The goal here is to create an `NSManagedObjectContext` object. That is the object you'll use to talk to Core Data. To get that `NSManagedObjectContext` object, the app needs to do several things:

1. Create an `NSManagedObjectModel` from the Core Data model you created earlier. This object represents the data model during runtime. You can ask it what sort of entities it has, what attributes these entities have, and so on. In most apps, you don't need to use the `NSManagedObjectModel` object directly.
2. Create an `NSPersistentStoreCoordinator` object. This object is in charge of the SQLite database.
3. Finally, create the `NSManagedObjectContext` object and connect it to the persistent store coordinator.

Together, these objects are also known as the “Core Data stack.”

Previously, you had to perform the above steps one-by-one in code, which could get a little messy. But as of iOS 10, there is a new object, the `NSPersistentContainer`, that takes care of everything.

That doesn't mean you should immediately forget what you just learned about the `NSManagedObjectModel` and the `NSPersistentStoreCoordinator`, but it does save you from writing a bunch of code.

The code that you just added creates an instance variable `persistentContainer` of type `NSPersistentContainer`. To get the `NSManagedObjectContext` that we're after, you can simply ask the `persistentContainer` for its `viewContext` property.

- For convenience, add another property to get the `NSManagedObjectContext` from the persistent container:

```
lazy var managedObjectContext: NSManagedObjectContext =  
    persistentContainer.viewContext
```

Now we're ready to start using Core Data!

- Build the app to make sure it compiles without errors. If you run it, you won't notice any difference because you're not actually using Core Data anywhere yet.

Passing the context

When the user presses the Done button in the Tag Location screen, the app currently just closes the screen. Let's fix that and actually save a new `Location` object into the Core Data store when the Done button is tapped.

You use the `NSManagedObjectContext` object to talk to Core Data. It is often described as a "scratchpad." You first make your changes to the context and then you call its `save()` method to store those changes permanently in the data store.

This means that every object that needs to do something with Core Data needs to have a reference to the `NSManagedObjectContext` object.

Getting the context

- Switch to `LocationDetailsViewController.swift`. First, import Core Data at the top, and then add a new instance variable:

```
var managedObjectContext: NSManagedObjectContext!
```

The problem is: how do you put the `NSManagedObjectContext` object from the app delegate into this property?

The context object is created by `AppDelegate`, but `AppDelegate` has no reference to the `LocationDetailsViewController`. That's not so strange since the Location Details view controller doesn't exist until the user taps the Tag Location button. Prior to that, there simply is no `LocationDetailsViewController` object in existence.

The answer is to pass along the `NSManagedObjectContext` object during the segue that presents the `LocationDetailsViewController`. The obvious place for that is

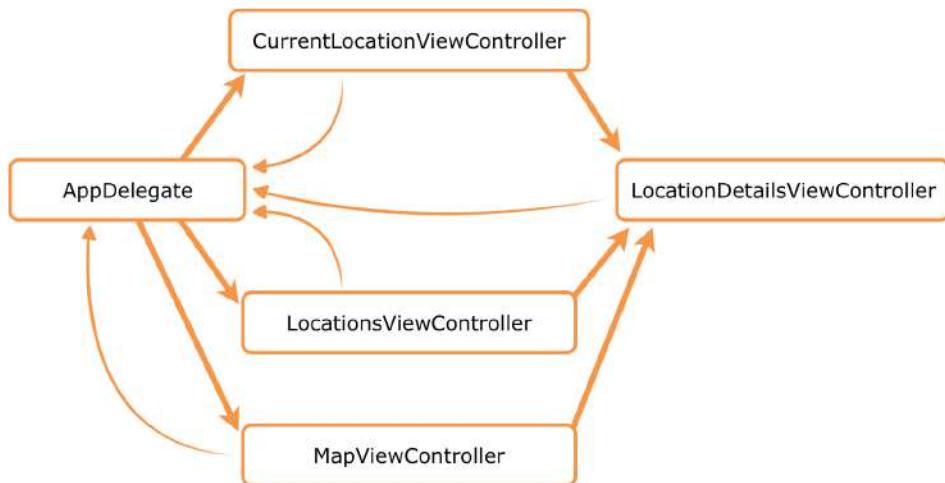
`prepare(for:sender:)` in `CurrentLocationViewController`. But then you need to find a way to get the `NSManagedObjectContext` object into the `CurrentLocationViewController` in the first place.

I come across a lot of code that does the following:

```
let delegate = UIApplication.shared.delegate as! AppDelegate  
let context = delegate.managedObjectContext  
// do something with the context
```

From anywhere in your source code, you can get a reference to the context simply by asking the `AppDelegate` for its `managedObjectContext` property. Sounds like a good solution, right? Not quite... Suddenly all your objects are dependent on the app delegate.

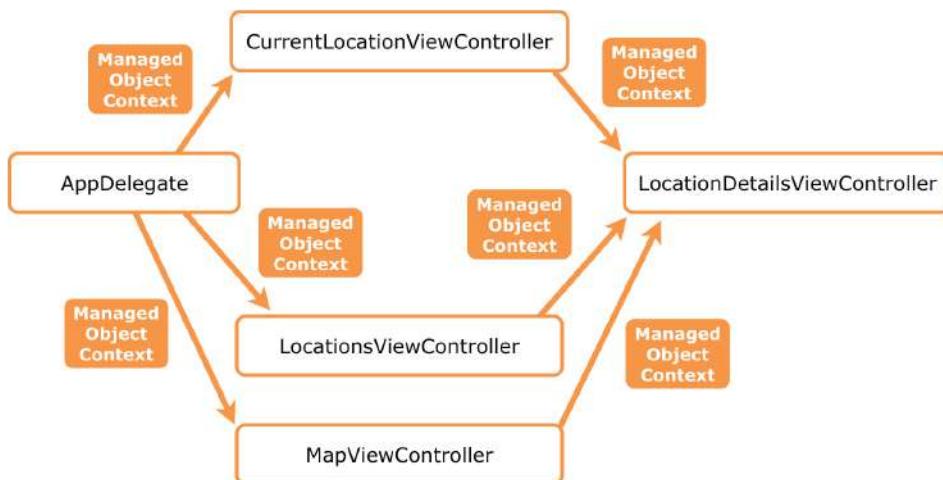
This introduces a dependency that can make your code very messy really quickly.



Bad: All classes depend on AppDelegate

As a general design principle, it is best to make your classes depend on each other as little as possible. The fewer interactions there are between the different parts of your program, the simpler it is to understand.

If many of your classes need to reach out to some shared object such as the app delegate, then you may want to rethink your design. A better solution is to give the `NSManagedObjectContext` to each object that needs it. Now all the arrows in the diagram go just one way:



Good: The context object is passed from one object to the next

Using this architecture, `AppDelegate` gives the managed object context to `CurrentLocationViewController`, which in turn will pass it on to the `LocationDetailsViewController` when it performs the segue. This technique is known as *dependency injection*.

This means `CurrentLocationViewController` needs its own property for the `NSManagedObjectContext`.

► Add the following property to `CurrentLocationViewController.swift` (and don't forget to add the Core Data import):

```
var managedObjectContext: NSManagedObjectContext!
```

► Add the following to `prepare(for:sender:)`, so that it passes on the context to the Tag Location screen:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "TagLocation" {
        // New code
        controller.managedObjectContext = managedObjectContext
    }
}
```



```
}
```

This should also explain why the `managedObjectContext` variable is declared as an implicitly unwrapped optional with the type `NSManagedObjectContext!`.

You should know by now that variables in Swift must always have a value. If they can be `nil` — which means “not a value” — then the variable must be made optional.

If you were to declare `managedObjectContext` without the exclamation point, like this:

```
var managedObjectContext: NSManagedObjectContext
```

Then Swift demands you give it a value in an `init` method — for objects loaded from a storyboard, such as view controllers, that method is `init?(coder:)`.

However, `prepare(for:sender:)` happens *after* the new view controller is instantiated, long after the call to `init?(coder:)`. As a result, inside `init?(coder:)` you can’t know what the value for `managedObjectContext` will be.

You have no choice but to leave the `managedObjectContext` variable `nil` for a short while until the segue happens, and therefore it must be an optional.

You could also have declared it like this:

```
var managedObjectContext: NSManagedObjectContext?
```

The difference between `?` and `!` is that the former requires you to manually unwrap the value with `if let` every time you want to use it.

That gets annoying really fast, especially when you know that `managedObjectContext` will get a proper value during the segue and that it will never become `nil` afterwards again. In that case, the exclamation mark is the best type of optional to use.

These rules for optionals may seem very strict — and possibly confusing — when you’re coming from another language such as Objective-C. But they are there for a good reason — by only allowing certain variables to have no value, Swift can make your programs safer and reduce the number of programming mistakes.

The fewer optionals you use, the better, but sometimes you can’t avoid them — as in this case with `managedObjectContext`.



Passing the context from AppDelegate

AppDelegate.swift now needs some way to pass the `NSManagedObjectContext` object to `CurrentLocationViewController`.

Unfortunately, Interface Builder does not allow you to make outlets for your view controllers on the App Delegate. Instead, you have to look up these view controllers by digging through the view hierarchy.

- Change the `application(_:didFinishLaunchingWithOptions:)` method to:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
                     [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    let tabController = window!.rootViewController
        as! UITabBarController

    if let tabViewControllers = tabController.viewControllers {
        let navController = tabViewControllers[0]
            as! UINavigationController
        let controller = navController.viewControllers.first
            as! CurrentLocationViewController
        controller.managedObjectContext = managedObjectContext
    }
    return true
}
```

In order to get a reference to the `CurrentLocationViewController`, you first have to find the `UITabBarController` and then look at its `viewControllers` array.

And since the first controller for the first tab is a navigation controller, then you have to go through the navigation controller's list of controllers to finally get at the `CurrentLocationViewController`.

Once you have a reference to the `CurrentLocationViewController` object, you pass it the `managedObjectContext`. It may not be immediately obvious from looking at the code, but something special happens at this point...

Remember the code for `persistentContainer` you added to `AppDelegate` earlier? You probably recognized it as a lazy loading variable since you've encountered something similar before. This is the point at which the closure for the variable is actually executed and a new `NSPersistentContainer` instance is created.

What actually happens inside the closure is fairly straightforward:

```
let container = NSPersistentContainer(name: "DataModel")
container.loadPersistentStores(completionHandler: {
    storeDescription, error in
    if let error = error {
        fatalError("Could not load data store: \(error)")
    }
})
return container
```

You instantiate a new `NSPersistentContainer` object with the name of the data model you created earlier, `DataModel`. Then you tell it to `loadPersistentStores()`, which loads the data from the database into memory and sets up the Core Data stack.

There is another closure here, given by the `completionHandler` parameter. The code in this closure gets invoked when the persistent container is done loading the data. If something went wrong, you print an error message — useful for debugging! — and terminate the app using the function `fatalError()`.

Now that you know what it does, you may be wondering why you didn't just put all of this code into a regular method like this:

```
var persistentContainer: NSPersistentContainer

init() {
    persistentContainer = createPersistentContainer()
}

func createPersistentContainer() -> NSPersistentContainer {
    // all the initialization code here
    return container
}
```

That would certainly be possible, but now the initialization of `persistentContainer` is spread over three different parts of the code: the declaration of the variable, the method that performs all the initialization logic, and the `init` method to tie it all together.

Isn't it nicer to keep all this stuff in one place, rather than in three different places? Swift lets you perform complex initialization right where you declare the variable. That's pretty nifty!

There's another thing going on here:

```
lazy var persistentContainer: NSPersistentContainer = { ... }()
```

Notice the `lazy` keyword? That means the entire block of code in the `{ ... }()` closure isn't actually performed right away. The context object won't be created until you ask for it. This is another example of *lazy loading*, similar to how you handled `DateFormatter` earlier.

The `managedObjectContext` property is also declared `lazy`:

```
lazy var managedObjectContext: NSManagedObjectContext =  
    persistentContainer.viewContext
```

This is necessary because its initial value comes from `persistentContainer`. It also used to be necessary to use `self` here to refer to `persistentContainer`. Otherwise, Xcode would give a compiler error. That appears not to be the case with Xcode 11, but if you run into this issue, then you know what to do!

- Run the app. Everything should still be the way it was, but behind the scenes a new database has been created for Core Data.

Browsing the data

Core Data stores the data in an SQLite database. That file is named **DataModel.sqlite** and it lives in the app's Library folder. That's similar to the Documents folder that you saw previously.

You can see it in Finder if you go to `~/Library/Developer/CoreSimulator` and then to the folder that contains the data for *MyLocations* on a particular simulator.

Core Data data store location

- The easiest way to find this folder is to add the following to **Functions.swift**:

```
let applicationDocumentsDirectory: URL = {  
    let paths = FileManager.default.urls(for: .documentDirectory,  
                                         in: .userDomainMask)  
    return paths[0]  
}()
```

This creates a new global constant, `applicationDocumentsDirectory`, containing the path to the app's Documents directory. It's a global because you're not putting this inside a class. This constant will exist for the duration of the app; it never goes out of scope. You could have made a method for this as you did for *Checklists*, but using a global constant works just as well.



As before, you're using a closure to provide the code that initializes this constant. Like all globals, this is evaluated lazily the very first time it is used.

Note: Globals have a bad reputation. Many programmers avoid them at all costs. The problem with globals is that they create hidden dependencies between the various parts of your program. And dependencies make the program hard to change and hard to debug.

But used well, globals can be very handy. It's feasible that your app will need to know the path to the Documents directory in several different places. Putting it in a global constant is a great way to solve that design problem.

- Add the following line to `application(_:didFinishLaunchingWithOptions:)` – a good place would be just before the final `return` statement:

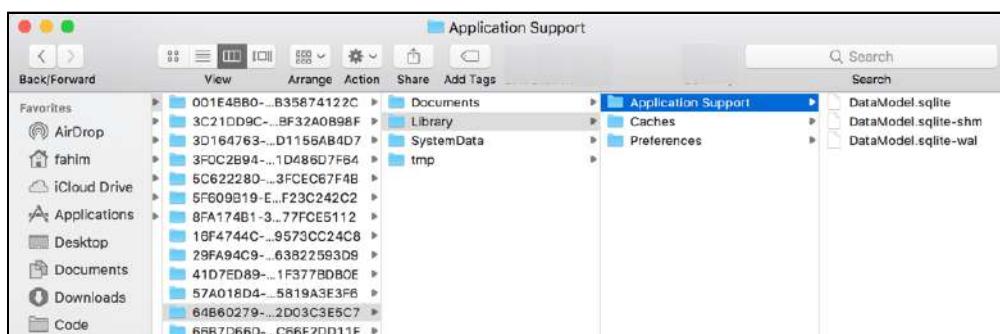
```
print(applicationDocumentsDirectory)
```

On my computer this prints out:

```
file:///Users/adamrush/Library/Developer/CoreSimulator/Devices/
CA23DAEA-DF30-43C3-8611-E713F96D4780/data/Containers/Data/
Application/64B60279-41D1-46A4-83A7-492D03C3E5C7/Documents/
```

- Open a new Finder window and press **Shift+⌘+G**. Then copy-paste the path without the `file://` bit (note that you leave out only two slashes out of the three...) to go to the Documents folder.

The database is not actually in the Documents folder, so go back up one level and enter the **Library** folder, and then **Application Support**:



The new database in the app's Documents directory

The **DataModel.sqlite-shm** and **-wal** files are also part of the data store.

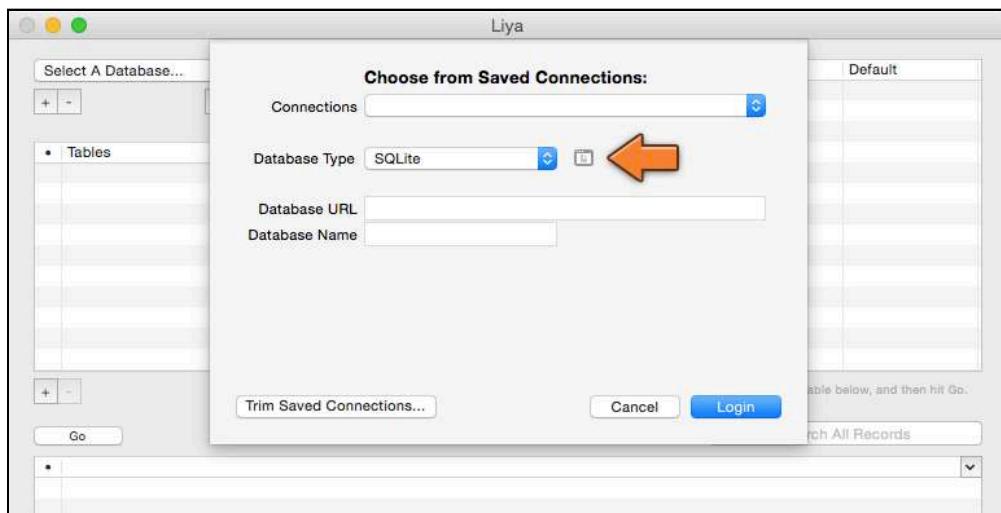
This database is still empty because you haven't stored any objects in it yet, but just for the fun of it, you'll take a peek inside.

There are several handy — and free! — tools that give you a graphical interface for interacting with your SQLite databases.

Browsing the Core Data store using a GUI app

You will use **Liya** to examine the data store file. Download it from the Mac App Store or cutedgesystems.com/software/liya/.

- Start Liya. It asks you for a database connection. Under **Database Type** choose **SQLite**.



Liya opens with this dialog box

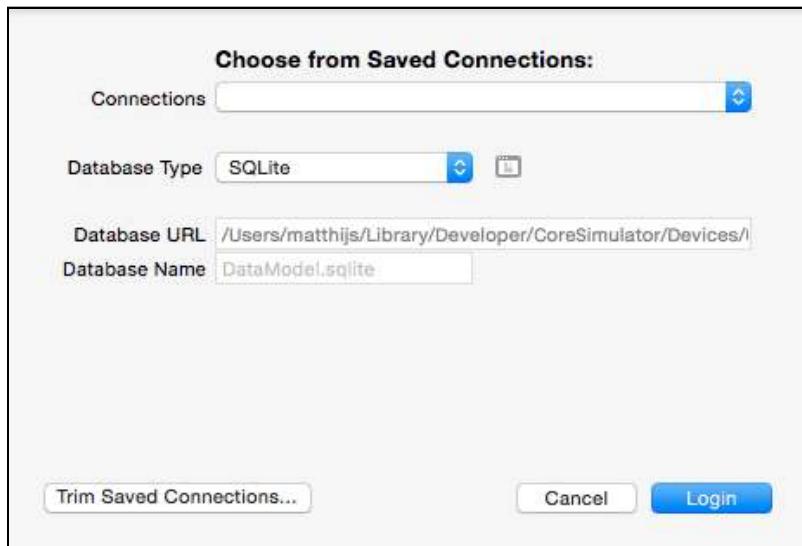
- On the right of the Database Type field is a small icon. Click this to open a file picker.

You can navigate to the **CoreSimulator/.../Library/Application Support** folder, but that's a lot of work (it's a very deeply nested folder).

If you have the Finder window still open, it's easier to drag the **DataModel.sqlite** file from Finder directly on to the open file picker. Click **Choose** when you're done.

Tip: You can also right-click the DataModel.sqlite file in Finder and choose **Open With ▶ Liya** from the pop-up menu.

The **Database URL** field should now contain the app's Document folder and **Database Name** should say DataModel.sqlite:



Connecting to the SQLite database

► Click **Login** to proceed.

The screen should look something like this:

Field	Type	Length	Null	Key	Default
Z_PK	integer		NO	PRI	
Z_ENT	integer		YES		
Z_OPT	integer		YES		
ZDATE	timestamp		YES		
ZLATITUDE	float		YES		
ZLONGITUDE	float		YES		
ZCATEGORY	varchar		YES		
ZLOCATIONDESCRIPTION	varchar		YES		
ZPLACEMARK	blob		YES		

The empty DataModel.sqlite database in Liya

The ZLOCATION table is where your Location objects will be stored. It's currently empty, but on the right you can already see the column names that correspond to

your fields: ZDATE, ZLATITUDE, and so on. Core Data also adds its own columns and tables (with the Z_ prefix).

You're not really supposed to change anything in this database by hand, but sometimes using a visual tool like this is handy to see what's going on. You'll come back to Liya once you've inserted new Location objects.

Note: An alternative to Liya is SQLiteStudio, [sqlitestudio.pl](#). You can find more tools, paid and free, on the Mac App Store by searching for “sqlite.”

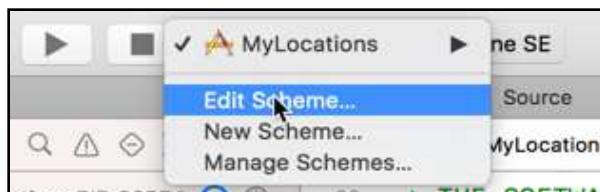
Troubleshooting Core Data issues

There is another handy tool for troubleshooting Core Data. By setting a special flag on the app, you can see the SQL statements that Core Data uses under the hood to talk to the data store.

Even if you have no experience with SQL, this is still valuable information. At least you can use it to tell whether Core Data is doing something or not. To enable this tool, you have to edit the project's *scheme*.

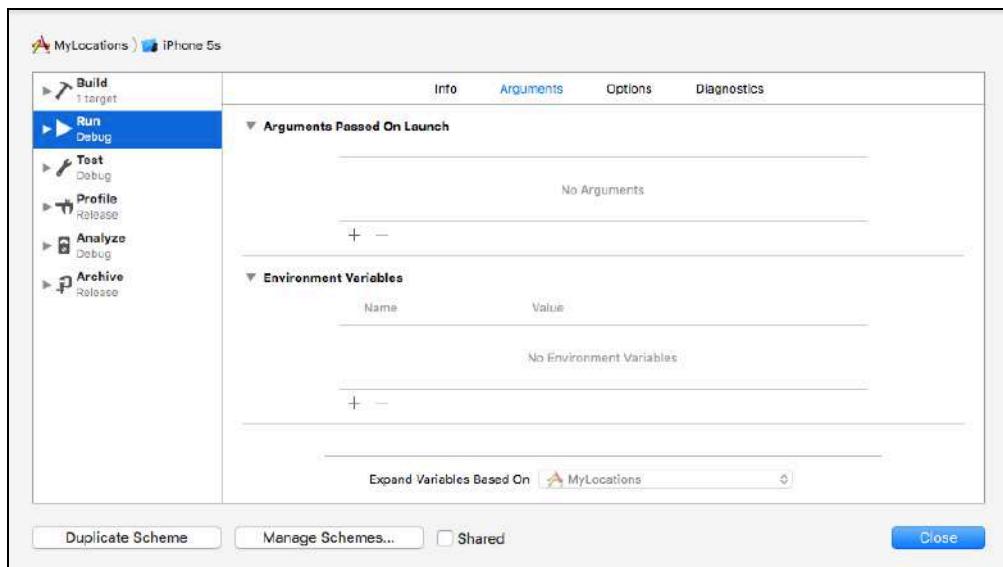
Schemes are how Xcode lets you configure your projects. A scheme is a bunch of settings for building and running your app. Standard projects have just one scheme, but you can add additional schemes, which is handy when your project becomes bigger.

- Click on the left part of the **MyLocations** ▶ **iPhone** bar at the top of the screen and choose **Edit Scheme...** from the menu.



The *Edit Scheme...* option

The following panel should pop up:



The scheme editor

- Choose the **Run** option on the left-hand side.
- Select the **Arguments** tab.
- In the **Arguments Passed On Launch** section, add the following:

```
-com.apple.CoreData.SQLDebug 1  
-com.apple.CoreData.Logging.stderr 1
```



Adding the SQLDebug launch argument

- Press **Close** to close this dialog, and run the app.

You should see something like this in the Xcode Console:

```
CoreData: annotation: Connecting to sqlite database file at "/  
Users/fahim/Library/Developer/CoreSimulator/Devices/CA23DAEA-  
DF30-43C3-8611-E713F96D4780/data/Containers/Data/Application/  
B3C8FED1-3218-454F-B86F-1482ED64433A/Library/Application  
Support/DataModel.sqlite"  
CoreData: sql: SELECT TBL_NAME FROM SQLITE_MASTER WHERE TBL_NAME  
= 'Z_METADATA'  
CoreData: sql: pragma recursive_triggers=1  
CoreData: sql: pragma journal_mode=wal  
CoreData: sql: SELECT Z_VERSION, Z_UUID, Z_PLIST FROM Z_METADATA  
CoreData: sql: SELECT TBL_NAME FROM SQLITE_MASTER WHERE TBL_NAME  
= 'Z_MODELCACHE'
```

This is the debug output from Core Data. If you understand SQL, some of this will look familiar. The specifics don't matter, but it's clear that Core Data is connecting to the data store at this point. Excellent!

Saving the locations

You've successfully initialized Core Data and passed the `NSManagedObjectContext` to the Tag Location screen. Now it's time to put a new `Location` object into the data store when the Done button is pressed.

- Add the following instance variable to `LocationDetailsViewController.swift`:

```
var date = Date()
```

You're adding this variable because you need to store the current date in the new `Location` object. You only want to make that `Date` object once.

- In `viewDidLoad()`, change the line that sets the `dateLabel`'s text to:

```
dateLabel.text = format(date: date)
```

This now uses the new property instead of creating the date on the fly.

- Change the `done()` method to the following:

```
@IBAction func done() {  
    let hudView = HudView.hud(inView: navigationController!.view,  
                             animated: true)  
    hudView.text = "Tagged"  
    // 1  
    let location = Location(context: managedObjectContext)
```

```
// 2
location.locationDescription = descriptionTextView.text
location.category = categoryName
location.latitude = coordinate.latitude
location.longitude = coordinate.longitude
location.date = date
location.placemark = placemark
// 3
do {
    try managedObjectContext.save()
    afterDelay(0.6) {
        hudView.hide()
        self.navigationController?.popViewControllerAnimated(true)
    }
} catch {
    // 4
    fatalError("Error: \(error)")
}
}
```

This is where you do all the work:

1. First, you create a new `Location` instance. Because this is a managed object, you have to use its `init(context:)` method. You can't just write `Location()` because then the `managedObjectContext` won't know about the new object.
2. Once you have created the `Location` instance, you can use it like any other object. Here you set its properties to whatever the user entered in the screen.
3. You now have a new `Location` object whose properties are all filled in, but if you were to look in the data store at this point, you'd still see no objects there. That won't happen until you `save()` the context. Saving takes any objects that were added to the context, or any managed objects that had their contents changed, and permanently writes these changes to the data store. That's why they call the context a "scratchpad"; its changes aren't persisted until you save them.

The `save()` method can fail for a variety of reasons and therefore you need to catch any potential errors. That's done using Swift error handling, which you've encountered before.

4. Output the error and then terminate the application via the system method `fatalError`. But where does the `error` variable that you output come from? This is a local constant that Swift automatically populates with the error that it caught — handy, huh?

- Run the app and tag a location. Enter a description and press the Done button.

If everything went well, Core Data will dump a whole bunch of debug information into the debug area:

```
CoreData: sql: BEGIN EXCLUSIVE
.
.
.
CoreData: sql: INSERT INTO ZLOCATION(Z_PK, Z_ENT, Z_OPT,
ZCATEGORY, ZDATE, ZLATITUDE, ZLOCATIONDESCRIPTION, ZLONGITUDE,
ZPLACEMARK) VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
CoreData: sql: COMMIT
.
.
.
CoreData: annotation: sql execution time: 0.0001s
```

These are the SQL statements that Core Data performs to store the new Location object in the database.

- In Liya, refresh the contents of the ZLOCATION table (press the Go button below the Tables list). There should now be one row in that table:

A new row was added to the table

Note: If you don't see any rows in the table, press the Stop button in Xcode first to exit the app. You can also try closing the Liya window and opening a new connection to the database. Sometimes, the Simulator data folder locations appear to change between app runs. So, you might need to set up a new database connection in Liya after each run if this happens.

As you can see, the columns in this table contain the property values from the Location object. The only column that is not readable is ZPLACEMARK. Its contents have been encoded as a binary “blob” of data. That is because it’s a Transformable attribute and the NSCoding protocol has converted its fields into a binary chunk of data.

If you don't have Liya or are a command line junkie, then there is another way to examine the contents of the database. You can use the Terminal app and the `sqlite3` tool, but you'd better know your SQL's from your ABC's if you want to go that route:



```
$ cd /Users/mattijis/Library/Developer/CoreSimulator/Devices/66422991-21E3-4394-8DCE-0584865EA854/data/Containers/Data/Application/F6C79307-5123-410D-8F4F-2952FD08DBF9/Library/Application\ Support
$ sqlite3 DataModel.sqlite
SQLite version 3.13.0 2016-05-02 15:00:23
Enter ".help" for usage hints.
sqlite> select * from ZLOCATION;
1|1|1|491139464.476779|37.33233141|-122.0312186|Apple Store|Apple HQ|plist@0?/?X$versionX$objectsY$archiver!Stop
sqlite> .q
$
```

Examining the database from the Terminal

Handling Core Data errors

To save the contents of the context to the data store, you did:

```
do {
    try managedObjectContext.save()
} catch {
    fatalError("Error: \(error)")
}
```

What if something goes wrong with the save? In that case, code execution jumps to the `catch` branch and you call the `fatalError()` function. That will immediately kill the app and return the user to the iPhone's Springboard. That's a nasty surprise for the user, and therefore, not recommended.

The good news is that Core Data only gives an error if you're trying to save something that is not valid. In other words, when there is some bug in your app.

Of course, you'll get all the bugs out during development so users will never experience any, right? The sad truth is that you'll never catch all your bugs. Some always manage to slip through.

Unfortunately, there isn't much else to do but crash when Core Data does give an error. Something went horribly wrong somewhere and now you're stuck with invalid data. If the app were allowed to continue, things would likely only get worse, as there is no telling what state the app is in. The last thing you want to do is to corrupt the user's data.

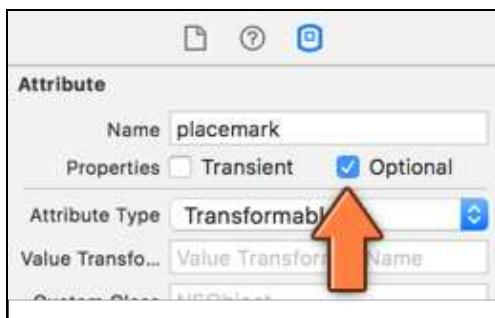
However, instead of making the app crash hard with `fatalError()`, it might be nice to tell the user about the issue first so at least they know what is happening. The crash is still inevitable, but now your users will know why the app suddenly stopped working.

In this section, you'll add a pop-up alert for handling such situations. Again, these errors should happen only during development, but just in case they do occur to an actual user, you'll try to handle it with at least a little bit of grace.

Faking errors for testing purposes

Here's a way to fake such a fatal error, just to illustrate what happens.

- Open the data model (`DataModel.xcdatamodeld` in the file list), and select the `placemark` attribute. In the Data Model inspector, uncheck the **Optional** flag.



Making the `placemark` attribute non-optional

That means `location.placemark` can never be `nil`. This is a constraint that Core Data will enforce. When you try to save a `Location` object to the data store whose `placemark` property is `nil`, Core Data will throw a tantrum. So that's exactly what you're going to do here, just to test your error handling code and to make sure the app fails gracefully.

- Run the app. It is possible that the app crashes right away...

What happens is that you have just changed the data model by making changes to the `placemark` attribute. When you launch the app, the `NSPersistentContainer` notices this and tries to perform a “migration” of the SQLite database to the new, updated data model.

The migration may succeed... or not... depending on what is currently in your data store. If you previously tagged a location that did not have a valid address — i.e. whose placemark is `nil` — then the migration to the new data model fails. After all, the new data model does not allow for placemarks that are `nil`.

If the app crashed for you, then the debug area says why:

```
reason=Cannot migrate store in-place: Validation error missing
attribute
values on mandatory destination attribute, . . .
{entity=Location, attribute=placemark, . . .}
```

The `DataModel.sqlite` file is out of date with respect to the changed data model, and Core Data can't automatically resolve this issue.

There are two ways to fix this:

1. Simply throw away the `DataModel.sqlite` file from the Library directory.
2. Remove the entire app from the Simulator.
► Remove the `DataModel.sqlite` file, as well as the `-shm` and `-wal` files, and run the app again.

It's important to know that changing the data model may require you to throw away the database file or Core Data cannot be initialized properly.

Note: Not all is lost if `NSPersistentContainer`'s migration fails. Core Data allows you to perform your own migrations when you release an update to your app with a new data model. Instead of crashing, this mechanism allows you to convert the contents of the user's existing data store to the new model. However, during development, it is just as easy to toss out the old database.

► Now here's the trick. Tap the Get My Location button and then tap immediately on Tag Location. If you do that quickly enough, you can beat the reverse geocoder to it and the Tag Location screen will say: "No Address Found." It only says that when `placemark` is `nil`.

If geocoding happens too fast, you can fake this by temporarily commenting out the line `self.placemark = p.last!` in `locationManager(_:didUpdateLocations:)` inside `CurrentLocationViewController.swift`. This will make it seem as if no address was found and the value of `placemark` stays `nil`.



- Tap the Done button to save the new Location object.

The app will crash:

```
libswiftCore.dylib`_swift_runtime_on_report:
0x109e3e920 <+0>: pushq %rbp
= Thread 1: Fatal error: Error: Error Domain=NSCocoaErrorDomain Code=1570 "The operation
couldn't be completed. (Cocoa error 1570.)" UserInfo={NSValidationErrorObject=<Location:
0x600000ba73e0> (entity: Location; id: 0x6000028d4540 <x-coredata:///Location/
tCB66E15D-3FC4-4884-BEC6-2BDD21E960B2>; data: {
category = "No Category";
date = "2018-07-07 09:06:17 +0000";
latitude = "18.785834";
locationDescription = One;
longitude = "-122.406417";
placemark = nil;
}), NSValidationErrorKey=placemark}
```

The app crashes after a Core Data error

At the very end of that error message above, you can see that it says:

NSValidationErrorKey=placemark

This means the `placemark` attribute did not validate properly. Because you set it to non-optional, Core Data does not accept a `placemark` value that is `nil`.

Of course, what you've just seen only happens when you run the app from Xcode — when it crashes, the debugger takes over and points at the line with the error. But that's not what the user sees.

- Stop the app. Now tap the app's icon in the Simulator to launch the app outside of Xcode. Repeat the same procedure to make the app crash. The app will simply cease functioning and disappear from the screen.

Imagine this happening to a user who just paid 99 cents (or more) for your app. They'll be horribly confused, "What just happened?!" They may even ask for their money back. It's better to show an alert when this happens. After the user dismisses that alert, you'll still make the app crash, but at least the user knows the reason why. (The alert message should probably ask them to contact you and explain what they did, so you can fix that bug in the next version of your app.)

Alerting the user about crashes

- Add the following code to **Functions.swift**:

```
let CoreDataSaveFailedNotification =
Notification.Name("CoreDataSaveFailedNotification")

func fatalCoreDataError(_ error: Error) {
    print("/** Fatal error: \(error)")
    NotificationCenter.default.post(
        name: CoreDataSaveFailedNotification, object: nil)
}
```

This defines a new global function for handling fatal Core Data errors.

- Replace the error handling code in the `done()` action (in **LocationDetailsViewController.swift**) with:

```
} } catch {
    fatalCoreDataError(error)
}
```

The call to `fatalCoreDataError()` has taken the place of `fatalError()`. So what does that new function do, actually?

It first outputs the error message to the Console using `print()` because it's always useful to log such errors. After dumping the debug info, the function does the following:

```
NotificationCenter.default.post(
    name: CoreDataSaveFailedNotification, object: nil)
```

I've been using the term "notification" to mean any generic event or message being delivered, but the iOS SDK also has an object called the `NotificationCenter` — not to be confused with Notification Center on your iOS device.

The code above uses `NotificationCenter` to post a notification. Any object in your app can subscribe to such notifications and when these occur, `NotificationCenter` will call a certain method in those listener objects.

Using this official notification system is yet another way that your objects can communicate with each other. The handy thing is that the object that sends the notification and the object that receives the notification don't need to know anything about each other.

The sender just broadcasts the notification to all and doesn't really care what happens to it. If anyone is listening, great. If not, then that's cool too.

UIKit defines a lot of standard notifications that you can subscribe to. For example, there is a notification that lets you know that the app is about to be suspended after the user taps the Home button.

You can also define your own notifications, and that is what you've done here. The new notification is called `CoreDataSaveFailedNotification`.

The idea is that there is one place in the app that listens for this notification, pops up an alert view, and terminates. The great thing about using `NotificationCenter` is that your Core Data code does not need to care about any of this.

Whenever a saving error occurs, no matter at which point in the app, the `fatalCoreDataError(_:)` function sends out this notification, safe in the belief that some other object is listening for the notification and will handle the error.

So who will actually handle the error? The app delegate is a good place for this. It's the top-level object in the app and it's always guaranteed to exist.

► Add the following method to `AppDelegate.swift`:

```
// MARK:- Helper methods
func listenForFatalCoreDataNotifications() {
    // 1
    NotificationCenter.default.addObserver(
        forName: CoreDataSaveFailedNotification,
        object: nil, queue: OperationQueue.main,
        using: { notification in
            // 2
            let message = """
There was a fatal error in the app and it cannot continue.

Press OK to terminate the app. Sorry for the inconvenience.
"""

            // 3
            let alert = UIAlertController(
                title: "Internal Error", message: message,
                preferredStyle: .alert)

            // 4
            let action = UIAlertAction(title: "OK",
                                      style: .default) { _ in
                let exception = NSError(
                    name: NSErrorName.internalInconsistencyException,
                    reason: "Fatal Core Data error", userInfo: nil)
                exception.raise()
            }
        }
    )
}
```

```
        alert.addAction(action)

        // 5
        let tabController = self.window!.rootViewController!
        tabController.present(alert, animated: true,
                              completion: nil)
    })
}
```

Here's how this works step-by-step:

1. Tell `NotificationCenter` that you want to be notified whenever a `CoreDataSaveFailedNotification` is posted. The actual code that is performed when that happens sits in a closure following `using:`.
2. Set up the error message to display. This could have been done using a normal string by inserting new lines (`\n`) as you've seen done before, but this shows another way to do this — using multiline strings.

Note that the multiline string starts and ends with a triple quote (""""") and that the first line of the string has to start on a new line and the closing triple quotes have to be on a new line as well. You can include new lines and other special characters like quotes within the string. So it can be really handy, even if it looks a little weird!

3. Create a `UIAlertController` to show the error message and use the multiline string from earlier as the message.
4. Add an action for the alert's OK button. The code for handling the button press is again a closure — these things are everywhere! Instead of calling `fatalError()`, the closure creates an `NSError` object to terminate the app. That's a bit nicer and it provides more information to the crash log.
5. To show the alert with `present(animated:completion:)` you need a view controller that is currently visible. You simply use the window's `rootViewController` — in this app that is the tab bar controller — since it will be visible at all times as per the current navigation flow of the app.

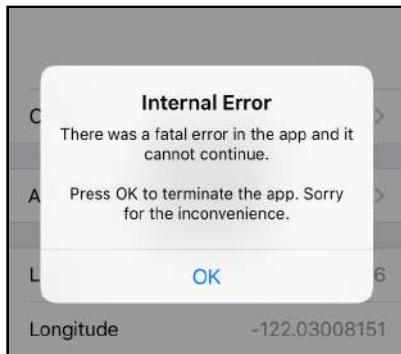
All that remains is calling this new method so that the notification handler is registered with `NotificationCenter`.

► Add the following to `application(_:didFinishLaunchingWithOptions:)`, just before the `return true` statement:

```
listenForFatalCoreDataNotifications()
```



- Run the app again and try to tag a location before the street address has been obtained. Even though the app still crashes when you tap the OK button on the alert, at least now it tells the user what's going on:



The app crashes with a message

Again, it's important to test your app thoroughly to make sure you're not giving Core Data any objects that do not validate. You want to avoid these save errors at all costs!

Ideally, users should never have to see that alert view, but it's good to have in place because there are no guarantees your app won't have bugs.

Note: You can legitimately use `managedObjectContext.save()` to let Core Data validate user input. There is no requirement that you make your app crash after an unsuccessful save, only if the error was unexpected and definitely shouldn't have happened!

Besides the “optional” flag, there are many more validation settings you can set for your entities. If you let users enter data that needs to go into these attributes, then it's perfectly acceptable to use `save()` to validate input. If it throws an error, then a user input is invalid and you need to handle it.

- In the data model, set the **placemark** attribute back to optional (and uncomment the code in **CurrentLocationViewController.swift**, if you did comment out the **placemark** line).

Run the app just to make sure everything works as it should.

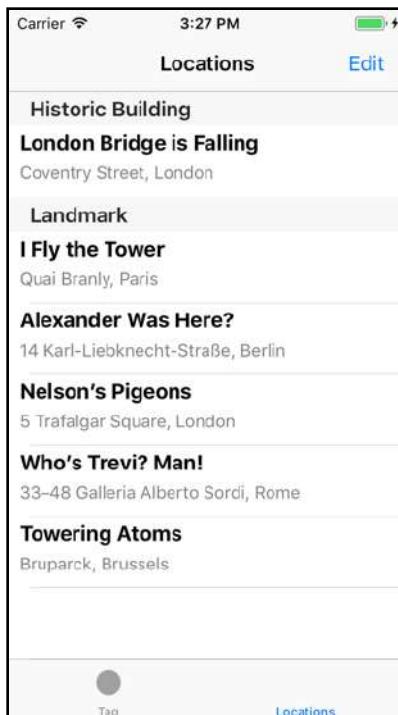
You can find the project files for this chapter under **32 – Saving Locations** in the Source Code folder.

33 Chapter 33: The Locations Tab

Eli Ganim

You've set up the data model and given the app the ability to save new locations to the data store. Next, you'll show these saved locations in a table view in the second tab.

The completed Locations screen will look like this:



The Locations screen



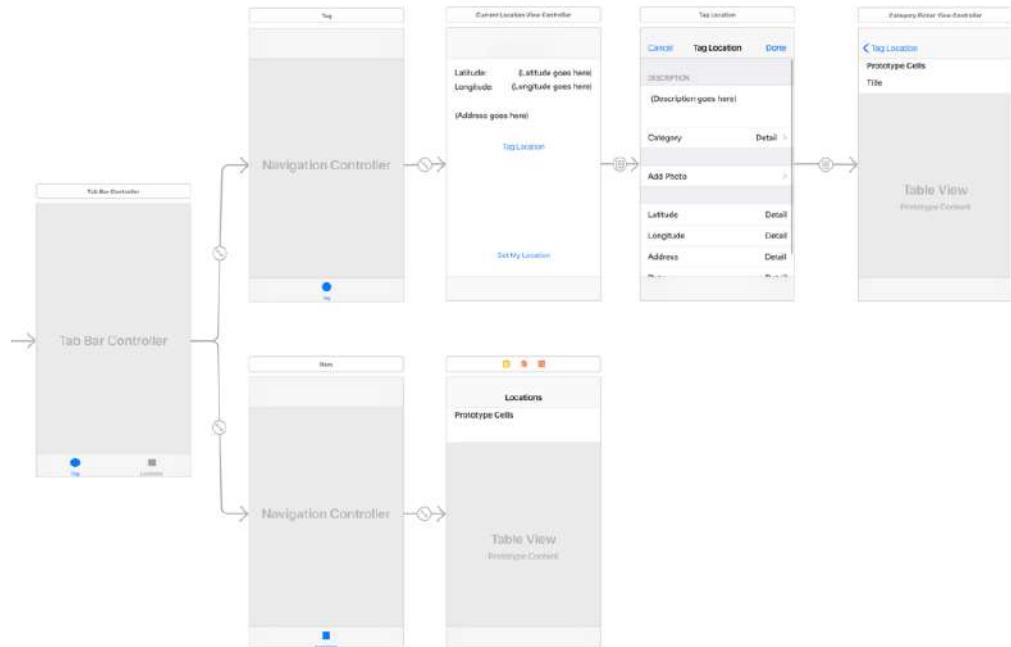
This chapter covers the following:

- **The locations tab:** Set up the second tab to display a list of saved locations.
- **Create a custom table view cell subclass:** Create a custom table view cell subclass to handle displaying location information.
- **Edit locations:** Add functionality to allow editing of items in the locations list.
- **Use NSFetchedResultsController:** How do you use `NSFetchedResultsController` to fetch data from your Core Data store?
- **Delete Locations:** Add the ability to the UI to delete locations, thus removing them from the Core Data store as well.
- **Table view sections:** Use built-in Core Data functionality to add the ability to display separate sections based on the location category.

The Locations tab

- Open the storyboard editor and delete the **Second Scene**. This is a leftover from the project template and you don't need it.
- Drag a new **Navigation Controller** on to the canvas — it has a table view controller attached to it, which is fine. You'll use that in a second.
- **Control-drag** from the Tab Bar Controller to this new Navigation Controller and select **Relationship Segue – view controllers**. This adds the navigation controller to the tab bar.
- The Navigation Controller now has a **Tab Bar Item** that is named "Item." Rename it to **Locations**.
- Double-click the navigation bar of the new table view controller (the one attached to the new Navigation Controller) and change the title to **Locations**. (If Xcode gives you trouble, use the Attributes inspector on the Navigation Item instead.)

The storyboard now looks like this:



The storyboard after adding the Locations screen

- Run the app and activate the Locations tab. It doesn't show anything useful yet:



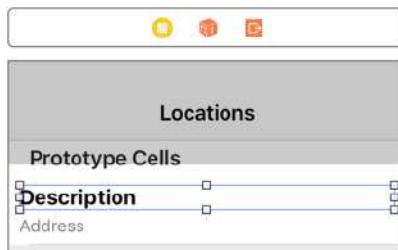
The Locations screen in the second tab

Designing the table view cell

Before you can show any data in the table, you first have to design the prototype cell.

- Set the prototype cell's Reuse Identifier to **LocationCell**.
- In the **Size inspector**, change **Row Height** to 58.
- Drag two Labels on to the cell. Give the top one the text **Description** and the bottom one the text **Address**. This is just so you know what they are for.
- Set the font of the Description label to **System Bold**, size 17. Give this label a tag of 100.
- Set the font of the Address label to **System**, size **14**. Set the Text color to black with 50% opacity (so it looks like a medium gray). Give it a tag of 101.

The cell will look something like this:



The prototype cell

Make sure that the labels are wide enough to span the entire cell, position them vertically to suit your taste and then set up AutoLayout constraints for the **left**, **top**, **right**, and **bottom** so that the labels stay in place even if the screen dimensions changed.

Previously, you would have had to set the table view row height in the Size inspector as well. But these days, with automatic sizing enabled by default, you don't have to worry about that.

But you might have to set the vertical resistance compression for one or the other of the labels since both of them have the same vertical compression resistance at the moment and so iOS will be unable to determine which takes precedence when the content for the labels will not fit well. You can lower the vertical compression resistance of the Address label or change the other label. It shouldn't make a huge difference here.

The basic table view controller

Let's write the code for the view controller. You've seen table view controllers several times now, so this should be easy.

You're going to fake the content first, because it's a good idea to make sure that the prototype cell works before you have to deal with Core Data.

- Add a new file to the project and name it **LocationsViewController.swift**.

Tip: If you want to keep your list of source files neatly sorted by name in the project navigator, then right-click the MyLocations group (the yellow folder icon) and choose **Sort by Name** from the menu.

- Change the contents of **LocationsViewController.swift** to:

```
import UIKit
import CoreData
import CoreLocation

class LocationsViewController: UITableViewController {
    var managedObjectContext: NSManagedObjectContext!

    // MARK: - Table View Delegates
    override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView,
        cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {
        let cell = tableView.dequeueReusableCell(
            withIdentifier: "LocationCell",
            for: indexPath)

        let descriptionLabel = cell.viewWithTag(100) as! UILabel
        descriptionLabel.text = "If you can see this"

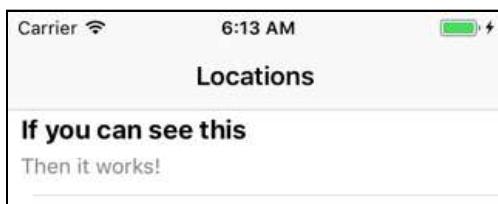
        let addressLabel = cell.viewWithTag(101) as! UILabel
        addressLabel.text = "Then it works!"

        return cell
    }
}
```



You've faked a single row with some placeholder text in the labels. You've also given this class an `NSManagedObjectContext` property even though you won't be using it yet.

- Switch to the storyboard, select the Locations scene, and in the **Identity inspector**, change the **Class** of the table view controller to **LocationsViewController**. (Be careful with the auto completion when you're doing this since you also have a `LocationsDetailViewController` and that might get auto added if you are not careful...)
- Run the app to make sure the table view works.



The table view with fake data

Excellent! Now it's time to fill the table with the Location objects from the data store.

Displaying Locations from data store

- Run the app and tag a handful of locations. If there is no data in the data store, then the app doesn't have much to show...

This new part of the app doesn't know anything yet about the `Location` objects that you have added to the data store. In order to display them in the table view, you need to obtain references to these objects somehow. You can do that by asking the data store. This is called *fetching*.

- First, add a new instance variable to `LocationsViewController.swift`:

```
var locations = [Location]()
```

This array will hold the list of `Location` objects.

- Add a `viewDidLoad()` implementation:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // 1
    let fetchRequest = NSFetchedResultsController<Location>()
```

```
// 2
let entity = Location.entity()
fetchRequest.entity = entity
// 3
let sortDescriptor = NSSortDescriptor(key: "date",
                                       ascending: true)
fetchRequest.sortDescriptors = [sortDescriptor]
do {
    // 4
    locations = try managedObjectContext.fetch(fetchRequest)
} catch {
    fatalCoreDataError(error)
}
```

This may look daunting but it's actually quite simple. You're going to ask the managed object context for a list of all `Location` objects in the data store, sorted by date.

1. The `NSFetchRequest` is the object that describes which objects you're going to fetch from the data store. To retrieve an object that you previously saved to the data store, you create a fetch request that describes the search parameters of the object — or objects — that you're looking for.
2. Here you tell the fetch request you're looking for `Location` entities.
3. The `NSSortDescriptor` tells the fetch request to sort on the `date` attribute, in ascending order so that the `Location` objects that the user added first will be at the top of the list. You can sort on any attribute here — later on, you'll sort on the `Location`'s `category` as well.

That completes the fetch request. It took a few lines of code, but basically you said: “Get all `Location` objects from the data store and sort them by date.”

4. Now that you have a fetch request, you can tell the context to execute it. The `fetch()` method returns an array with the sorted objects, or throws an error in case something went wrong. That's why this happens inside a `do-try-catch` block.

If everything goes well, you assign the results of the fetch to the `locations` instance variable.

Note: To create the fetch request you wrote `NSFetchRequest<Location>`.

The `< >` mean that `NSFetchRequest` is a *generic*. Recall that arrays are also generics — to create an array you specify the type of objects that go into the



array, either using the shorthand notation `[Location]`, or the longer `Array<Location>`.

To use an `NSFetchRequest`, you need to tell it what type of object you’re going to be fetching. Here, you create an `NSFetchRequest<Location>` so that the result of `fetch()` is an array of `Location` objects.

Now that you’ve loaded the list of `Location` objects into an instance variable, you can change the table view’s data source methods.

- Change the data source methods to:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return locations.count
}

override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
       (withIdentifier: "LocationCell",
         for: indexPath)

    let location = locations[indexPath.row]

    let descriptionLabel = cell.viewWithTag(100) as! UILabel
    descriptionLabel.text = location.locationDescription

    let addressLabel = cell.viewWithTag(101) as! UILabel
    if let placemark = location.placemark {
        var text = ""
        if let s = placemark.subThoroughfare {
            text += s + " "
        }
        if let s = placemark.thoroughfare {
            text += s + ", "
        }
        if let s = placemark.locality {
            text += s
        }
        addressLabel.text = text
    } else {
        addressLabel.text = ""
    }
    return cell
}
```



This should have no surprises for you. You get the `Location` object for the row from the array and then use its properties to fill the labels. Because `placemark` is an optional, you use `if let` to unwrap it.

- Run the app. Now switch to the Locations tab and... crap! It crashes.

The error message should say something like:

```
fatal error: unexpectedly found nil while unwrapping an Optional value
```

Exercise: What did you forget?

Answer: You added a `managedObjectContext` property to `LocationsViewController`, but never gave this property a value. Therefore, there is nothing to fetch `Location` objects from. (If you already noticed this and were like, "How come we are not passing the value from AppDelegate?", good job!)

- Switch to `AppDelegate.swift`. In `application(_:didFinishLaunchingWithOptions:)`, change the `if let tabBarViewControllers` block, as follows:

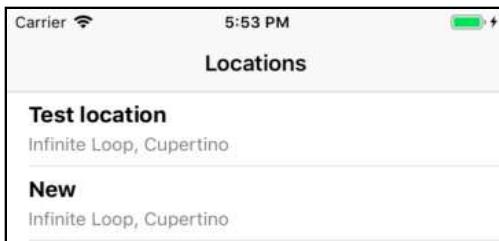
```
if let tabViewControllers = tabController.viewControllers {  
    // First tab  
    var navController = tabViewControllers[0]  
        as! UINavigationController  
    let controller1 = navController.viewControllers.first  
        as! CurrentLocationViewController  
    controller1.managedObjectContext = managedObjectContext  
    // Second tab  
    navController = tabViewControllers[1]  
        as! UINavigationController  
    let controller2 = navController.viewControllers.first  
        as! LocationsViewController  
    controller2.managedObjectContext = managedObjectContext  
}
```

There are a couple of minor changes to the existing code — one is to make `navController` a variable so that it can be re-used for the second tab, and the second is to rename the `controller` constant to `controller1` to separate it from the the second view controller which would be of a different type.

The code for the second tab looks up the `LocationsViewController` in the storyboard and gives it a reference to the managed object context, similar to what you did for the first tab.



- Run the app again and switch to the Locations tab. Core Data properly fetches the objects and displays them:



The list of Locations

Note that the list doesn't update yet if you tag a new location. You have to restart the app to see the new Location object appear. You'll solve this later on.

Creating a custom table view cell subclass

Using `viewWithTag(_:_)` to find the labels from the table view cell works, but it doesn't look very object-oriented to me.

It would be much nicer if you could make your own `UITableViewCell` subclass and give it outlets for the labels. Fortunately, you can, and it's pretty easy!

- Add a new file to the project using the **Cocoa Touch Class** template. Name it **LocationCell** and make it a subclass of `UITableViewCell`. (Make sure that the class name does not change when you set the subclass — that can be a little annoying.)
- Add the following outlets to **LocationCell.swift**, inside the class definition:

```
@IBOutlet weak var descriptionLabel: UILabel!
@IBOutlet weak var addressLabel: UILabel!
```

- Open the storyboard and select the prototype cell that you made earlier. In the **Identity inspector**, set **Class** to **LocationCell**.

- Now you can connect the two labels to the two outlets. This time the outlets are not on the view controller but on the cell, so use the `LocationCell`'s **Connections inspector** to connect the `descriptionLabel` and `addressLabel` outlets.

That is all you need to do to make the table view use your own table view cell class. But, you do need to update `LocationsViewController` to make use of it.

- In **LocationsViewController.swift**, replace `tableView(tableView: cellForRowAtIndexPath)` with the following:

```
override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "LocationCell",
        for: indexPath) as! LocationCell

    let location = locations[indexPath.row]
    cell.configure(for: location)

    return cell
}
```

As before, this asks for a cell using `dequeueReusableCell(withIdentifier:for:)`, but now this will be a `LocationCell` object instead of a regular `UITableViewCell`. That's why you've added the type cast.

Note that the string `LocationCell` is the re-use identifier from the placeholder cell, but `LocationCell` is the class of the actual cell object that you're getting. They have the same name but one is a `String` and the other is a `UITableViewCell` subclass with extra properties.

Once you have the cell reference, you call a new method, `configure(for:)` to put the `Location` object into the table view cell.

- Add this new method to **LocationCell.swift**:

```
// MARK:- Helper Method
func configure(for location: Location) {
    if location.locationDescription.isEmpty {
        descriptionLabel.text = "(No Description)"
    } else {
        descriptionLabel.text = location.locationDescription
    }

    if let placemark = location.placemark {
        var text = ""
        if let s = placemark.subThoroughfare {
            text += s + " "
        }
        if let s = placemark.thoroughfare {
            text += s + ", "
        }
        if let s = placemark.locality {
            text += s
        }
    }
}
```



```
    addressLabel.text = text
} else {
    addressLabel.text = String(format:
        "Lat: %.8f, Long: %.8f", location.latitude,
                                    location.longitude)
}
}
```

Instead of using `viewWithTag(_:)` to find the description and address labels, you now simply use the `descriptionLabel` and `addressLabel` properties of the cell.

- Run the app to make sure everything still works. If you have a location without a description the table cell will now say “(No Description).” If there is no placemark, the address label contains the GPS coordinates.

Using a custom subclass for your table view cells, there is no limit to how complex the cell functionality can be.

Editing locations

You will now connect the `LocationsViewController` to the Location Details screen, so that when you tap a row in the table, it lets you edit that location’s description and category.

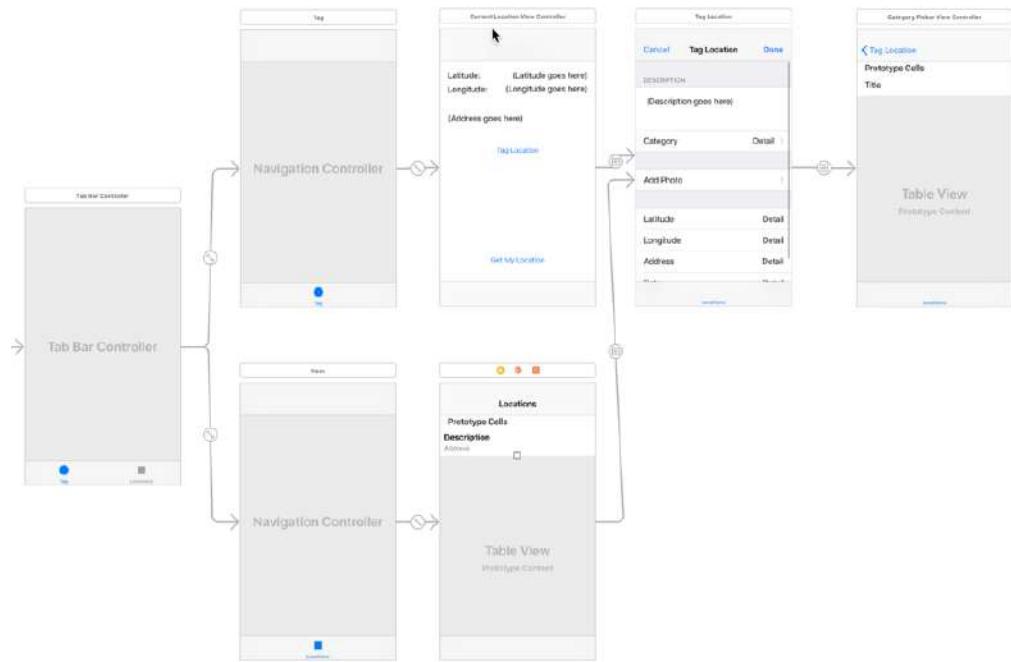
You’ll re-use the `LocationDetailsViewController` but have it edit an existing `Location` object rather than add a new one.

Creating edit segue

- Go to the storyboard. Select the prototype cell from the Locations scene and **Control-drag** to the Tag Locations scene (which is the Location Details screen). Add a **Show** selection segue and name it **EditLocation**.



At this point the storyboard should look like this:



The Location Details screen is now also connected to the Locations screen

There are now two segues from two different screens going to the same view controller.

This is the reason why you should build your view controllers to be as independent of their “calling” controllers as possible. You can then easily re-use them somewhere else in your app.

Soon, you will be calling this same screen from yet another place. In total there will be three segues to it.

► Go to **LocationsViewController.swift** and add the following code:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                     sender: Any?) {
    if segue.identifier == "EditLocation" {
        let controller = segue.destination
            as! LocationDetailsViewController
        controller.managedObjectContext = managedObjectContext

        if let indexPath = tableView.indexPath(for: sender
                                              as! UITableViewCell) {
```

```
        let location = locations[indexPath.row]
        controller.locationToEdit = location
    }
}
```

This method is invoked when the user taps a row in the Locations screen. It figures out which `Location` object belongs to the row and puts it in the new `locationToEdit` property of `LocationDetailsViewController`. This property doesn't exist yet, but you'll add it in a moment.

The Any type

The type of the `sender` parameter is `Any`. You have seen this type in a few places before. What is it?

Objective-C has a special type, `id`, that means “any object.” It’s similar to `NSObject` except that it doesn’t make any assumptions at all about the underlying type of the object. `id` doesn’t have any methods, properties or instance variables, it’s a completely naked object reference.

All objects in an Objective-C program can be treated as having type `id`. As a result, a lot of the APIs from iOS frameworks depend on this special `id` type. This is a powerful feature of Objective-C, but unfortunately, a dynamic type like `id` doesn’t really fit in a *strongly typed* language such as Swift.

Still, we can’t avoid `id` completely because it’s so prevalent in iOS frameworks. The Swift equivalent of `id` is the `Any` type.

The `sender` parameter from `prepare(for:sender:)` can be any kind of object, and so has type `Any` (thanks to the question mark it can also be `nil`).

If the segue is triggered from a table view, `sender` is of type `UITableViewCell`. If triggered from a button, `sender` is of type `UIButton` (or `UIBarButtonItem`), and so on.

Objects that appear as type `Any` are not very useful in that form, and you’ll have to tell Swift what sort of object it really is. In the code that you just wrote, `indexPath(for:)` expects a `UITableViewCell` object, not an `Any` object.



You know that `sender` in this case really is a `UITableViewCell` because the only way to trigger this segue is to tap a table view cell. With the `as!` type cast you're giving Swift your word (scout's honor!) that it can safely interpret `sender` as a `UITableViewCell`.

Of course, if you were to hook up this segue to something else, such as a button, then this assumption is no longer valid and the app will crash.

Setting up the edit view controller

When editing an existing `Location` object, you have to do a few things differently in the `LocationDetailsViewController`. The title of the screen shouldn't be "Tag Location" but "Edit Location." You also must put the values from the existing `Location` object into the various cells.

The value of the new `locationToEdit` property determines whether the screen operates in "add" mode or in "edit" mode.

- Add these properties to `LocationDetailsViewController.swift`:

```
var locationToEdit: Location?  
var descriptionText = ""
```

`locationToEdit` needs to be an optional because in "add" mode it will be `nil`.

- Update `viewDidLoad()` to check whether `locationToEdit` is set:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    if let location = locationToEdit {  
        title = "Edit Location"  
    }  
    . . .  
}
```

If `locationToEdit` is not `nil`, you're editing an existing `Location` object. In that case, the title of the screen becomes "Edit Location."

Note: Xcode gives a warning on the line `if let location = locationToEdit` because you're not using the value of `location` anywhere. If you click the yellow icon, Xcode suggests that you replace it with `if locationToEdit != nil`. You *will* use `location` in a bit, so ignore Xcode's suggestion.

- Also change this line in `viewDidLoad()`:

```
descriptionTextView.text = descriptionText
```

You load the value of the new `descriptionText` variable into the text view.

Now how do you get the values from the `locationToEdit` object into the text view and labels of this view controller? Swift has a really cool **property observer** feature that is perfect for this.

- Change the declaration of the `locationToEdit` property to the following:

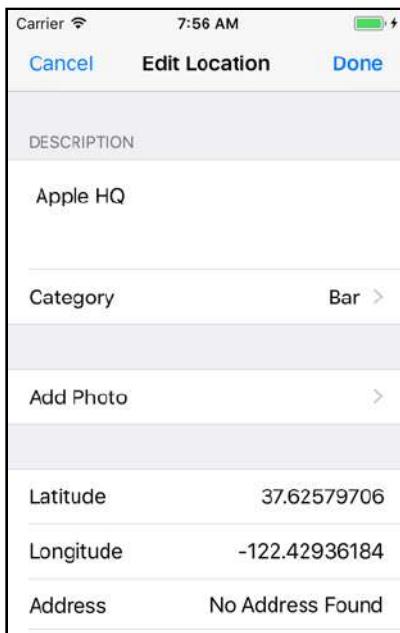
```
var locationToEdit: Location? {
    didSet {
        if let location = locationToEdit {
            descriptionText = location.locationDescription
            categoryName = location.category
            date = location.date
            coordinate = CLLocationCoordinate2DMake(
                location.latitude, location.longitude)
            placemark = location.placemark
        }
    }
}
```

If a variable has a `didSet` block, then the code in this block is performed whenever you put a new value into that variable — very handy!

Here, you take the opportunity to fill in the view controller's instance variables with the `Location` object's values.

Because `prepare(for:sender:)` — and therefore `locationToEdit`'s `didSet` — is called before `viewDidLoad()`, this puts the right values on the screen before it becomes visible.

- Run the app, go to the Locations tab and tap on a row. The Edit Location screen should now appear with the data from the selected location:



Editing an existing location

- Change the description of the location and press Done.

Nothing happened?! Well, that's not quite true. Stop the app and run it again. You will see that a new location has been added with the changed description, but the old one is still there as well.

Fixing the edit screen

There are two problems to solve:

1. When editing an existing location you must save changes to the location instead of creating a new entry.
2. The Locations screen doesn't update to reflect any changes to the data.

The first fix is easy.

- Still in **LocationDetailsViewController.swift**, change the top part of `done()`:

```
@IBAction func done() {
    let hudView = HudView.hud(inView: . . .)

    let location: Location
    if let temp = locationToEdit {
        hudView.text = "Updated"
        location = temp
    } else {
        hudView.text = "Tagged"
        location = Location(context: managedObjectContext)
    }

    location.locationDescription = descriptionTextView.text
    . . .
```

The change is straightforward: you only ask Core Data for a new `Location` object if you don't already have one. You also make the text in the HUD say "Updated" when the user is editing an existing `Location`.

Note: I've been harping on about the fact that Swift requires all non-optional variables and constants to always have a value. But here you declare `let location` without giving it an initial value. What gives?

Well, the `if` statement that follows this declaration always puts a value into `location`, either the unwrapped value of `locationToEdit`, or a new `Location` object obtained from Core Data. After the `if` statement, `location` is guaranteed to have a value. Swift is cool with that.

- Run the app again and edit a location. Now the HUD should say "Updated."
- Stop the app and run it again to verify that the object was indeed properly changed. (You can also look at it directly in the SQLite database, of course.)

Exercise: Why do you think the table view isn't being updated after you change a `Location` object? Tip: Recall that the table view also doesn't update when you tag new locations.

Answer: You fetch the `Location` objects in `viewDidLoad()`. But `viewDidLoad()` is only performed once, when the app starts. After the initial load of the Locations screen, its contents are never refreshed.



The `LocationDetailsViewController` could tell you through delegate methods that a location has been added or changed. But since you're using Core Data, there is a better way to do this.

Using NSFetchedResultsController

As you are no doubt aware by now, table views are everywhere in iOS apps. A lot of the time when you're working with Core Data, you want to fetch objects from the data store and show them in a table view. And when those objects change, you want to do a live update of the table view in response, to show the changes to the user.

So far, you've filled the table view by manually fetching the results, but then you also need to manually check for changes and perform the fetch again to update the table. With `NSFetchedResultsController`, all that manual work is no longer needed.

It works like this: you give `NSFetchedResultsController` a fetch request, just like the `NSFetchRequest` you made earlier, and tell it to go fetch the objects. So far nothing new.

But, you don't put the results from that fetch into your own array. Instead, you read them straight from the fetched results controller. In addition, you make the view controller the delegate for the `NSFetchedResultsController`. Through this delegate, the view controller is informed that objects have been changed, added or deleted so that it can update the table in response.

► In `LocationsViewController.swift`, replace the `locations` instance variable with a new `fetchedResultsController` variable:

```
lazy var fetchedResultsController:  
    NSFetchedResultsController<Location> = {  
    let fetchRequest = NSFetchedResultsController<Location>()  
  
    let entity = Location.entity()  
    fetchRequest.entity = entity  
  
    let sortDescriptor = NSSortDescriptor(key: "date",  
                                         ascending: true)  
    fetchRequest.sortDescriptors = [sortDescriptor]  
  
    fetchRequest.fetchBatchSize = 20  
  
    let fetchedResultsController = NSFetchedResultsController(  
        fetchRequest: fetchRequest,  
        managedObjectContext: self.managedObjectContext,  
        sectionNameKeyPath: nil, cacheName: "Locations")  
}
```

```
fetchedResultsController.delegate = self  
return fetchedResultsController  
}()
```

This again uses the lazy initialization pattern with a closure to set everything up. It's good to get into the habit of lazily loading objects. You don't allocate them until you first use them. This makes your apps quicker to start and it saves memory.

The code in the closure does the same thing that you used to do in `viewDidLoad()`: it makes an `NSFetchRequest` and gives it an entity and a sort descriptor.

Note: Note that the new variable is not just `NSFetchedResultsController` but `NSFetchedResultsController<Location>`, since it's a generic. You need to tell the fetched results controller what type of objects to fetch.

This is new:

```
fetchRequest.fetchBatchSize = 20
```

If you have a huge table with hundreds of objects, then it requires a lot of memory to keep all of these objects around, even though you can only see a handful of them at a time.

The `NSFetchedResultsController` is pretty smart about this and will only fetch the objects that you can actually see, which cuts down on memory usage. This is all done in the background without you having to worry about it. The fetch batch size setting allows you to tweak how many objects will be fetched at a time.

Once the fetch request is set up, you create the star of the show:

```
let fetchedResultsController = NSFetchedResultsController(  
    fetchRequest: fetchRequest,  
    managedObjectContext: self.managedObjectContext,  
    sectionNameKeyPath: nil, cacheName: "Locations")
```

The `cacheName` needs to be a unique name that `NSFetchedResultsController` uses to cache the search results. It keeps this cache around even after your app quits, so the next time the fetch request is lightning fast, as the `NSFetchedResultsController` doesn't have to make a round-trip to the database but can simply read from the cache.

We'll talk about the `sectionNameKeyPath` parameter shortly.



The line that sets `fetchedResultsController.delegate` to `self` currently gives an error message because `LocationsViewController` does not conform to the right delegate protocol yet. You'll fix that in minute.

Now that you have a fetched results controller, you clean up `viewDidLoad()`.

► Change `viewDidLoad()` like this:

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

// MARK:- Helper methods
func performFetch() {
    do {
        try fetchedResultsController.performFetch()
    } catch {
        fatalCoreDataError(error)
    }
}
```

You still perform the initial fetch in `viewDidLoad()`, using the new `performFetch()` helper method. However, if any `Location` objects change after that initial fetch, the `NSFetchedResultsController`'s delegate methods are called to let you know about these changes. I'll show you how in a second.

It's always a good idea to explicitly set the delegate to `nil` when you no longer need the `NSFetchedResultsController`, just so you don't get any more notifications that were still pending.

► For that reason, add a `deinit` method:

```
deinit {
    fetchedResultsController.delegate = nil
}
```

The `deinit` method is invoked when this view controller is destroyed. It may not be strictly necessary to `nil` out the delegate here, but it's a bit of defensive programming that won't hurt.

Note that in this app the `LocationsViewController` will never actually be deallocated because it's one of the top-level view controllers in the tab bar. Still, it's good to get into the habit of writing `deinit` methods.

Because you removed the `locations` array, you should also change the table's data source methods.

► Change `tableView(_:numberOfRowsInSection:)` to:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    let sectionInfo = fetchedResultsController.sections![section]
    return sectionInfo.numberOfObjects
}
```

The fetched results controller's `sections` property returns an array of `NSFetchedResultsSectionInfo` objects that describe each section of the table view. The number of rows is found in the section info's `numberOfObjects` property.

(Currently there is only one section, but later you'll split up the locations by category and then each category gets its own section.)

► Change `tableView(_:cellForRowAt:)` to:

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
       (withIdentifier: "LocationCell",
        for: indexPath) as! LocationCell

    let location = fetchedResultsController.object(at: indexPath)
    cell.configure(for: location)

    return cell
}
```

Instead of looking into the `locations` array like you did before, you now ask the `fetchedResultsController` for the object at the requested index-path. Because it is designed to work closely with table views, `NSFetchedResultsController` knows how to deal with index-paths, so that's very convenient.

► Make the same change in `prepare(for:sender:)`.

There is still one piece of the puzzle missing. You need to implement the delegate methods for `NSFetchedResultsController` in `LocationsViewController`. Let's use an *extension* for that, to keep the code organized.

Organizing the code using extensions

An extension lets you add code to an existing class, without having to modify the original class source code. When you make an extension you say, “here are a bunch of extra methods that also need to go into that class,” and you can do that even if you didn’t write the original class to begin with.

You’ve seen an extension used in `Location+CoreDataProperties.swift`. That was done to make it easier for Xcode to regenerate this file without overwriting the contents of `Location+CoreDataClass.swift`.

You can also use extensions to organize your source code. Here you’ll use an extension just for the `NSFetchedResultsControllerDelegate` methods, so they are not all tangled up with `LocationsViewController`’s other code. By putting this code in a separate unit, you keep the responsibilities separate.

This makes it easy to spot which part of `LocationsViewController` plays the role of the delegate. All the fetched results controller delegate stuff happens just in this extension, not in the main body of the class — you could even place this extension in a separate Swift file if you wanted.

- Add the following code to the bottom of `LocationsViewController.swift`, outside of the class implementation:

```
// MARK:- NSFetchedResultsController Delegate Extension
extension LocationsViewController: NSFetchedResultsControllerDelegate {

    func controllerWillChangeContent(_ controller: NSFetchedResultsController<NSFetchRequestResult>) {
        print("/** controllerWillChangeContent")
        tableView.beginUpdates()
    }

    func controller(_ controller: NSFetchedResultsController<NSFetchRequestResult>, didChange anObject: Any, at indexPath: IndexPath?, for type: NSFetchedResultsChangeType, newIndexPath: IndexPath?) {

        switch type {
        case .insert:
            print("/** NSFetchedResultsChangeInsert (object)")
            tableView.insertRows(at: [newIndexPath!], with: .fade)

        case .delete:
            print("/** NSFetchedResultsChangeDelete (object)")
            tableView.deleteRows(at: [indexPath!], with: .fade)
        }
    }
}
```

```
case .update:
    print("/** NSFetchedResultsChangeUpdate (object)")
    if let cell = tableView.cellForRow(at: indexPath!)
        as? LocationCell {
        let location = controller.object(at: indexPath!)
            as! Location
        cell.configure(for: location)
    }

case .move:
    print("/** NSFetchedResultsChangeMove (object)")
    tableView.deleteRows(at: [indexPath!], with: .fade)
    tableView.insertRows(at: [newIndexPath!], with: .fade)
}

@unknown default:
    fatalError("Unhandled switch case of
NSFetchedResultsChangeType")
}

func controller(_ controller:
    NSFetchedResultsController<NSFetchRequestResult>,
    didChange sectionInfo: NSFetchedResultsSectionInfo,
    atSectionIndex sectionIndex: Int,
    for type: NSFetchedResultsChangeType) {
switch type {
case .insert:
    print("/** NSFetchedResultsChangeInsert (section)")
    tableView.insertSections(IndexSet(integer: sectionIndex),
                            with: .fade)
case .delete:
    print("/** NSFetchedResultsChangeDelete (section)")
    tableView.deleteSections(IndexSet(integer: sectionIndex),
                            with: .fade)
case .update:
    print("/** NSFetchedResultsChangeUpdate (section)")
case .move:
    print("/** NSFetchedResultsChangeMove (section)")
}
@unknown default:
    fatalError("Unhandled switch case of
NSFetchedResultsChangeType")
}

func controllerDidChangeContent(_ controller:
    NSFetchedResultsController<NSFetchRequestResult>) {
    print("/** controllerDidChangeContent")
    tableView.endUpdates()
}
}
```

Yowza, that's a lot of code. Don't let this freak you out! This is the standard way of implementing these delegate methods. For many apps, this exact code will suffice and you can simply copy it over. Look it over for a few minutes to see if this code makes sense to you. You've made it this far, so I'm sure it won't be too hard.

`NSFetchedResultsController` will invoke these methods to let you know that certain objects were inserted, removed, or just updated. In response, you call the corresponding methods on the `UITableView` to insert, remove or update rows. That's all there is to it.

With the `print()` statements in these methods you can follow along in the Console as to what is happening. Also note that you're using the `switch` statement here. A series of `if`'s would have worked just as well but `switch` reads better.

- Run the app. Edit an existing location and press the Done button.

The debug area now shows:

```
*** controllerWillChangeContent
*** NSFetchedResultsControllerChangeUpdate (object)
*** controllerDidChangeContent
```

`NSFetchedResultsController` noticed that an existing object was updated and, through updating the table, called your `cell.configure(for:)` method to redraw the contents of the cell. By the time the Edit Location screen disappears from sight, the table view is updated and your change is visible.

This also works for adding new locations.

- Tag a new location and press the Done button.

The debug area says:

```
*** controllerWillChangeContent
*** NSFetchedResultsControllerChangeInsert (object)
*** controllerDidChangeContent
```

This time it's an "insert" notification. The delegate methods tell the table view to do `insertRows(at:with:)` in response and the new `Location` object is inserted in the table.

That's how easy it is. You make a new `NSFetchedResultsController` object with a fetch request and implement the delegate methods.

The fetched results controller keeps an eye on any changes that you make to the data store and notifies its delegate in response.



It doesn't matter where in the app you make these changes, they can happen on any screen. When that screen saves the changes to the managed object context, the fetched results controller picks up on it right away.

"It's not a bug, it's an undocumented feature"

There is a nasty Core Data bug that has been there for the last few iOS versions. Here is how you can reproduce it:

1. Quit the app.
2. Run the app again and tag a new location.
3. Switch to the Locations tab.

You'd expect the new location to appear in the Locations tab, but it doesn't.

The error message is:

```
CoreData: FATAL ERROR: The persistent cache of section  
information does not match the current configuration. You have  
illegally mutated the NSFetchedResultsController's fetch  
request, its predicate, or its sort descriptor without either  
disabling caching or using +deleteCacheWithName:
```

We did no such thing! Interestingly, this problem does not occur when you switch to the Locations tab before you tag the new location.

There are two possible fixes:

1. You can delete the cache of the NSFetchedResultsController. To do this, add the following line to `viewDidLoad()` before the call to `performFetch()`:

```
NSFetchedResultsController<Location>.deleteCache(withName:  
"Locations")
```

This is not a great solution because it negates the point of having a cache in the first place.

2. You can force the LocationsViewController to load its view immediately when the app starts up. Without this, it delays loading the view until you switch tabs, causing Core Data to get confused. To apply this fix, add the following to `application(_:didFinishLaunchingWithOptions:)`, immediately below the line that sets `controller2.managedObjectContext`:

```
let _ = controller2.view
```



If this problem affects you, then implement one of the above solutions — my suggestion is option #2. Then throw away DataModel.sqlite and run the app again. Verify that the bug no longer occurs.

iOS is pretty great but unfortunately it's not free of bugs — what software is?. If you encounter what you perceive to be a bug in one of the iOS frameworks, then report it at bugreport.apple.com. Feel free to report this Core Data bug as practice.

Deleting locations

Everyone makes mistakes. So, it's likely that users will want to delete locations from their list at some point. This is a very easy feature to add: you just have to remove the Location object from the data store and the NSFetchedResultsController will make sure it gets dropped from the table — again, through its delegate methods.

- Add the following method to **LocationsViewController.swift** (under the table view delegate section):

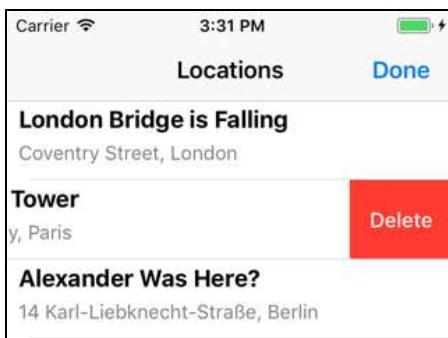
```
override func tableView(_ tableView: UITableView,
                      commit editingStyle: UITableViewCell.EditingStyle,
                      forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        let location = fetchedResultsController.object(at: indexPath)
        managedObjectContext.delete(location)
        do {
            try managedObjectContext.save()
        } catch {
            fatalError("Core Data error")
        }
    }
}
```

You've seen `tableView(_:commit:forRowAt:)` before. It's part of the table view's data source protocol. As soon as you implement this method in your view controller, it enables swipe-to-delete.

This method gets the Location object from the selected row and then tells the context to delete that object. This will trigger the NSFetchedResultsController to send a notification to the delegate, which then removes the corresponding row from the table. That's all you need to do!

- Run the app and remove a location using swipe-to-delete. The Location object is dropped from the database and its row disappears from the screen with a brief animation.





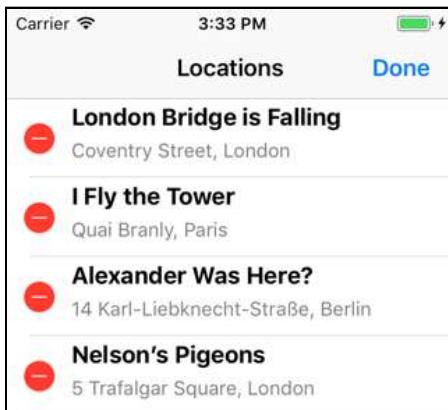
Swipe to delete rows from the table

Many apps have an Edit button in the navigation bar that triggers a mode that also lets you delete — and sometimes move — rows. This is extremely easy to add.

- Add the following line to `viewDidLoad()` in `LocationsViewController.swift`:

```
navigationItem.rightBarButtonItem = editButtonItem
```

That's all there is to it. Every view controller has a built-in Edit button that can be accessed through the `editButtonItem` property. Tapping that button puts the table in editing mode:



The table view in edit mode

- Run the app and verify that you can now also delete rows by pressing the Edit button.

Pretty sweet, huh? There's more cool stuff that `NSFetchedResultsController` makes really easy, such as splitting up the rows into sections.

Table view sections

The Location objects have a category field. It would be nice to group the locations by category in the table. The table view supports organizing rows into sections and each of these sections can have its own header. Putting your rows into sections is a lot of work if you're doing it by hand, but NSFetchedResultsController practically gives you section support for free.

- Change the creation of the sort descriptors in the fetchedResultsController initialization block:

```
lazy var fetchedResultsController: . . . = {  
    . . .  
    let sort1 = NSSortDescriptor(key: "category", ascending: true)  
    let sort2 = NSSortDescriptor(key: "date", ascending: true)  
    fetchRequest.sortDescriptors = [sort1, sort2]  
    . . .
```

Instead of one sort descriptor object, you now have two. First you sort the Location objects by category and inside each of the category groups you sort by date.

- Also change the initialization of the NSFetchedResultsController object:

```
let fetchedResultsController = NSFetchedResultsController(  
    fetchRequest: fetchRequest,  
    managedObjectContext: self.managedObjectContext,  
    sectionNameKeyPath: "category", // change this  
    cacheName: "Locations")
```

The only difference here is that the sectionNameKeyPath parameter is set to "category", which means the fetched results controller will group the search results based on the value of the category attribute.

You're not done yet — the table view's data source also has methods for sections. So far you've only used the methods for rows, but now that you're adding sections to the table, you need to implement a few additional methods.

- Add the following methods to the table view delegate section:

```
override func numberOfSections(in tableView: UITableView)  
    -> Int {  
    return fetchedResultsController.sections!.count  
}  
  
override func tableView(_ tableView: UITableView,  
    titleForHeaderInSection section: Int) -> String? {  
    let sectionInfo = fetchedResultsController.sections![section]
```



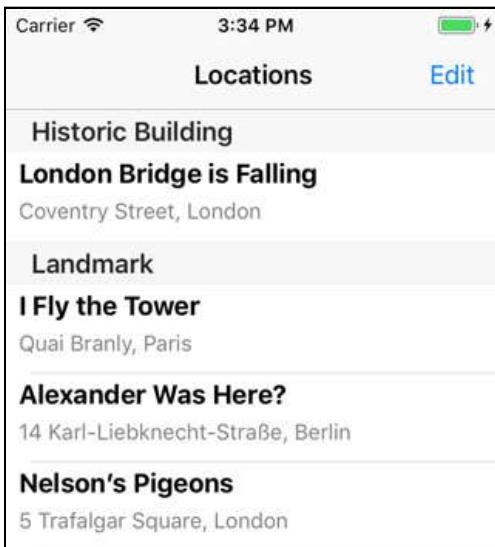
```
    return sectionInfo.name  
}
```

Because you let `NSFetchedResultsController` do all the work already, the implementation of these methods is very simple. You ask the fetcher object for a list of the sections, which is an array of `NSFetchedResultsSectionInfo` objects, and then look inside that array to find out how many sections there are and what their names are.

Exercise: Why do you need to write `sections!` with an exclamation point?

Answer: the `sections` property is an optional, so it needs to be unwrapped before you can use it. Here you know for sure that `sections` will never be `nil` — after all, you just told `NSFetchedResultsController` to group the search results based on the value of their “category” field — so you can safely force unwrap it using the exclamation mark. Are you starting to get the hang of these optionals already?

► Run the app. Play with the categories on the Locations tab and notice how the table view automatically updates. All thanks to `NSFetchedResultsController`!



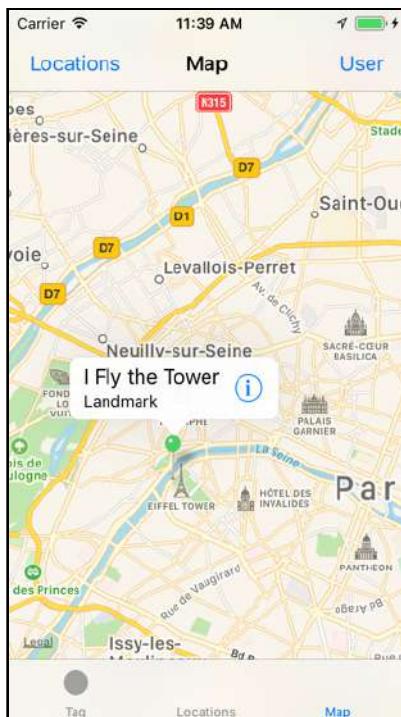
The locations are now grouped in sections

You can find the project files for this chapter under **33 – Locations Tab** in the Source Code folder.

Chapter 34: Maps

Eli Ganim

Showing the locations in a table view is useful, but not very visually appealing. Given that the iOS SDK comes with an awesome map view control, it would be a shame not to use it! In this chapter, you will add a third tab to the app that will look like this when you are finished:



The completed Map screen



This is what you'll do in this chapter:

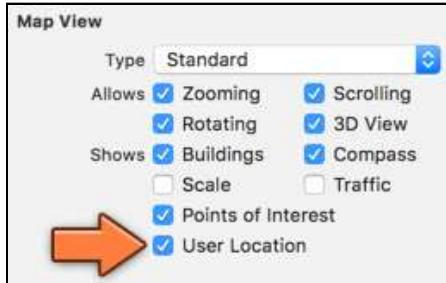
- **Add a map view:** Learn how to add a map view to your app and get it to show the current user location or pins for a given set of locations.
- **Make your own pins:** Learn to create custom pins to display information about points on a map.

Adding a map view

First visit: the storyboard.

- From the Objects Library, drag a **View Controller** on to the canvas.
- Control-drag from the Tab Bar Controller to this new View Controller to add it to the tabs (choose **Relationship segue – view controllers**).
- The new view controller now has a **Tab Bar Item**. Change its title to **Map** (via the Attributes inspector).
- Drag a **Map Kit View** into the view controller. Make it cover the entire area of the screen, so that the lower part of the map view sits under the tab bar. (The size of the Map View should be 320×568 points.)
- Add left, top, right, and bottom Auto Layout constraints to the Map View via the **Add New Constraints** menu, pinning it to the main view.
- In the **Attributes inspector** for the Map View, enable **Shows: User Location**.

That will put a blue dot on the map at the user's current coordinates.

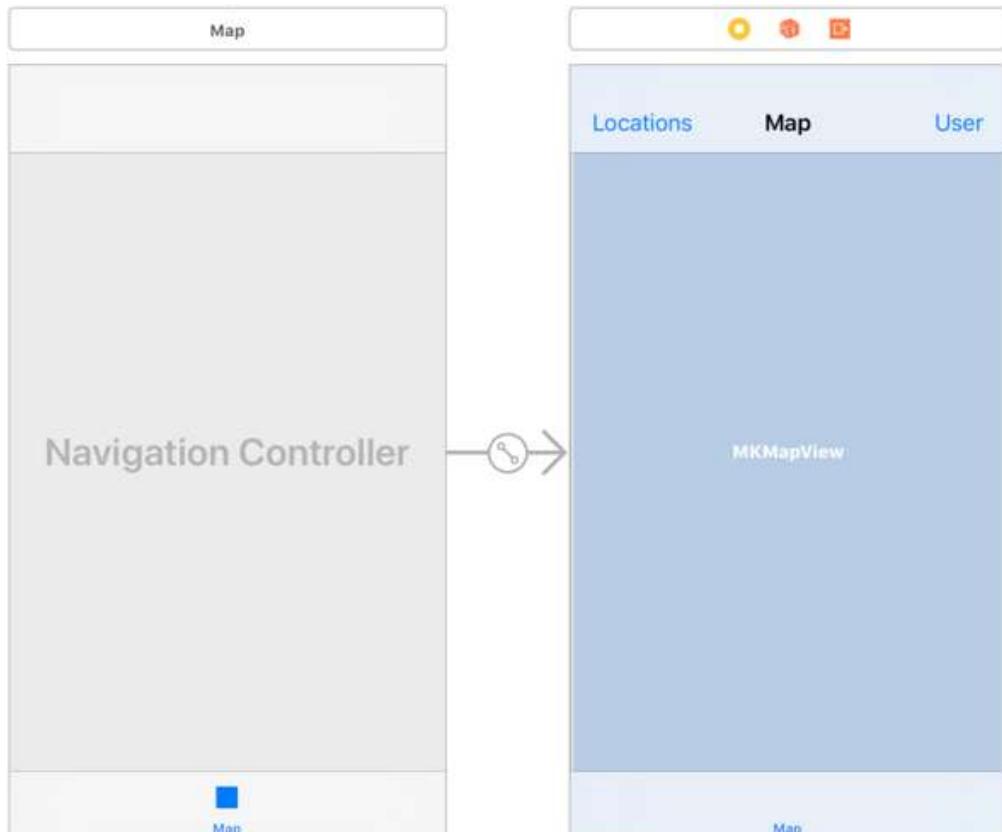


Enable show user location for the Map View

- Select the new view controller and select **Editor** ▶ **Embed In** ▶ **Navigation Controller**. This wraps your view controller in a navigation controller, and makes the new navigation controller the view controller displayed by the Tab Bar Controller.

- ▶ Change the view controller's (not the new navigation controller, but its root view controller) Navigation Item title to **Map**.
- ▶ Drag a **Bar Button Item** into the left-hand slot of the navigation bar and set the title to **Locations**. Drag another into the right-hand slot and set its title to **User**. Later on you'll use nice icons for these buttons, but for now these labels will do.

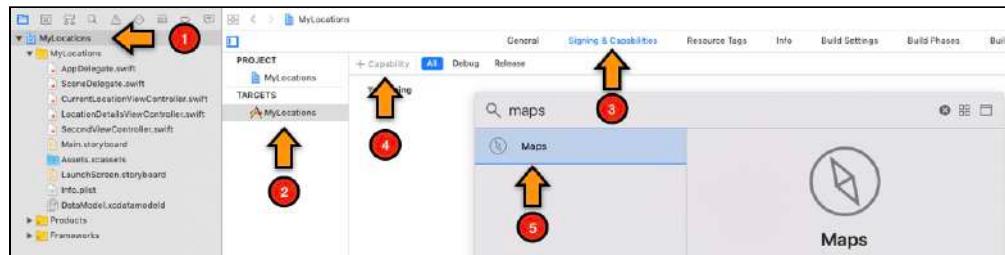
This part of the storyboard should look like this:



The design of the Map screen

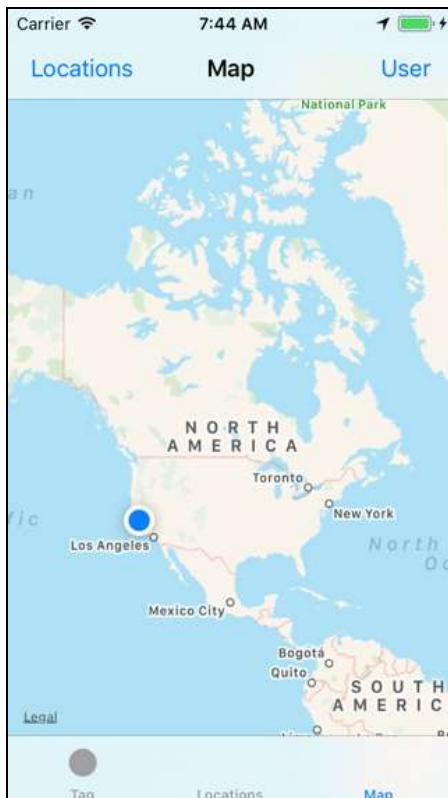
In older versions of Xcode, the app would compile without any problems at this point, but would crash when you switched to the Map tab. This does not appear to be the case with the latest version of Xcode, but if you do run into this issue, here's what you need to do.

- Go to the **Project Settings** screen and select the **Signing & Capabilities** tab.
Click on the **+ Capability** button. Search for *maps* and double click the **Maps** capability to add it.



Enabling the app to use maps

- Run the app. Choose a location in Simulator's Debug menu and switch to the Map. The screen should look something like this – the blue dot shows the current location:



The map shows the user's location

Sometimes, the map might show a different location than the current user location and you might not see the blue dot. If that happens, you can pan the map by clicking the mouse and dragging it across the simulator window. Also, to zoom in or out, hold down the Alt/Option key while dragging the mouse.

Zooming in

Next, you’re going to show the user’s location in a little more detail because that blue dot could be almost anywhere in California!

- Add a new Swift source file to the project and name it **MapViewController**.
- Replace the contents of **MapViewController.swift** with the following:

```
import UIKit
import MapKit
import CoreData

class MapViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView!

    var managedObjectContext: NSManagedObjectContext!

    // MARK:- Actions
    @IBAction func showUser() {
        let region = MKCoordinateRegion(
            center: mapView.userLocation.coordinate,
            latitudinalMeters: 1000, longitudinalMeters: 1000)
        mapView.setRegion(mapView.regionThatFits(region),
                           animated: true)
    }

    @IBAction func showLocations() {
    }
}

extension MapViewController: MKMapViewDelegate {
```

This is a standard view controller — not one of the specialized types like a table view controller. It has an outlet for the map view and two action methods that will be connected to the buttons in the navigation bar. The view controller is also the delegate of the map view, courtesy of the extension.

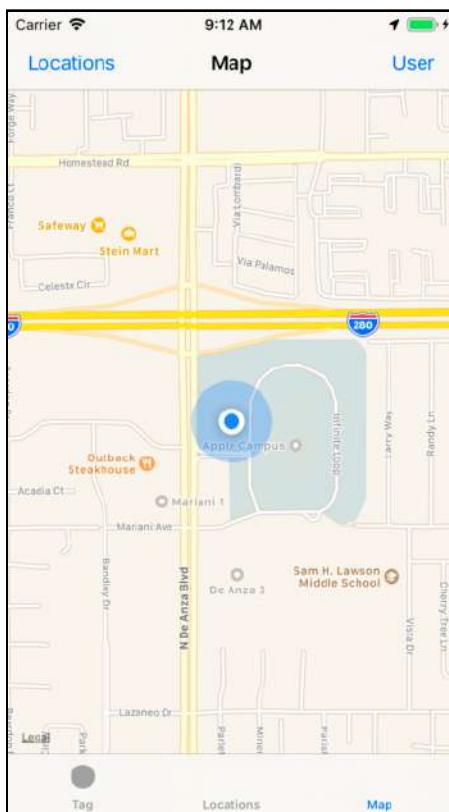
- In the storyboard, select the Map scene (the one with the view controller, not the one with the navigation controller) and in the **Identity inspector** set its **Class** to **MapViewController**.



- ▶ Connect the Locations button to the `showLocations` action and the User button to the `showUser` action. (In case you forgot how, Control-drag from the button to the yellow circle for the view controller.)
- ▶ Connect the Map View with the `mapView` outlet (Control-drag from the view controller to the Map View), and its delegate with the view controller (Control-drag the other way around).

Currently the view controller only implements the `showUser()` action method. When you press the **User** button, it zooms in to the map to a region that is 1000 by 1000 meters (a little more than half a mile in both directions) around the user's position.

Try it out:



Pressing the User button zooms in to the user's location

Showing pins for locations

The other button, Locations, is going to show the region that contains all the user's saved locations. Before you can do that, you first have to fetch those locations from the data store.

Even though this screen doesn't have a table view, you could still use an `NSFetchedResultsController` object to handle all the fetching and automatic change detection. But this time, we're going to do this the hard way – you'll do the fetching by hand.

- Add a new array to `MapViewController.swift`:

```
var locations = [Location]()
```

- Also add this new method:

```
// MARK:- Helper methods
func updateLocations() {
    mapView.removeAnnotations(locations)

    let entity = Location.entity()

    let fetchRequest = NSFetchedResultsController<Location>()
    fetchRequest.entity = entity

    locations = try! managedObjectContext.fetch(fetchRequest)
    mapView.addAnnotations(locations)
}
```

The fetch request is nothing new, except this time you're not sorting the `Location` objects. The order of the `Location` objects in the array doesn't really matter to the map view – only their latitude and longitude coordinates matters.

You've already seen how to handle errors with a `do-try-catch` block. But if you're certain that a particular method call will never fail, you can dispense with the `do` and `catch` and just write `try!` with an exclamation point. As with other things in Swift that have exclamation points, if it turns out that you were wrong, the app will crash without mercy. But in this case there isn't much that can go wrong. So, you can choose to live a little more dangerously.

Once you've obtained the `Location` objects, you call `mapView.addAnnotations()` to add a pin for each location on the map.

The idea is that `updateLocations()` will be executed every time there is a change in the data store. How you'll do that is of later concern, but the point is that when this



happens, the `locations` array may already exist and may contain `Location` objects. If so, you first remove the pins for these old objects with `removeAnnotations()`.

Xcode says the lines with `mapView.addAnnotations()` and `removeAnnotations()` have errors. This is to be expected and you'll fix it in a minute.

► First, add the `viewDidLoad()` method:

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

This fetches the `Location` objects and shows them on the map when the view loads. Nothing special here.

Before this class can use the `managedObjectContext`, you have to give it a reference to that object first. As before, that happens in `AppDelegate`.

► In `AppDelegate.swift`, extend

`application(_:didFinishLaunchingWithOptions:)` to pass the context object to the `MapViewController` as well. This goes inside the `if let` statement:

```
// Third tab
navController = tabViewControllers[2] as! UINavigationController
let controller3 = navController.viewControllers.first
    as! MapViewController
controller3.managedObjectContext = managedObjectContext
```

You're not quite done yet. In `updateLocations()` you told the map view to add the `Location` objects as annotations — an annotation is a pin on the map — but `MKMapView` expects an array of `MKAnnotation` objects, not your own `Location` class.

Luckily, `MKAnnotation` is a protocol. So, you can turn the `Location` objects into map annotations by making the class conform to that protocol.

► Change the `class` line from `Location+CoreDataClass.swift` to:

```
public class Location: NSManagedObject, MKAnnotation {
```

Just because `Location` is an object that is managed by Core Data doesn't mean you can't add your own stuff to it. It's still an object!

Exercise: Xcode now says “Use of undeclared type `MKAnnotation`.” Why is that?

Answer: You still need to import MapKit. Add that line at the top of the file.

Exercise: Xcode still shows an error about the class not conforming to the MKAnnotation protocol. What is wrong now?

Answer: You said Location conforms to the MKAnnotation protocol — you have to provide all the required features from that protocol in the Location class. Xcode makes this easy since it provides a "Fix" option to add protocol stubs.

Note: If you use the "Fix" option, you'll still get errors since the stubs are just that — empty placeholders. So you still have to actually do some work to flesh things out.

The MKAnnotation protocol requires the class to implement the coordinate property. There are two other properties — title and subtitle — which are optional, but we'll implement those as well.

The annotation needs to know the coordinate in order to place the pin in the correct place on the map. The title and subtitle are used to display additional information about the location for each pin.

► Add the following code to **Location+CoreDataClass.swift**:

```
public var coordinate: CLLocationCoordinate2D {
    return CLLocationCoordinate2DMake(latitude, longitude)
}

public var title: String? {
    if locationDescription.isEmpty {
        return "(No Description)"
    } else {
        return locationDescription
    }
}

public var subtitle: String? {
    return category
}
```

Do you notice anything special here? All three items are instance variables — because of var — but they also have a block of source code associated with them.



These variables are **read-only computed properties**. That means they don't actually store a value in a memory location. Whenever you access the `coordinate`, `title`, or `subtitle` variables, they perform the logic from their code blocks. That's why they are *computed* properties: they compute something.

These properties are read-only because they only return a value — you can't assign them a new value using the assignment operator.

The following is OK because it reads the value of the property:

```
let s = location.title
```

But you cannot do this:

```
location.title = "Time for a change"
```

The only way the `title` property can change is if the `locationDescription` value changes. You could also have written this as a method:

```
func title() -> String? {
    if locationDescription.isEmpty {
        return "(No Description)"
    } else {
        return locationDescription
    }
}
```

This is equivalent to using the computed property. Whether to use a method or a computed property is often a matter of taste and you'll see both ways used throughout the iOS frameworks. By the way, it is also possible to make *read-write* computed properties that *can* be changed, but the `MKAnnotation` protocol doesn't use those.

One more thing that you might have noticed about the variables above is the fact that they all have a `public` attribute. You've never used a `public` attribute for variables before. So why here?

That's because the `MKAnnotation` protocol declares all three properties as `public`. You have to match the protocol declaration exactly and so your properties must have the `public` attribute as well. If you don't, Xcode will start whining! Try removing the `public` attribute from one variable and see what happens...

- Run the app and switch to the Map screen. It should now show pins for all the saved locations. Below each pin you should see the value of the `title` property from the `MKAnnotation` protocol.





The map shows pins for the saved locations

If you tap on a pin, the category for the location, which comes from the `subtitle` property, would be added below the title while the pin itself would scale up to indicate that it is currently selected.

Note: So far, all the protocols you've seen were used for making delegates. But that's not the case here — `Location` is not a delegate of anything.

The `MKAnnotation` protocol simply lets you pretend that `Location` is an annotation that can be placed on a map view. You can use this trick with any object you want; as long as the object implements the `MKAnnotation` protocol, it can be shown on a map.

Protocols let objects wear different hats.

Showing a region

Tapping the User button makes the map zoom to the user's current coordinates, but the same thing doesn't happen yet for the location pins.

By looking at the highest and lowest values for the latitude and longitude of all the Location objects, you can calculate a region and then tell the map view to zoom to that region.

► In **MapViewController.swift**, add the following new method:

```
func region(for annotations: [MKAnnotation]) ->
    MKCoordinateRegion {
    let region: MKCoordinateRegion

    switch annotations.count {
    case 0:
        region = MKCoordinateRegion(
            center: mapView.userLocation.coordinate,
            latitudinalMeters: 1000, longitudinalMeters: 1000)

    case 1:
        let annotation = annotations[annotations.count - 1]
        region = MKCoordinateRegion(
            center: annotation.coordinate,
            latitudinalMeters: 1000, longitudinalMeters: 1000)

    default:
        var topLeft = CLLocationCoordinate2D(latitude: -90,
                                              longitude: 180)
        var bottomRight = CLLocationCoordinate2D(latitude: 90,
                                                longitude: -180)

        for annotation in annotations {
            topLeft.latitude = max(topLeft.latitude,
                                  annotation.coordinate.latitude)
            topLeft.longitude = min(topLeft.longitude,
                                   annotation.coordinate.longitude)
            bottomRight.latitude = min(bottomRight.latitude,
                                       annotation.coordinate.latitude)
            bottomRight.longitude = max(bottomRight.longitude,
                                         annotation.coordinate.longitude)
        }

        let center = CLLocationCoordinate2D(
            latitude: topLeft.latitude -
                (topLeft.latitude - bottomRight.latitude) / 2,
            longitude: topLeft.longitude -
                (topLeft.longitude - bottomRight.longitude) / 2)

        let extraSpace = 1.1
    }
}
```



```
let span = MKCoordinateSpan(  
    latitudeDelta: abs(topLeft.latitude -  
                        bottomRight.latitude) * extraSpace,  
    longitudeDelta: abs(topLeft.longitude -  
                        bottomRight.longitude) * extraSpace)  
  
    region = MKCoordinateRegion(center: center, span: span)  
}  
  
return mapView.regionThatFits(region)  
}
```

`region(for:)` has three situations to handle. It uses a `switch` statement to look at the number of annotations and then chooses the corresponding case:

1. There are no annotations. You center the map on the user's current position.
2. There is only one annotation. You center the map on that one annotation.
3. There are two or more annotations. You calculate the extent of their reach and add a little padding. See if you can make sense of those calculations. The `max()` function looks at two values and returns the larger of the two; `min()` returns the smaller; `abs()` always makes a number positive — absolute value.

Note that this method does not use `Location` objects for anything. It assumes that all the objects in the array conform to the `MKAnnotation` protocol and it only looks at that part of the object. As far as `region(for:)` is concerned, what it deals with are annotations. It just so happens that these annotations are represented by your `Location` objects.

That is the power of using protocols. It also allows you to use this method in any app that uses Map Kit, without modifications. Pretty neat.

► Add the following code to `showLocations()`:

```
@IBAction func showLocations() {  
    let theRegion = region(for: locations)  
    mapView.setRegion(theRegion, animated: true)  
}
```

This calls `region(for:)` to calculate a reasonable region that fits all the `Location` objects and then sets that region on the map view.

► Finally, change `viewDidLoad()`:

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

```
if !locations.isEmpty {  
    showLocations()  
}  
}
```

It's a good idea to show the user's locations the first time you switch to the Map tab. So `viewDidLoad()` calls `showLocations()` if the user has any saved locations.

- Run the app and switch to the Map tab, the map view should be zoomed in on your saved locations — because you have the code in `viewDidLoad`, remember? (This only works well if the locations aren't too far apart, of course.)



The map view zooms in to fit all your saved locations

Making your own pins

You made the `MapViewController` conform to the `MKMapViewDelegate` protocol, but so far, you haven't done anything with that.



This delegate is useful for creating your own annotation views. Currently, a default pin is displayed with a title below it, but you can change this to anything you like.

Creating custom annotations

- Add the following code to the extension at the bottom of **MapViewController.swift**:

```
func mapView(_ mapView: MKMapView,
    viewFor annotation: MKAnnotation) ->
    MKAnnotationView? {
    // 1
    guard annotation is Location else {
        return nil
    }
    // 2
    let identifier = "Location"
    var annotationView = mapView.dequeueReusableCell(withIdentifier: identifier)
    if annotationView == nil {
        let pinView = MKPinAnnotationView(annotation: annotation,
            reuseIdentifier: identifier)
        // 3
        pinView.isEnabled = true
        pinView.canShowCallout = true
        pinView.animatesDrop = false
        pinView.pinTintColor = UIColor(red: 0.32, green: 0.82,
            blue: 0.4, alpha: 1)

        // 4
        let rightButton = UIButton(type: .detailDisclosure)
        rightButton.addTarget(self,
            action: #selector(showLocationDetails(_:)),
            for: .touchUpInside)
        pinView.rightCalloutAccessoryView = rightButton
        annotationView = pinView
    }

    if let annotationView = annotationView {
        annotationView.annotation = annotation

        // 5
        let button = annotationView.rightCalloutAccessoryView
            as! UIButton
        if let index = locations.firstIndex(of: annotation
            as! Location) {
            button.tag = index
        }
    }
}
```

```
    return annotationView  
}
```

This is very similar to what a table view data source does in `cellForRowAtIndexPath`, except that you're not dealing with table view cells here but with `MKAnnotationView` objects. This is what happens step-by-step :

1. Because `MKAnnotation` is a protocol, there may be other objects apart from the `Location` object that want to be annotations on the map. An example is the blue dot that represents the user's current location.

You should leave such annotations alone. So, you use the special `is` type check operator to determine whether the annotation is really a `Location` object. If it isn't, you return `nil` to signal that you're not making an annotation for this other kind of object. The guard statement you're using here works like an `if`: it only continues if the condition — `annotation is Location` — is true.

2. This is similar to creating a table view cell. You ask the map view to re-use an annotation view object. If it cannot find a recyclable annotation view, then you create a new one.

Note that you're not limited to using `MKPinAnnotationView` for your annotations. This is the standard annotation view class, but you can also create your own `MKAnnotationView` subclass and make it look like anything you want. Pins are only one option.

3. This sets some properties to configure the look and feel of the annotation view. Previously the pins were red, but you make them green here.
4. This is where it gets interesting. You create a new `UIButton` object that looks like a detail disclosure button — ①. You use the target-action pattern to hook up the button's "Touch Up Inside" event with a new method `showLocationDetails()`, and add the button to the annotation view's accessory view.
5. Once the annotation view is constructed and configured, you obtain a reference to that detail disclosure button again and set its `tag` to the index of the `Location` object in the `locations` array. That way, you can find the `Location` object later in `showLocationDetails()` when the button is pressed.

► Add the `showLocationDetails()` method but leave it empty for now. Put it in the main class, not the extension.

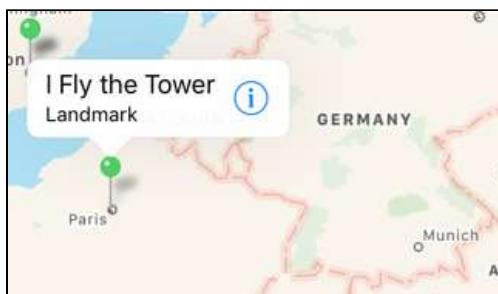
```
@objc func showLocationDetails(_ sender: UIButton) {  
}
```



Because you've told the button its `#selector` is `showLocationDetails`, the app won't compile unless you add at least an empty version of this method.

This method takes one parameter, `sender`, that refers to the control that sent the action message. In this case, the sender will be the ⓘ button. That's why the type of the `sender` parameter is `UIButton`.

► Run the app. The pins don't look the same as the standard pins from before, and are green. There's no title below each pin, but there's a callout when you tap a pin, and the callout has a custom button. If the pins don't change, then make sure you connected the view controller as the delegate of the map view in the storyboard.



The annotations use your own view

Guard

In the map view delegate method, you wrote the following:

```
guard annotation is Location else {  
    return nil  
}
```

The guard statement lets you try something. If the result is `nil` or `false`, the code from the `else` block is performed.

If everything works like it's supposed to, the code simply skips the `else` block and continues.

You could also have written it as follows:

```
if annotation is Location {  
    // do all the other things  
}  
else {  
    return nil  
}
```

This uses the familiar `if` statement. But notice how the code that handles the situation when annotation is *not* a Location is now all the way at the bottom of the method. If you have several of these `if` statements, your code ends up looking like this:

```
if condition1 {  
    if condition2 {  
        if condition3 {  
            . . .  
        } else {  
            return nil // condition3 is false  
        }  
    } else {  
        return nil // condition2 is false  
    }  
} else {  
    return nil // condition1 is false  
}
```

This kind of structure is known as the “Pyramid of Doom.” There’s nothing wrong with it per se, but it can make the program flow hard to decipher. With `guard` you can write this as:

```
guard condition1 else {  
    return nil // condition1 is false  
}  
guard condition2 else {  
    return nil // condition2 is false  
}  
guard condition3 else {  
    return nil // condition3 is false  
}  
. . .
```

Now all the conditions are checked first and any errors or unexpected situations are handled straight away. Many programmers find this easier to read.

Adding annotation actions

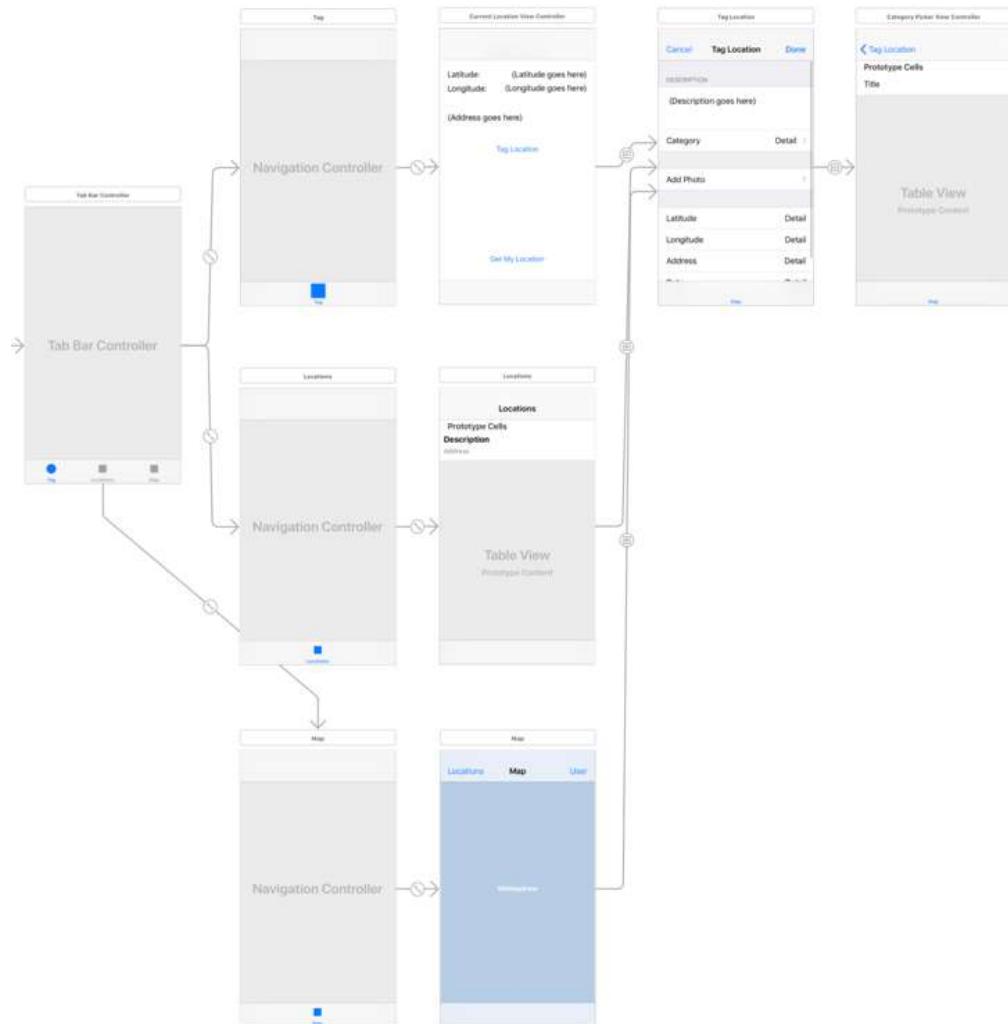
Tapping a pin on the map now brings up a callout with a blue ⓘ button. What should this button do? Show the Edit Location screen, of course!

- Open the storyboard. Find the Map View Controller, and **Control-drag** from the yellow circle at the top to the Tag Location scene, which is the Location Details View Controller.

Make this a new **Show** segue named **EditLocation**.

Tip: If making this connection gives you problems because the storyboard won't fit on your screen, then try Control-dragging from (or to) the Document Outline. You can also zoom out to show more of the storyboard.

The storyboard should now look something like this:



The Location Details screen is connected to all three screens

It's hard to see clearly at this level of zoom, but you should see that there are now three segues going to the Tag Location scene.

- Back in **MapViewController.swift**, change `showLocationDetails(_:)` to trigger the segue:

```
func showLocationDetails(sender: UIButton) {
    performSegue(withIdentifier: "EditLocation", sender: sender)
}
```

Because the segue isn't connected to any particular control in the view controller, you have to perform the segue manually. You pass along the button object as the `sender`, so you can read its `tag` property later.

- Add the `prepare(for:sender:)` method:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                     sender: Any?) {
    if segue.identifier == "EditLocation" {
        let controller = segue.destination
            as! LocationDetailsViewController
        controller.managedObjectContext = managedObjectContext

        let button = sender as! UIButton
        let location = locations[button.tag]
        controller.locationToEdit = location
    }
}
```

This is very similar to what you did in the Locations screen, except that now you get the `Location` object to edit from the `locations` array, using the `tag` property of the `sender` button as the index in that array.

- Run the app, tap on a pin and edit the location.

It works, except... the annotation's callout doesn't change until you tap the pin again. Likewise, changes on the other screens, such as adding or deleting a location, have no effect on the map.

This is the same problem you had earlier with the Locations screen. Because the list of `Location` objects is only fetched once in `viewDidLoad()`, any changes that happen afterwards are overlooked.

Live-updating annotations

The way you're going to fix this for the Map screen is by using notifications. Recall that you have already put `NotificationCenter` to use for dealing with Core Data save errors.

As it happens, Core Data also sends out a bunch of notifications when changes are made to the data store. You can subscribe to these notifications and update the map view when you receive them.

- In `MapViewController.swift`, change the `managedObjectContext` property declaration to:

```
var managedObjectContext: NSManagedObjectContext! {
    didSet {
        NotificationCenter.default.addObserver(forName:
            Notification.Name.NSManagedObjectContextObjectsDidChange,
            object: managedObjectContext,
            queue: OperationQueue.main) { notification in
                if self.isViewLoaded {
                    self.updateLocations()
                }
            }
        }
}
```

This is another example of a property observer put to good use.

As soon as `managedObjectContext` is given a value — which happens in `AppDelegate` during app startup — the `didSet` block tells the `NotificationCenter` to add an observer for the `NSManagedObjectContextObjectsDidChange` notification.

This notification with the very long name is sent out by the `managedObjectContext` whenever the data store changes. In response you would like the following closure to be called. For clarity, here's what happens in the closure:

```
if self.isViewLoaded {
    self.updateLocations()
}
```

This couldn't be simpler: you just call `updateLocations()` to fetch all the `Location` objects again. This throws away all the old pins and it makes new pins for all the newly fetched `Location` objects. Granted, it's not a very efficient method if there are hundreds of annotation objects, but for now it gets the job done.

Note: You use `isViewLoaded` to make sure `updateLocations()` only gets called when the map view is loaded. Because this screen sits in a tab, the view from `MapViewController` does not actually get loaded from the storyboard until the user switches to the Map tab.

So the view may not be loaded yet when the user tags a new location. In that case, it makes no sense to call `updateLocations()` — it could even crash the app since the `MKMapView` object doesn't exist at that point!

- ▶ Run the app. First go to the Map screen to see your existing location pins. Then tag a new location. The map should have added a new pin for it, although you may have to press the Locations bar button to make the new pin appear if it's outside the visible range.

Have another look at that closure. The `notification in` bit is the parameter for the closure. Like functions and methods, closures can take parameters.

Because this particular closure gets called by `NotificationCenter`, you're given a `Notification` object in the `notification` parameter. Since you're not using this `notification` object anywhere in the closure, you could also write it like this:

```
{ _ in  
    . . .  
}
```

You've already seen the `_` underscore used in a few places in the code. This symbol is called the *wildcard* and you can use it whenever a name is expected but you don't really care about it.

Here, the `_` tells Swift you're not interested in the closure's parameter. It also helps to reduce visual clutter in the source code; it's obvious at a glance that this parameter — whatever it may be — isn't being used in the closure.

So whenever you see the `_` used in Swift source code it just means, "there's something here but the programmer has chosen to ignore it."

Exercise: The `Notification` object has a `userInfo` dictionary. From that dictionary it is possible to figure out which objects were inserted/deleted/updated. For example, use the following `print()`s to examine this dictionary:

```
if let dictionary = notification.userInfo {
```

```
    print(dictionary[NSInsertedObjectsKey])
    print(dictionary[NSUpdatedObjectsKey])
    print(dictionary[NSDeletedObjectsKey])
}
```

Note: This will print out an (optional) collection of `Location` objects or `nil` if there were no changes. Your mission, should you choose to accept it: try to make the reloading of the locations more efficient by only inserting or deleting the items that have changed. Good luck! If you get stuck, you can find the solutions from other readers on the raywenderlich.com forums.

That's it for the Map screen.

You can find the project files for this chapter under **34 – Maps** in the Source Code folder.



35

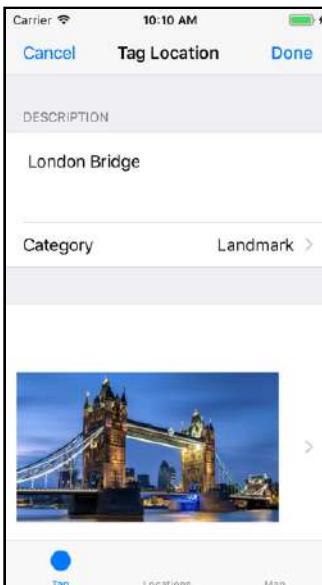
Chapter 35: Image Picker

Eli Ganim

Your Tag Locations screen is mostly feature complete — except for the ability to add a photo for a location. Time to fix that!

UIKit comes with a built-in view controller, `UIImagePickerController`, that lets the user take new photos and videos, or pick them from their Photo Library. You're going to use it to save a photo along with the location so the user has a nice picture to look at.

This is what your screen will look like when you're done:



A photo in the Tag Location screen



In this chapter, you will do the following:

- **Add an image picker:** Add an image picker to your app to allow you to take photos with the camera or to select existing images from your photo library.
- **Show the image:** Show the picked image in a table view cell.
- **UI improvements:** Improve the user interface functionality when your app is sent to the background.
- **Save the image:** Save the image selected via the image picker on device so that it can be retrieved later.
- **Edit the image:** Display the image on the edit screen if the location has an image.
- **Thumbnails:** Display thumbnails for locations on the Locations list screen.

Adding an image picker

Just as you need to ask the user for permission before you can get GPS information from the device, you need to ask for permission to access the user's photo library.

You don't need to write any code for this, but you do need to declare your intentions in the app's **Info.plist**. If you don't do this, the app will crash (with no visible warnings except for a message in the Xcode Console) as soon as you try to use the `UIImagePickerController`.

Info.plist changes

► Open **Info.plist** and add a new row — either use the plus (+) button on existing rows, or right-click and select **Add Row**, or use the **Editor** ➤ **Add Item** menu option.

For the key, use **NSPhotoLibraryUsageDescription**, or choose **Privacy — Photo Library Usage Description** from the dropdown list.

For the value, type: **Add photos to your locations**.

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
Privacy - Photo Library Usage Description	String	Add photos to your locations.
Privacy - Location When In Use Usage Description	String	This app lets you keep track of interesting places. It needs

Adding a usage description in Info.plist

► Also add the key **NSCameraUsageDescription** (or choose **Privacy — Camera Usage Description**) and give it the same description.



Now when the app opens the photo picker or the camera for the first time, iOS will tell the user what the app intends to use the photos for, using the description you just added to Info.plist.

Using the camera to add an image

- In **LocationDetailsViewController.swift**, add the following extension to the end of the source file:

```
extension LocationDetailsViewController:  
    UIImagePickerControllerDelegate,  
    UINavigationControllerDelegate {  
  
    // MARK:- Image Helper Methods  
    func takePhotoWithCamera() {  
        let imagePicker = UIImagePickerController()  
        imagePicker.sourceType = .camera  
        imagePicker.delegate = self  
        imagePicker.allowsEditing = true  
        present(imagePicker, animated: true, completion: nil)  
    }  
}
```

The `UIImagePickerController` is a view controller like any other, but it is built into UIKit and it takes care of the entire process of taking new photos or picking them from the user's photo library. All you need to do is create a `UIImagePickerController` instance, set its properties to configure the picker, set its delegate, and then present it. When the user closes the image picker screen, the delegate methods will let you know the result of the operation.

That's exactly how you've been designing your own view controllers — except that you don't need to add the `UIImagePickerController` to the storyboard.

Note: You're doing this in an extension because it allows you to group all the photo-picking related functionality together.

If you wanted to, you could put these methods in the main class body. That would work fine too, but view controllers tend to become very big with many methods that all do different things.

As a way to preserve your sanity, it's nice to extract conceptually related methods — such as everything that has to do with picking photos — and place them together in their own extension.



You could even move each of these extensions to their own source file, for example “LocationDetailsViewController+PhotoPicking.swift,” but having less files to manage is a good thing!

- Add the following methods to the extension:

```
// MARK:- Image Picker Delegates
func imagePickerController(_ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info:
        [UIImagePickerController.InfoKey : Any]) {
    dismiss(animated: true, completion: nil)
}

func imagePickerControllerDidCancel(_ picker:
    UIImagePickerController) {
    dismiss(animated: true, completion: nil)
}
```

Currently these delegate methods simply remove the image picker from the screen. Soon, you’ll take the image the user picked and add it to the `Location` object, but for now, you just want to make sure the image picker shows up.

Note that the view controller — in this case the extension — must conform to both `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` for this to work, but you don’t have to implement any of the `UINavigationControllerDelegate` methods.

- Now change `tableView(_:didSelectRowAt:)` in the class as follows:

```
override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {
    if indexPath.section == 0 && indexPath.row == 0 {
        .
        .
    } else if indexPath.section == 1 && indexPath.row == 0 {
        takePhotoWithCamera()
    }
}
```

Add Photo is the first row in the second section. When it’s tapped, you call the `takePhotoWithCamera()` method that you just added.

- Run the app, tag a new location or edit an existing one, and tap **Add Photo**.

If you’re running the app in Simulator, bam! It crashes. The error message is this:

```
*** Terminating app due to uncaught exception
```

```
'NSInvalidArgumentException', reason: 'Source type 1 not available'
```

The culprit for the crash is the line:

```
imagePickerController.sourceType = .camera
```

Not all devices have a camera, and Simulator does not. If you try to use the `UIImagePickerController` with a `sourceType` that is not supported by the device or Simulator, the app crashes.

If you run the app on your device — and if it has a camera, which it probably does if it's a recent model — then you should see something like this:



The camera interface

That is very similar to what you see when you take pictures using the iPhone's Camera app. *MyLocations* doesn't let you record video, but you can certainly enable this feature in your own apps, if you wanted to.

Using the photo library to add an image

You can still test the image picker on Simulator, but instead of using the camera, you have to use the photo library.

- Add another method to the extension:

```
func choosePhotoFromLibrary() {
    let imagePicker = UIImagePickerController()
    imagePicker.sourceType = .photoLibrary
    imagePicker.delegate = self
    imagePicker.allowsEditing = true
    present(imagePicker, animated: true, completion: nil)
}
```

This method does essentially the same thing as `takePhotoWithCamera`, except now you set the `sourceType` to `.photoLibrary`.

- Change `didSelectRowAt` to call `choosePhotoFromLibrary()` instead of `takePhotoWithCamera()`.

- Run the app in Simulator and tap **Add Photo**.

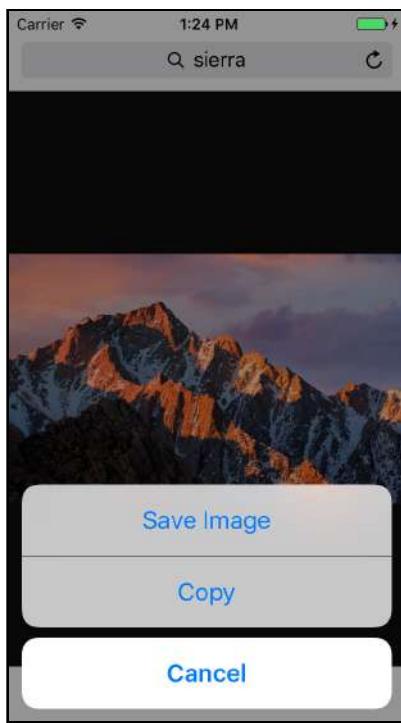
At this point, depending on your iOS version, you may need to give *MyLocations* permission to access the photo library. If you tap **Don't Allow**, the photo picker screen remains empty. If you accidentally do that, you can undo this choice in the Settings app, under **Privacy** ▶ **Photos**. Choose **OK** to allow the app to use the photo library.

However, with iOS 12, you probably won't get the prompt and so you should be fine and you should see a handful of stock images. On older iOS versions, it was possible that you would not see any images at all.

- If you don't see any images for some reason, stop the app and click on the built-in **Photos** app in Simulator. This should display a handful of sample photos. Run the app again and try picking a photo. You may or may not see these sample photos now. If not, you'll have to add your own.

There are several ways you can add new photos to Simulator. You can go into **Safari** (on Simulator), search the internet for an image, press down on the image until a menu appears, and then choose **Save Image**:





Adding images to Simulator

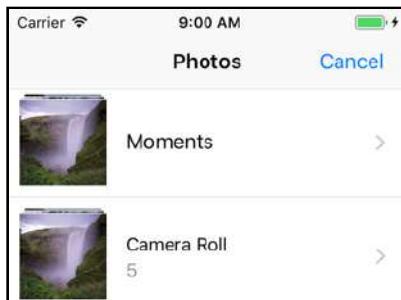
Instead of surfing the internet for images, you can also simply drag and drop an image file on to Simulator window. This adds the picture to your library in the Photos app.

Finally, you can use the Terminal and the `simctl` command. Type the following, all on one line (the last part, `~/Desktop/MyPhoto.JPG`, should be replaced with an actual path to an image you want to add):

```
/Applications/Xcode.app/Contents/Developer/usr/bin/simctl  
addmedia booted ~/Desktop/MyPhoto.JPG
```

The `simctl` tool can be used to manage your Simulator's — type `simctl help` for a list of options. The command `addmedia booted` adds the specified media file to the active Simulator.

- Run the app again. Now you should be able to choose a photo from the Photo Library:



The photos in the library

- Choose one of the photos. The screen now changes to:



The user can tweak the photo

This happens because you set the image picker's `allowsEditing` property to `true`. With this setting enabled, the user can do some quick editing of the photo before making their final choice — in Simulator you can hold down Alt/Option while dragging to rotate and zoom the photo.

So, there are two types of image pickers you can use: the camera and the Photo Library. But the camera won't work everywhere. It's a bit limiting to restrict the app to just picking photos from the library, though.

You'll have to make the app a little smarter and allow the user to choose the camera when it is present.

Choosing between camera and photo library

First, you check whether the camera is available. When it is, you show an **action sheet** to let the user choose between the camera and the Photo Library.

- Add the following methods to **LocationDetailsViewController.swift**, in the photo extension:

```
func pickPhoto() {
    if UIImagePickerController.isSourceTypeAvailable(.camera) {
        showPhotoMenu()
    } else {
        choosePhotoFromLibrary()
    }
}

func showPhotoMenu() {
    let alert = UIAlertController(title: nil, message: nil,
                                 preferredStyle: .actionSheet)

    let actCancel = UIAlertAction(title: "Cancel", style: .cancel,
                                  handler: nil)
    alert.addAction(actCancel)

    let actPhoto = UIAlertAction(title: "Take Photo",
                                style: .default, handler: nil)
    alert.addAction(actPhoto)

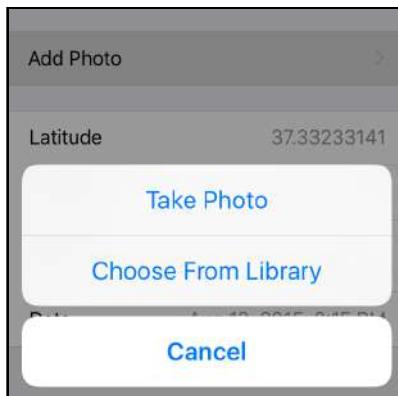
    let actLibrary = UIAlertAction(title: "Choose From Library",
                                  style: .default, handler: nil)
    alert.addAction(actLibrary)

    present(alert, animated: true, completion: nil)
}
```

You use `UIImagePickerController`'s `isSourceTypeAvailable()` method to check whether there's a camera present. If not, you call `choosePhotoFromLibrary()` as that is your only option. But when the device does have a camera, you show a `UIAlertController` on the screen.

Unlike the alert controllers you've used before, this one has the `.actionSheet` style. An action sheet works very much like an alert view, except that it slides in from the bottom of the screen and offers the user one of several choices.

- In `didSelectRowAtIndexPath`, change the call to `choosePhotoFromLibrary()` to `pickPhoto()` instead. This is the last time you'll change this line, honest.
- Run the app on your device to see the action sheet in action:



The action sheet that lets you choose between camera and photo library

Tapping any of the buttons in the action sheet simply dismisses the action sheet but doesn't do anything else yet.

By the way, if you want to test this action sheet in Simulator, then you can fake the availability of the camera by writing the following in `pickPhoto()`:

```
if true ||
UIImagePickerController.isSourceTypeAvailable(.camera) {
```

That will always show the action sheet because the condition is now always true.

The choices in the action sheet are provided by `UIAlertAction` objects. The `handler:` parameter determines what happens when you press the corresponding button in the action sheet.

Right now the handlers for all three choices – Take Photo, Choose From Library, Cancel – are `nil`, so nothing will happen.

- Change these lines to the following:

```
let actPhoto = UIAlertAction(title: "Take Photo",
    style: .default, handler: { _ in
        self.takePhotoWithCamera()
})
```

```
let actLibrary = UIAlertAction(title: "Choose From Library",
    style: .default, handler: { _ in
        self.choosePhotoFromLibrary()
})
```

This gives `handler`: a closure that calls the corresponding method from the extension. You use the `_` wildcard to ignore the parameter that is passed to this closure — a reference to the `UIAlertAction` itself.

- Run the app and make sure the buttons from the action sheet work properly.

There may be a small delay between pressing any of these buttons before the image picker appears, but that's because it's a big component and iOS needs a few seconds to load it up.

Notice that the Add Photo cell remains selected (dark gray background) when you cancel the action sheet. That doesn't look so good.

- In `tableView(_:didSelectRowAt)`, add the following line before the call to `pickPhoto()`:

```
tableView.deselectRow(at: indexPath, animated: true)
```

This first deselects the Add Photo row. Try it out, it looks better this way. The cell background quickly fades from gray back to white as the action sheet slides into the screen.

Showing the image

Now that the user can pick a photo, you should display it somewhere — what's the point otherwise, right? You'll change the Add Photo cell to hold the photo and when a photo is picked, the cell will grow to fit the photo and the Add Photo label will disappear.

- Add two new outlets to the class in **LocationDetailsViewController.swift**:

```
@IBOutlet weak var imageView: UIImageView!
@IBOutlet weak var addPhotoLabel: UILabel!
```

- In the storyboard, drag an Image View into the Add Photo cell. It doesn't really matter how big it is or where you put it. You'll programmatically move it to the proper place later. (This is the reason you made this a custom cell way back when, so you could add this image view to it.)



Adding an Image View to the Add Photo cell

- Connect the Image View to the view controller's `imageView` outlet. Also connect the Add Photo label to the `addPhotoLabel` outlet.
- Select the Image View. In the **Attributes inspector**, check its **Hidden** attribute (in the Drawing section). This makes the image view initially invisible, until you have a photo to give it.

- Add **left, top, right, bottom, and height** Auto Layout constraints to the Image View:

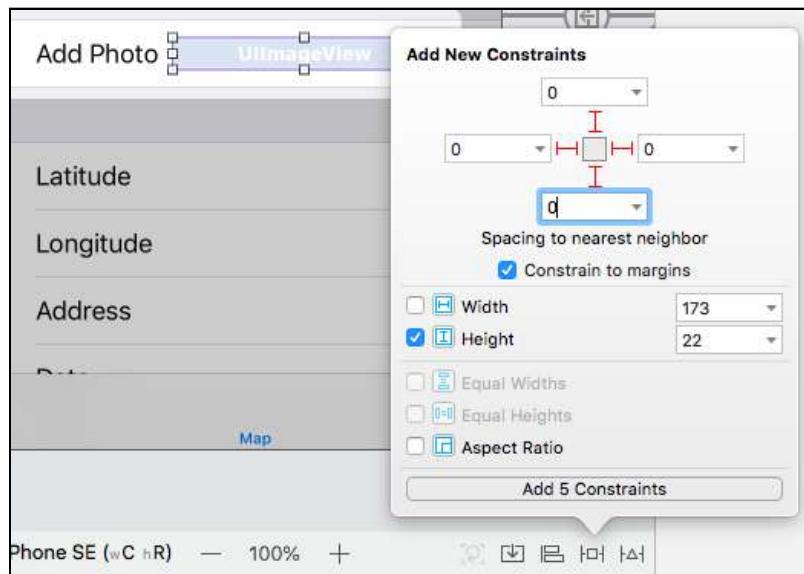


Image View Auto Layout constraints

You will use some of these Auto Layout constraints to move things out of the way, or to expand the image view to fill the cell when an image is displayed. But first, you need a variable to hold the picked image.

- Add a new instance variable to **LocationDetailsViewController.swift**:

```
var image: UIImage?
```

If no photo is picked yet, `image` will be `nil`, so the variable has to be an optional.

- Add a new method to the class:

```
func show(image: UIImage) {
    imageView.image = image
    imageView.isHidden = false
    addPhotoLabel.text = ""
}
```

This puts the image from the parameter into the image view, makes the image view visible, and removes the title from the Add Photo label so that the Auto Layout constraints would move the image over into the space occupied by the label.

- Change the `imagePickerController(_:didFinishPickingMediaWithInfo:)` method from the photo picking extension to the following:

```
func imagePickerController(_ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info:
        [UIImagePickerController.InfoKey : Any]) {

    image = info[UIImagePickerController.InfoKey.editedImage]
        as? UIImage
    if let theImage = image {
        show(image: theImage)
    }

    dismiss(animated: true, completion: nil)
}
```

This is the method that gets called when the user has selected a photo in the image picker.

You can tell by the notation `[UIImagePickerController.InfoKey : Any]` that the `info` parameter is a dictionary. Whenever you see `[A : B]` you're dealing with a dictionary that has keys of type “A” and values of type “B.”

The `info` dictionary contains data describing the image that the user picked. You use the `UIImagePickerController.InfoKey.editedImage` key to retrieve a `UIImage` object that contains the final image after the user moved and/or scaled it — you can also get the original image if you wish, using a different key.

Once you have the photo, you store it in the `image` instance variable so you can use it later.

Dictionaries always return optionals, because there is a theoretical possibility that the key you asked for — `UIImagePickerController.InfoKey.editedImage` in this case — doesn’t actually exist in the dictionary.

Since the `image` instance variable is an optional, you simply assign the value from the dictionary.

If `info[UIImagePickerController.InfoKey.editedImage]` is `nil`, then `image` will be `nil` too. You do need to cast the value from the meaningless `Any` to `UIImage` using the `as?` operator. In this case you need to use the optional cast, `as?` instead of `as!`, because `image` is an optional instance variable.

Once you have the image and it is not `nil`, the call to `show(image:)` puts it in the Add Photo cell.

Exercise: See if you can rewrite the above logic to use a `didSet` property observer on the `image` instance variable. If you succeed, then placing the photo into `image` will automatically update the `UIImageView`, without needing to call `show(image:)`.

- Run the app and choose a photo. Whoops, it looks like you have a small problem here:



The photo is stretched

If you recall, you set the height for the Image View to something like 22 points when you set the Auto Layout constraints earlier because that's how tall the image needed to be to fit the original row. However, when you're displaying the image, a larger value is needed — something like 260 points.

But, of course, if you set the Image View height to 260 at the outset, the image picker cell would start out too tall. So how do you fix this?

Simple enough — you can actually set up connections for Auto Layout constraints too and change the constraint values via code during runtime!

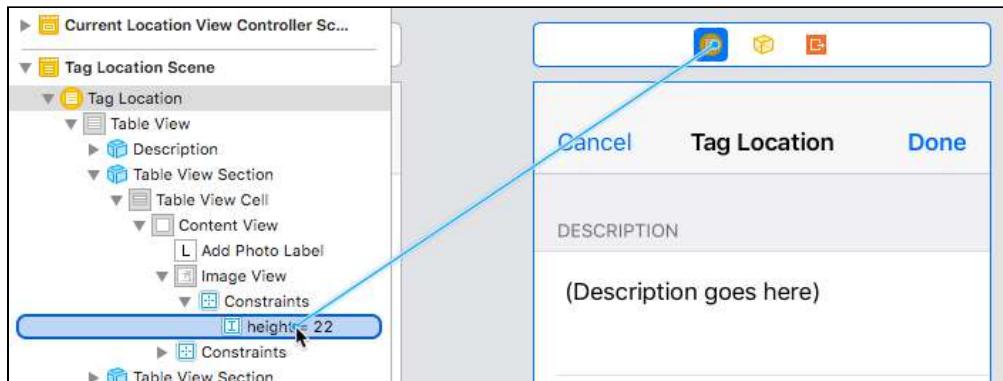
Resizing table view cell to show image

- Add a new outlet for the image height constraint to `LocationDetailsViewController.swift`:

```
@IBOutlet weak var imageHeight: NSLayoutConstraint!
```

- Switch to the storyboard and then connect the new outlet to the height constraint for the image — the easiest way to do this is via the Document Outline since you can pick the exact constraint you want from there.

Simply Control-drag from the circle for the view controller to the correct constraint in the Document Outline and then pick the outlet name, **imageHeight**, from the pop-up menu:



Connect the outlet for the constraint

Now, all you have to do is change the Image View's height constraint to 260 when you display an image!

- Change the `show(image:)` method:

```
func show(image: UIImage) {  
    // Add the following lines  
    imageHeight.constant = 260  
    tableView.reloadData()  
}
```

You simply change the height of the image to 260 points and then refresh the table view to set the photo row to the proper height.

- Try it out. The cell now resizes and is big enough for the whole photo.

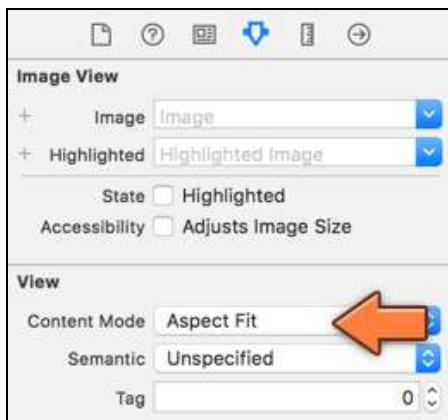


The photo displays correctly

There's one small tweak that you can make. By default, an image view will stretch the image to fit the entire content area. That's probably not what you want for this app.

Setting the image to display correctly

- Go to the storyboard and select the Image View (it may be hard to see on account of it being hidden, but you can still find it in the Document Outline). In the **Attributes inspector**, set its **Content Mode** to **Aspect Fit**.



Changing the image view's content mode

This will keep the image's aspect ratio intact as it is resized to fit within the image view. Play a bit with the other content modes to see what they do. (Aspect Fill is similar to Aspect Fit, except that it tries to fill up the entire view.)



The aspect ratio of the photo is kept intact

That looks a bit better, but there are now larger margins at the top and bottom of the image.

Exercise: Make the height of the photo table view cell dynamic, depending on the aspect ratio of the image. This is a tough one! You can keep the width of the image view at 260 points. This should correspond to the width of the `UIImage` object. You get the aspect ratio by doing `image.size.width / image.size.height`. With this ratio you can calculate what the height of the image view and the cell should be. Good luck! You can find solutions from other readers at forums.raywenderlich.com

UI improvements

The user can take a photo — or pick one — now but the app doesn't save it to the data store yet. Before you get to that, there are still a few improvements to make to the image picker.

Apple recommends that apps remove any alert or action sheet from the screen when the user presses the Home button to move the app to the background.

The user may return to the app hours or days later and they will have forgotten what they were going to do. The presence of the alert or action sheet is confusing and the user might think, "What's that thing doing here?!"

To prevent this from happening, you'll make the Tag Location screen a little more attentive. When the app goes to the background, it will dismiss the action sheet if that is currently showing. You'll do the same for the image picker.

Handling background mode

You saw in the *Checklists* app that the `AppDelegate` is notified by the operating system when the app is about to go to the background through its `applicationDidEnterBackground(_:)` method.

View controllers don't have such a method, but fortunately, iOS sends out "going to the background" notifications through `NotificationCenter` that you can configure the view controller to listen to.

Earlier you used the notification center to observe notifications from Core Data. This time you'll listen for the `UIApplicationDidEnterBackground` notification.



- In **LocationDetailsViewController.swift**, add a new method:

```
func listenForBackgroundNotification() {
    NotificationCenter.default.addObserver(forName:
        UIApplication.didEnterBackgroundNotification,
        object: nil, queue: OperationQueue.main) { _ in

        if self.presentedViewController != nil {
            self.dismiss(animated: false, completion: nil)
        }

        self.descriptionTextView.resignFirstResponder()
    }
}
```

This adds an observer for `UIApplication.didEnterBackgroundNotification`. When this notification is received, `NotificationCenter` will call the closure.

Notice that you’re using the “trailing” closure syntax here; the closure is not a parameter to `addObserver(forName, ...)` but immediately follows the method call.

If there is an active image picker or action sheet, you dismiss it. You also hide the keyboard if the text view is active.

The image picker and action sheet are both presented as modal view controllers that appear above everything else. If such a modal view controller is active, `UIViewController`’s `presentedViewController` property has a reference to that modal view controller.

So, if `presentedViewController` is not `nil` you call `dismiss()` to close the modal screen. (By the way, this has no effect on the category picker; that does not use a modal segue but a push segue.)

- Call the `listenForBackgroundNotification()` method from within `viewDidLoad()`.
- Try it out. Open the image picker (or the action sheet if you’re on a device that has a camera) and exit to the home screen to put the app to sleep.

Then tap the app’s icon to activate the app again. You should now be back on the Tag Location screen — or Edit Location screen if you opted to edit an existing one. The image picker — or action sheet — has automatically closed.

That seems to work, cool!

Removing notification observers

At this point, with iOS versions up to iOS 9.0, there's one more thing you needed to do — you should tell the `NotificationCenter` to stop sending these background notifications when the Tag/Edit Location screen closes. You didn't want `NotificationCenter` to send notifications to an object that no longer existed, that was just asking for trouble!

However, as of iOS 9.0, this is no longer necessary since the system handles all of this for you. But, we'll go ahead and unregister the observer just so that you can see how that works — and also to illustrate another issue that we'll get to soon.

The `deinit` method is a good place to unregister observers.

- First, add a new instance variable:

```
var observer: Any!
```

This will hold a reference to the observer, which is necessary to unregister it later.

The type of this variable is `Any!`, meaning that you don't really care what sort of object this is.

- In `listenForBackgroundNotification()`, change the first line so that it stores the return value of the call to `addObserver()` into this new instance variable:

```
func listenForBackgroundNotification() {
    observer = NotificationCenter.default.addObserver(forName: . . .
    .
```

- Finally, add the `deinit` method:

```
deinit {
    print("/** deinit \(self)\"")
    NotificationCenter.default.removeObserver(observer!)
}
```

You add a `print()` here so you have proof that the view controller really does get destroyed when you close the Tag/Edit Location screen.

- Run the app, edit an existing location, and tap Done to close the screen.

No matter how hard you try, you won't find the `/** deinit` message anywhere in the Xcode Console.

Guess what? The `LocationDetailsViewController` doesn't get destroyed for some reason. That means the app is leaking memory... Of course, this was all a big setup to bring up the topic of closures and capturing.

Remember how in closures you always have to specify `self` when you want to access an instance variable or call a method? That is because closures capture any variables that are used inside the closure.

When it captures a variable, the closure simply stores a reference to that variable. This allows it to use the variable at some later point when the closure is actually performed.

Why is this important? If the code inside the closure uses a local variable, the method that created this variable may no longer be active by the time the closure is performed. After all, when a method ends all locals are destroyed. But when such a local is captured by a closure, it stays alive until the closure is also done with it.

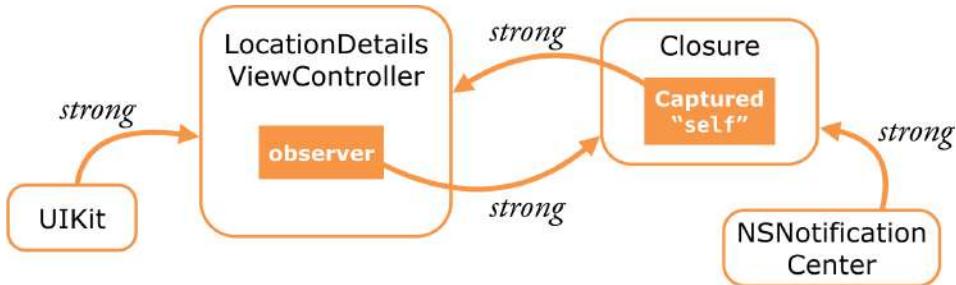
Because the closure needs to keep the objects from those captured variables alive in the time between capturing and actually performing the closure, it stores a *strong* reference to those objects. In other words, capturing means the closure becomes a shared owner of the captured objects.

What may not be immediately obvious is that `self` is also one of those variables and therefore gets captured by the closure. Sneaky! That's why Swift requires you to explicitly write out `self` inside closures, so you won't forget this value is being captured.

In the context of `LocationDetailsViewController`, `self` refers to the view controller itself. So, as the closure captures `self`, it creates a strong reference to the `LocationDetailsViewController` object, and the closure becomes a co-owner of this view controller.

Remember, as long as an object has owners, it is kept alive. So this closure is keeping the view controller alive, even after you closed it!

This is known as an *ownership cycle*, because the view controller itself has a strong reference back to the closure through the `observer` variable.



The relationship between the view controller and the closure

In case you're wondering, the view controller's other owner is UIKit. The observer is also being kept alive by NotificationCenter.

This sounds like a classic catch-22 problem! Fortunately, there is a way to break the ownership cycle. You can give the closure a *capture list*. What's *that* you ask? All will be explained soon!

► Change `listenForBackgroundNotification()` to the following:

```

func listenForBackgroundNotification() {
    observer = NotificationCenter.default.addObserver(
        forName: UIApplication.didEnterBackgroundNotification,
        object: nil, queue: OperationQueue.main) { [weak self] _ in

        if let weakSelf = self {
            if weakSelf.presentedViewController != nil {
                weakSelf.dismiss(animated: false, completion: nil)
            }
            weakSelf.descriptionTextView.resignFirstResponder()
        }
    }
}
  
```

There are a couple of new things here. Let's look at the first part of the closure:

```

{ [weak self] _ in
    .
    .
}
  
```

The `[weak self]` bit is the capture list for the closure. It tells the closure that the variable `self` will still be captured, but as a weak reference. As a result, the closure no longer keeps the view controller alive.

Weak references are allowed to become `nil`, which means the captured `self` is now an optional inside the closure. You need to unwrap it with `if let` before you can send messages to the view controller.

Other than that, the closure still does the exact same things as before.

- Try it out. Open the Tag/Edit Location screen and close it again. You should now see the `print()` from `deinit` in the Xcode Console.

That means the view controller gets destroyed properly and the notification observer is removed from `NotificationCenter`. Good riddance!

Do note that as of iOS 9.0 and above, even if you do not remove the observer explicitly, the system would handle this for you and automatically remove the observer when the view controller is deallocated. So you don't have to worry about any side effects from an errant observer any longer.

But it's always a good idea to clean up after yourself. Use `print()`'s to make sure your objects really get deallocated! Xcode also comes with Instruments, a handy tool that you can use to detect such issues.

Saving the image

The ability to pick photos is rather useless if the app doesn't also save them. So, that's what you'll do here.

It is possible to store images in the Core Data store as **BLOBs (Binary Large OBjects)**, but that is not recommended. Large blocks of data are better off stored as regular files in the app's Documents directory.

Note: Core Data has an “Allows external storage” feature that is designed to make this process completely transparent for the developer. In theory, you can put data of any size into your entities and Core Data automatically decides whether to put the data into the SQLite database or store it as an external file.

Unfortunately, this feature doesn't work very well in practice. Until this part of Core Data becomes rock solid, you'll be doing it by hand.

When the image picker gives you a `UIImage` object for a photo, that image only lives in the iPhone's working memory.



The image may also be stored as a file somewhere if the user picked it from the photo library, but that's not the case if they just snapped a new picture. Besides, the user may have resized or cropped the image.

So you have to save that `UIImage` to a file of your own if you want to keep it. The photos in `MyLocations` will be saved in JPEG format.

You need a way to associate that JPEG file with your `Location` object. The obvious solution is to store the filename in the `Location` object. You won't store the entire filename, just an ID, which is a positive number. The image file itself will be named **Photo-XXX.jpg**, where XXX is the numeric ID.

Data model changes

- ▶ Open the Data Model editor. Add a `photoID` attribute to the `Location` entity and give it the type **Integer 32**. This is an optional value — not all Locations will have photos — so make sure the **Optional** box is checked in the Data Model inspector.
- ▶ Add a property for this new attribute to `Location+CoreDataProperties.swift`:

```
@NSManaged public var photoID: NSNumber?
```

Remember that for an object that is managed by Core Data, you have to declare the property as `@NSManaged`.

You may be wondering why you're declaring the type of `photoID` as `NSNumber` and not as `Int` or, more precisely, `Int32`. Remember that Core Data is an Objective-C framework, so you're limited by the possibilities of that language. `NSNumber` is how number objects are handled in Objective-C.

For various reasons, you can't represent an `Int` value as an optional in Objective-C. Instead, you'll use the `NSNumber` class. Swift will automatically convert between `Int` values and this `NSNumber`, so it's no big deal.

You'll now add some other properties to the `Location` object to make working with photos a little easier.

- ▶ Add the `hasPhoto` computed property to `Location+CoreDataClass.swift`:

```
var hasPhoto: Bool {  
    return photoID != nil  
}
```

This determines whether the `Location` object has a photo associated with it or not. Swift's optionals make this easy.

- Also add the `photoURL` property:

```
var photoURL: URL {  
    assert(photoID != nil, "No photo ID set")  
    let filename = "Photo-\(photoID!.intValue).jpg"  
    return applicationDocumentsDirectory.appendingPathComponent(  
        filename)  
}
```

This property computes the full URL for the JPEG file for the photo. Note that iOS uses URLs to refer to files, even those saved on the local device.

You'll save these JPEG files in the app's Documents directory. To get the URL to that directory, you use the global variable `applicationDocumentsDirectory` that you added to `Functions.swift` earlier.

Notice the use of `assert()` to make sure the `photoID` is not `nil`. An *assertion* is a special debugging tool that is used to check that your code always does something valid. If not, the app will crash with a helpful error message. You'll see more of this later when we talk about finding bugs — and squashing them.

Assertions are a form of defensive programming. Most of the crashes you've seen so far were actually caused by assertions in UIKit. They allow the app to crash in a controlled manner. Without these assertions, programming mistakes could crash the app at random moments, making it very hard to find out what went wrong.

If the app were to ask a `Location` object for its `photoURL` without having given it a valid `photoID` earlier, the app will crash with the message “No photo ID set.” If so, there is a bug in the code somewhere because this is not supposed to happen. Internal consistency checks like this can be very useful.

Assertions are usually enabled only while you're developing and testing your app and disabled when you upload the final build of your app to the App Store. By then, there should be no more bugs in your app — or so you would hope! It's a good idea to use `assert()` in strategic places to catch yourself making programming errors.

- Add a `photoImage` property:

```
var photoImage: UIImage? {  
    return UIImage(contentsOfFile: photoURL.path)  
}
```

This returns a `UIImage` object by loading the image file. You'll need this later to show the photos for existing `Location` objects.

Note that this property has the optional type `UIImage?` — that's because loading the image may fail if the file is damaged or removed. Of course, that *shouldn't* happen, but no doubt you've heard of Murphy's Law... As I've repeatedly said, it's good to get into the habit of defensive programming.

There is one more thing to add, a `nextPhotoID()` method. This is a class method, meaning that you don't need to have a `Location` instance to call it. You can call this method anytime from anywhere.

► Add the method:

```
class func nextPhotoID() -> Int {  
    let userDefaults = UserDefaults.standard  
    let currentID = userDefaults.integer(forKey: "PhotoID") + 1  
    userDefaults.set(currentID, forKey: "PhotoID")  
    userDefaults.synchronize()  
    return currentID  
}
```

You need to have some way to generate a unique ID for each `Location` object. All `NSManagedObjects` have an `objectID` method, but that returns something unreadable such as:

```
<x-coredata://C26CC559-959C-49F6-BEF0-F221D6F3F04A/Location/p1>
```

You can't really use that in a filename. So instead, you're going to put a simple integer in `UserDefault`s and update it every time the app asks for a new ID — this is similar to what you did in the last app to make `ChecklistItem` IDs for use with local notifications.

It may seem a little silly to use `UserDefault`s for this when you're already using Core Data as the data store, but with `UserDefault`s, the `nextPhotoID()` method is only five lines. You've seen how verbose the code is for fetching something from Core Data and then saving it again. This is just as easy. Of course, if you wanted to, as an exercise, you could try to implement these IDs using Core Data...

That does it for `Location`. Now you have to save the image and fill in the `Location` object's `photoID` field. This happens in the `Location Details View Controller`'s `done()` action.

Saving the image to a file

- In **LocationDetailsViewController.swift**, in the `done()` method, add the following in between where you set the properties of the `Location` object and where you save the managed object context:

```
// Save image
if let image = image {
    // 1
    if !location.hasPhoto {
        location.photoID = Location.nextPhotoID() as NSNumber
    }
    // 2
    if let data = image.jpegData(compressionQuality: 0.5) {
        // 3
        do {
            try data.write(to: location.photoURL, options: .atomic)
        } catch {
            print("Error writing file: \(error)")
        }
    }
}
```

This code is only performed if `image` is not `nil` — in other words, when the user has picked a photo.

1. You need to get a new ID and assign it to the `Location`'s `photoID` property, but only if you're adding a photo to a `Location` that didn't already have one. If a photo existed, you simply keep the same ID and overwrite the existing JPEG file.
2. The `image.jpegData(compressionQuality: 0.5)` call converts the `UIImage` to JPEG format and returns a `Data` object. `Data` is an object that represents a blob of binary data, usually the contents of a file.
3. You save the `Data` object to the path given by the `photoURL` property. Also notice the use of a `do-try-catch` block again.

- Run the app, tag a location, choose a photo, and press Done to exit the screen. Now the photo you picked should be saved in the app's Documents directory as a regular JPEG file.



The photo is saved in the app's Documents folder

Note: The first time you run the app after adding a new attribute to the data model (photoID), the `NSPersistentContainer` performs a migration of the data store behind the scenes to make sure the data store is in sync again with the data model. If this doesn't work for you for some reason, then remove the old `DataModel.sqlite` file from the Library/Application Support folder and try again — or, simply reset Simulator or remove the app from your test device.

- Tag another location and add a photo to it. Hmm... if you look into the app's Documents directory, this seems to have overwritten the previous photo.

Exercise: Try to debug this one on your own. What is going wrong here? This is a tough one!

Answer: When you create a new `Location` object, its `photoID` property gets a default value of 0. That means each `Location` initially has a `photoID` of 0. That should really be `nil`, which means "no photo."

- In **LocationDetailsViewController.swift**, add the following line near the top of `done()`:

```
@IBAction func done() {
    if let temp = locationToEdit {
    } else {
        location.photoID = nil           // add this
    }
}
```

You now set the `photoID` of a new `Location` object to `nil` so that the `hasPhoto` property correctly recognizes that these `Locations` as not having a photo yet.

- Run the app again and tag multiple locations with photos. Verify that now each photo is saved individually.

Verifying photoID in SQLite

If you have Liya or another SQLite inspection tool, you can verify that each `Location` object has been given a unique `photoID` value (in the `ZPHOTOID` column):

Field	Type	Length	Null	Key	Default	Class
Z_PK	integer	0	NO	PRI	0	NSNumber
Z_ENT	integer	0	YES	0	0	NSNumber
Z_OPT	integer	0	YES	0	0	NSNumber
ZPHOTOID	integer	0	YES	0	0	NSNumber
ZDATE	timestamp	0	YES	0	0	NSDate
ZLATITUDE	float	0	YES	0	0	NSDecimalNumber
ZLONGITUDE	float	0	YES	0	0	NSDecimalNumber
ZCATEGORY	varchar	0	YES	0	0	NSString
ZLOCATIONDESCRIPTION	varchar	0	YES	0	0	NSString
ZPLACEMARK	blob	0	YES	0	0	NSData

Z_PK	Z_ENT	Z_OPT	ZPHOTOID	ZDATE	ZLATITUDE	ZLONGITUDE	ZCATEGORY	ZLOCATIONDES...	ZPLA...
3	1	2	0	2014-09-27 1...	51.58838759	4.77649758	Landmark	City Center	
4	1	1	1	2014-09-27 1...	51.58983	4.77317	Landmark	The Harbor	
5	1	1	2	2014-09-27 1...	51.59138	4.77916	Park	Valkenbergpark	

The Location objects with unique photoId values in Liya

Editing the image

So far, all the changes you've made were for the Tag Location screen and adding new locations. Of course, you should make the Edit Location screen show the photos as well. The change to `LocationDetailsViewController` is quite simple.

- Change `viewDidLoad()` in `LocationDetailsViewController.swift` to:

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

    if let location = locationToEdit {
        title = "Edit Location"
        // New code block
        if location.hasPhoto {
            if let theImage = location.photoImage {
                show(image: theImage)
            }
        }
        // End of new code
    }
    ...
}
```

If the `Location` that you're editing has a photo, this calls `show(image:)` to display it in the photo cell.

Recall that the `photoImage` property returns an optional, `UIImage?`, so you use `if let` to unwrap it. This is another bit of defensive programming.

Sure, if `hasPhoto` is `true` there should always be a valid image file present. But it's possible to imagine a scenario where there isn't — the JPEG file could have been erased or corrupted — even though that "should" never happen. I'm sure you've had your own share of computer gremlins eating important files.

Note also what you **don't** do here: the `Location`'s image is *not* assigned to the `image` instance variable. If the user doesn't change the photo, then you don't need to write it out to a file again — it's already in that file and doing perfectly fine, thank you.

If you were to put the photo in the `image` variable, then `done()` would overwrite the existing file with the exact same data, which is a little silly. Therefore, the `image` instance variable will only be set when the user picks a new photo.

- Run the app and take a peek at the existing locations from the Locations or Map tabs. The Edit Location screen should now show the photos for the locations you're editing.

- Verify that you can also change the photo and that the JPEG file in the app's Documents directory gets overwritten when you press the Done button.

There's another editing operation the user can perform on a location: deletion. What happens to the image file when a location is deleted? At the moment nothing. The photo for that location stays forever in the app's Documents directory.

Cleaning up on location deletion

Let's add some code to remove the photo file, if it exists, when a Location object is deleted.

- First add a new method to **Location+CoreDataClass.swift**:

```
func removePhotoFile() {
    if hasPhoto {
        do {
            try FileManager.default.removeItem(at: photoURL)
        } catch {
            print("Error removing file: \(error)")
        }
    }
}
```

This code snippet can be used to remove any file or folder. The `FileManager` class has all kinds of useful methods for dealing with the file system.

- Deleting locations happens in **LocationsViewController.swift**. Add the following line to `tableView(_:commit:forRowAt:)`:

```
override func tableView(_ tableView: UITableView,
                      commit editingStyle: UITableViewCell.EditingStyle,
                      forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        let location = fetchedResultsController.object(at:
                                                       indexPath)

        location.removePhotoFile() // add this line
        managedObjectContext.delete(location)
        .
    }
}
```

The new line calls `removePhotoFile()` on the `Location` object just before it is deleted from the Core Data context.

- Try it out. Add a new location and give it a photo. You should see the JPEG file in the Documents directory.

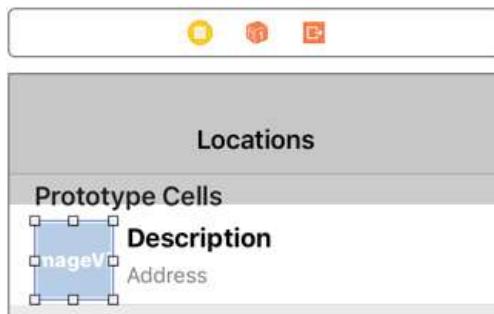
From the Locations screen, delete the location you just added and look in the Documents directory to make sure the JPEG file truly is a goner.

Thumbnails

Now that locations can have photos, it's a good idea to show thumbnails for these photos in the Locations tab. That will liven up this screen a little... a plain table view with just a bunch of text isn't particularly exciting.

Storyboard changes

- Go to the storyboard editor. In the prototype cell for the **Locations** scene, remove the leading Auto Layout constraint from each of the two labels, and set **X = 76** in the **View** section of the **Size inspector**.
- Drag a new **Image View** into the cell. Place it at the top-left corner of the cell. Give it the following position: **X = 16, Y = 2**. Make it 52 by 52 points big.



The table view cell has an image view

- Add **top**, **left**, **height**, and **width** Auto Layout constraints for the currently set values for the new Image View.
- Select each of the labels and set their left constraint again so that each one is positioned relative to the image view — the spacing should be 8 points if you set all the positions and sizes above correctly.
- Connect the image view to a new `UIImageView` outlet on `LocationCell`, named `photoImageView`.

Exercise: Make this connection with the Assistant editor. Tip: you should connect the image view to the cell, not to the view controller.

Now you can put any image into the table view cell simply by passing it to the `LocationCell`'s `photoImageView` property.

Code changes

- Go to `LocationCell.swift` and add the following method:

```
func thumbnail(for location: Location) -> UIImage {
    if location.hasPhoto, let image = location.photoImage {
        return image
    }
    return UIImage()
}
```

This returns either the image from the `Location` or an empty placeholder image.

You should read this `if` statement as, “if the location has a photo, and I can unwrap `location.photoImage`, then return the unwrapped image.”

You have previously seen the `&&` (logical and) used to combine two conditions, but you cannot write the above like this:

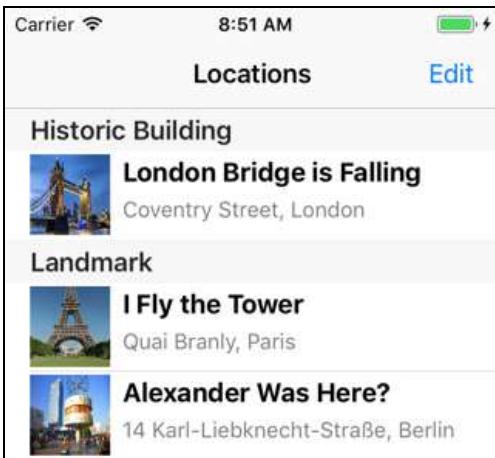
```
if location.hasPhoto && let image = location.photoImage
```

The `&&` only works if both conditions are booleans, but here you’re unwrapping an optional as well. In that case you must combine the two conditions with a comma.

- Call this new method from the end of `configure(for:)`:

```
photoImageView.image = thumbnail(for: location)
```

- Try it out. The Locations tab should now look something like this:



Images in the Locations table view

You've got thumbnails, all right!

But look closely and you'll see that the images are a little squashed again. That's because you didn't set the Aspect Fit content mode on the image view — but there's a bigger problem here. Literally.

These photos are potentially huge — 2592 by 1936 pixels or more — even though the image view is only 52 pixels square. To make them fit, the image view needs to scale down the images by a lot — which is also why they look a little “gritty.”

What if you have tens or even hundreds of locations? That is going to require a ton of memory and processing speed just to display these tiny thumbnails. A better solution is to scale down the images before you put them into the table view cell.

And what better way to do that than using an extension?

Extensions

So far you've used extensions on your view controllers to group related functionality together, such as delegate methods. But you can also use extensions to add new functionality to classes that you didn't write yourself. That includes classes such as `UIImage` from the iOS frameworks.

If you ever catch yourself thinking, “Gee, I wish object X had such-and-such a method,” then you can probably add that method by using an extension.

Suppose you want `String` to have a method for adding random words to a string. You could add the `addRandomWord()` method to `String` as follows.

First you create a new source file, for example `String+RandomWord.swift`. It would look like this:

```
import Foundation

extension String {
    func addRandomWord() -> String {
        let words = ["rabbit", "banana", "boat"]
        let value = Int.random(in: 0 ..< words.count)
        let word = words[value]
        return self + word
    }
}
```

You can now call `addRandomWord()` on any `String` value in your code:

```
let someString = "Hello, "
let result = someString.addRandomWord()
print("The queen says: \(result)")
```

Extensions are pretty cool because they make it simple to add new functionality to an existing class. In other programming languages you would have to make a subclass and put your new methods in there, but extensions are often a cleaner solution.

Besides new methods, you can also add new computed properties, but you can't add regular instance variables. You can also use extensions on types that don't even allow inheritance, such as `structs` and `enums`.

Thumbnails via `UIImage` extension

You are going to add an extension to `UIImage` that lets you resize the image. You'll use it as follows:

```
return image.resized(withBounds: CGSize(width: 52, height: 52))
```

The `resized(withBounds:)` method is new. The “bounds” is the size of the rectangle (or square in this case) that encloses the image. If the image itself is not square, then the resized image may actually be smaller than the bounds.

Let's write the extension.

- Add a new file to the project and choose the **Swift File** template. Name the file **UIImage+Resize.swift**.

- Replace the contents of this new file with:

```
import UIKit

extension UIImage {
    func resized(withBounds bounds: CGSize) -> UIImage {
        let horizontalRatio = bounds.width / size.width
        let verticalRatio = bounds.height / size.height
        let ratio = min(horizontalRatio, verticalRatio)
        let newSize = CGSize(width: size.width * ratio,
                             height: size.height * ratio)

        UIGraphicsBeginImageContextWithOptions(newSize, true, 0)
        draw(in: CGRect(origin: CGPoint.zero, size: newSize))
        let newImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        return newImage!
    }
}
```

This method first calculates how big the image should be in order to fit inside the bounds rectangle. It uses the “aspect fit” approach to keep the aspect ratio intact.

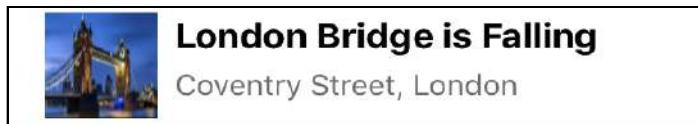
Then it creates a new image context and draws the image into that. We haven’t really dealt with graphics contexts before, but they are an important concept in Core Graphics – in case you’re wondering, it has nothing to do with the managed object context from Core Data, even though they both use the term “context.”

Let’s put this extension to work.

- Switch to **LocationCell.swift**. Update the `thumbnail(for:)` method:

```
func thumbnail(for location: Location) -> UIImage {
    if location.hasPhoto, let image = location.photoImage {
        return image.resized(withBounds: CGSize(width: 52,
                                                height: 52))
    }
    return UIImage()
}
```

- Run the app. The thumbnails should look like this:



The photos are shrunk to the size of the thumbnails

The images are a little blurry and they still seem to be stretched out. This is because the content mode on the image view is still wrong.

Previously it shrunk the big photos to 52 by 52 points, but now the thumbnails may actually be smaller than 52 points (unless the photo was perfectly square) and they get scaled up to fill the entire image view rectangle.

- Go to the storyboard and set the **Content Mode** of the image view to **Center**.
► Run the app again and now the photos look A-OK:



The thumbnails now have the correct aspect ratio

Exercise: Change the resizing function in the `UIImage` extension to resize using the “Aspect Fill” rules instead of the “Aspect Fit” rules. Both keep the aspect ratio intact but Aspect Fit keeps the entire image visible while Aspect Fill fills up the entire rectangle and may cut off parts on the sides. In other words, Aspect Fit scales to the longest side but Aspect Fill scales to the shortest side.



Aspect Fit
Keeps the entire image
but adds empty border



Aspect Fill
Fills up the whole frame
but cuts off sides

Aspect Fit vs. Aspect Fill

Handling low-memory situations

The `UIImagePickerController` is very memory-hungry. Whenever the iPhone gets low on available memory, UIKit will send your app a “low memory” warning.

When that happens, you should reclaim as much memory as possible, or iOS might be forced to terminate the app. And that’s something to avoid — users generally don’t like apps that suddenly quit on them!

Chances are that your app gets one or more low-memory warnings while the image picker is open, especially when you run it on a device that has other apps suspended in the background. Photos take up a lot of space — especially when your camera is 5 or more megapixels — so it’s no wonder that memory fills up quickly.

You can respond to memory warnings by overriding the `didReceiveMemoryWarning()` method in your view controllers to free up any memory you no longer need. This is often done for things that can easily be recalculated or recreated later, such as thumbnails or other cached objects.

UIKit is already pretty smart about low memory situations and it will do everything it can to release memory, including the thumbnail images of rows that are not (or no longer) visible in your table view.

For *MyLocations* there’s not much that you need to do to free up additional memory, you can rely on UIKit to automatically take care of it. But in your own apps you might want to take extra measures, depending on the sort of cached data that you have.

By the way, on Simulator you can trigger a low memory warning using the **Debug ▾ Simulate Memory Warning** menu item. It’s smart to test your apps under low memory conditions, because they are likely to encounter such situations out in the wild once they’re running on user devices.

Great! That concludes all the functionality for this app. Now it’s time to fine-tune its looks.

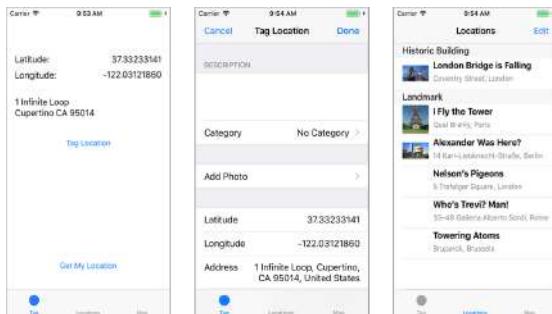
You can find the project files for this chapter under **35 – Image Picker** in the Source Code folder.

Chapter 36: Polishing the App

Eli Ganim

Apps with appealing visuals sell better than ugly ones. Now that the app works as it should, it's time to make it look good!

You're going to go from this:



To this:



The main screen gets the biggest makeover, but you'll also tweak the others a little.



raywenderlich.com

980

You'll do the following in this chapter:

- **Convert placemarks to strings:** Refactor the code to display placemarks as text values so that the code is centralized and easier to use.
- **Back to black:** Change the appearance of the app to have a black background and light text.
- **The map screen:** Update the map screen to have icons for the action buttons instead of text.
- **Fix the table views:** Update all the table views in the app to have black backgrounds with white text.
- **Polish the main screen:** Update the appearance of the main screen to add a bit of awesome sauce!
- **Make some noise:** Add sound effects to the app.
- **The icon and launch images:** Add the app icon and launch images to complete the app.

Converting placemarks to strings

Let's begin by improving the code. I'm not really happy with the way the reverse geocoded street address gets converted from a `CLPlacemark` object into a string. It works, but the code is unwieldy and repetitive.

There are three places where this happens:

- `CurrentLocationViewController`, the main screen.
- `LocationDetailsViewController`, the Tag/Edit Location screen.
- `LocationsViewController`, the list of saved locations.

Let's start with the main screen. `CurrentLocationViewController.swift` has a method named `string(from:)` where this conversion happens. It's supposed to return a string that looks like this:

```
subThoroughfare thoroughfare  
locality administrativeArea postalCode
```

This string goes into a `UILabel` that has room for two lines, so you use the `\n` character sequence to create a line-break between the thoroughfare and locality.



The problem is that any of these properties may be `nil`. So, the code has to be smart enough to skip the empty ones, that's what all the `if let` lets are for.

There's a lot of repetition going on in this method. You can refactor this.

Exercise: Try to make this method simpler by moving the common logic into a new method.

Answer: Here's a possible solution. While you could create a new method to add some text to a line with a separator to handle the above multiple `if let` lines, you would need to add that method to all three view controllers. Of course, you could add the method to the `Functions.swift` file to centralize the method too...

But better still, what if you created a new `String` extension since this functionality is for adding some text to an existing string? Sounds like a plan?

► Add a new file to the project using the **Swift File** template. Name it `String+AddText`.

► Add the following to `String+AddText.swift`:

```
extension String {
    mutating func add(text: String?,
                      separatedBy separator: String) {
        if let text = text {
            if !isEmpty {
                self += separator
            }
            self += text
        }
    }
}
```

Most of the code should be pretty self-explanatory. You ask the string to add some text to itself, and if the string is currently not empty, you add the specified separator first before adding the new text.

Mutating

Notice the `mutating` keyword. You haven't seen this before. Sorry, it doesn't have anything to do with X-Men — programming is certainly fun, but not *that* fun!

When a method changes the value of a `struct`, it must be marked as `mutating`. Recall that `String` is a `struct`, which is a value type, and therefore cannot be modified when declared with `let`.



The `mutating` keyword tells Swift that the `add(text:separatedBy:)` method can only be used on strings that are made with `var`, but not on strings made with `let`. If you try to modify `self` in a method on a struct that is not marked as `mutating`, Swift considers this an error. You don't need to use the `mutating` keyword on methods inside a `class` because classes are reference types and can always be mutated, even if they are declared with `let`.

- Switch over to **CurrentLocationViewController.swift** and replace `string(from:)` with the following:

```
func string(from placemark: CLPlacemark) -> String {  
    var line1 = ""  
    line1.add(text: placemark.subThoroughfare, separatedBy: "")  
    line1.add(text: placemark.thoroughfare, separatedBy: " ")  
  
    var line2 = ""  
    line2.add(text: placemark.locality, separatedBy: "")  
    line2.add(text: placemark.administrativeArea,  
              separatedBy: " ")  
    line2.add(text: placemark.postalCode, separatedBy: " ")  
  
    line1.add(text: line2, separatedBy: "\n")  
    return line1  
}
```

That looks a lot cleaner. The logic that decides whether or not to add a `CLPlacemark` property to the string now lives in your new `String` extension, so you no longer need all those `if let` statements. You also use `add(text:separatedBy:)` to add `line2` to `line1` with a newline character in between.

- Run the app to see if it works.

There's still a small thing you can do to improve the new `add(text:separatedBy:)` method. Remember default parameter values? You can use them here.

- In **String+AddText.swift**, change the line that defines the method to:

```
mutating func add(text: String?,  
                  separatedBy separator: String = "") {
```

Now, instead of:

```
line1.add(text: placemark.subThoroughfare, separatedBy: "")
```

You can write:

```
line1.add(text: placemark.subThoroughfare)
```



The default value for `separator` is an empty string. If the `separatedBy` parameter is left out, `separator` will be set to `""`.

► Make these changes in `CurrentLocationViewController.swift`:

```
func string(from placemark: CLPlacemark) -> String {
    . . .
    line1.add(text: placemark.subThoroughfare)
    line2.add(text: placemark.locality)
    . . .
```

Where the separator is an empty string, you leave out the `separatedBy: ""` part of the method call. Note that the other instances of `add(text:separatedBy:)` in the method don't have empty strings as the separator but instead, have a space.

Now you have a pretty clean solution that you can re-use in the other two view controllers.

► In `LocationDetailsViewController.swift`, replace the `string(from:)` code with:

```
func string(from placemark: CLPlacemark) -> String {
    var line = ""
    line.add(text: placemark.subThoroughfare)
    line.add(text: placemark.thoroughfare, separatedBy: " ")
    line.add(text: placemark.locality, separatedBy: ", ")
    line.add(text: placemark.administrativeArea,
             separatedBy: ", ")
    line.add(text: placemark.postalCode, separatedBy: " ")
    line.add(text: placemark.country, separatedBy: ", ")
    return line
}
```

It's slightly different from how the main screen does it. There are no newline characters and some of the elements are separated by commas instead of just spaces. Newlines aren't necessary here because the label will wrap.

The final place where placemarks are shown is `LocationsViewController`. However, this class doesn't have a `string(from:)` method. Instead, the logic for formatting the address lives in `LocationCell`.

► Go to `LocationCell.swift`. Change the relevant part of `configure(for:)`:

```
func configure(for location: Location) {
    if let placemark = location.placemark {
        var text = ""
        text.add(text: placemark.subThoroughfare)
```

```
text.add(text: placemark.thoroughfare, separatedBy: " ")  
text.add(text: placemark.locality, separatedBy: ", ")  
addressLabel.text = text  
} else {  
    . . .
```

You only show the street and the city, so the conversion is simpler.

And that's it for placemarks.

Back to black

Right now the app looks like a typical iOS app: lots of white, gray tab bar, blue tint color. Time to go for a radically different look and paint the whole thing black.

In iOS 13, Apple added support for Dark Mode, which lets you switch the entire UI from light to dark. You'll learn how easy it is to support Dark Mode in the next app you build, however you still need to know how to customize the app's look for the cases where you want the color scheme to be something other than light or dark.

- Open the storyboard and go to the **Current Location View Controller**. Select the top-level view and change its **Background Color** to **Black Color**.
- Select all the labels (probably easiest from the Document Outline since they are now invisible) and set their **Color** to **White Color**.
- Change the **Font** of the **(Latitude/Longitude goes here)** labels to **System Bold 17**.
- Select the two buttons and change their **Font** to **System Bold 20**, to make them slightly larger. You may need to resize their frames to make the text fit (remember, ⌘= is the magic keyboard shortcut).
- In the **File inspector**, change **Global Tint** to the color **Red: 255, Green: 238, Blue: 136**. That makes the buttons and other interactive elements yellow, which stands out nicely against the black background.
- Select the Get My Location button and change its **Text Color** to **White Color**. This provides some contrast between the two buttons.

The storyboard should look like this:



The new yellow-on-black design

When you run the app, there are two obvious problems:

1. The status bar text has become invisible — it is black text on a black background.
2. The grey tab bar sticks out like a sore thumb. Also, the yellow tint color doesn't get applied to the tab bar icons.

To fix this, you can use the `UIAppearance` API — this is a set of methods that lets you customize the look of the standard UIKit controls.

Using `UIAppearance`

When customizing the UI, you can customize your app on a per-control basis, as you've done up to this point, or you can use the “appearance proxy” to change the look of all of the controls of a particular type at once. That's what you're going to do here.

- Add the following method to `AppDelegate.swift`:

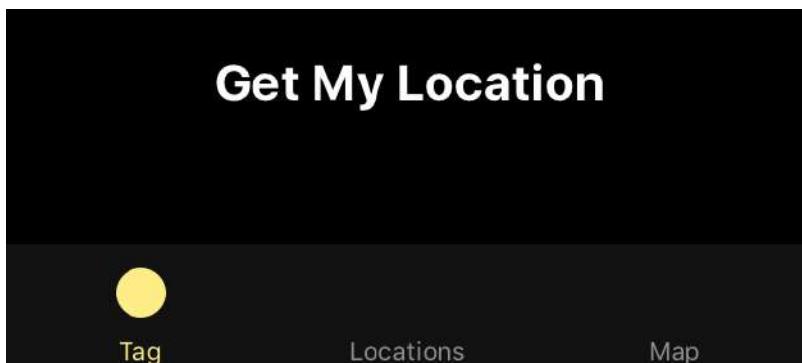
```
func customizeAppearance() {
    UINavigationBar.appearance().barTintColor = UIColor.black
    UINavigationBar.appearance().titleTextAttributes = [
        NSAttributedString.Key.foregroundColor:
            UIColor.white ]
    UITabBar.appearance().barTintColor = UIColor.black
    let tintColor = UIColor(red: 255/255.0, green: 238/255.0,
                           blue: 136/255.0, alpha: 1.0)
    UITabBar.appearance().tintColor = tintColor
}
```

This changes the “bar tint” or background color of all navigation bars and tab bars in the app to black in one fell swoop. It also sets the color of the navigation bar’s title label to white and applies the tint color to the tab bar.

- Call this method from the top of `application(_:didFinishLaunchingWithOptions:)`:

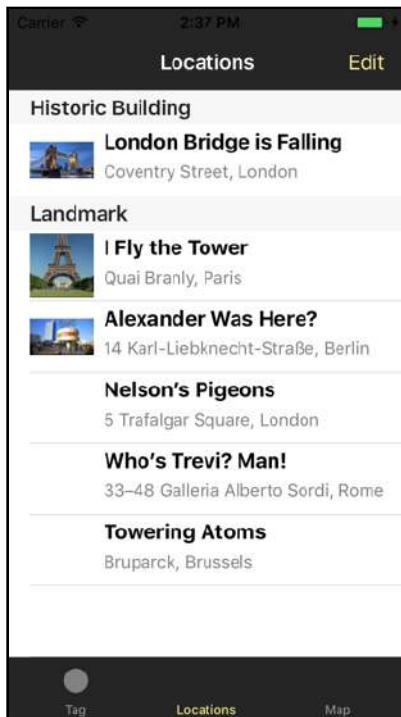
```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions . . .) -> Bool {
    customizeAppearance()
    . . .
}
```

This looks better already.



The tab bar is now nearly black and has yellow icons

On the Locations and Map screens you can clearly see that the bars now have a dark tint:



The navigation and tab bars appear in a dark color

Keep in mind that the bar tint is not the true background color. The bars are still translucent, which is why they appear as a medium gray rather than pure black.

Tab bar icons

The icons in the tab bar could also do with some improvement. The Xcode Tabbed Application template put a bunch of cruft in the app that you're no longer using — let's get rid of it all.

- Remove the **SecondViewController.swift** file from the project.
- Remove the **first** and **second** images from the asset catalog (**Assets.xcassets**).

Tab bar images should be basic grayscale images of up to 30×30 points — that is 60×60 pixels for Retina and 90×90 pixels for Retina HD. You don't have to tint the images; iOS will automatically draw them in the proper color.

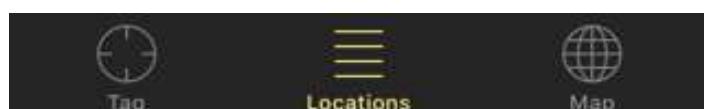
- The resources for this **tutorial** include an **Images** directory. Add the files from this folder to the asset catalog.
- Go to the storyboard. Select the **Tab Bar Item** of the navigation controller embedding the Current Location screen. In the **Attributes inspector**, under **Image** choose **Tag** — this is the name of one of the images you've just added.



Choosing an image for a Tab Bar Item

- For the Tab Bar Item of the navigation controller attached to the Locations screen, choose the **Locations** image.
- For the Tab Bar Item of the navigation controller embedding the Map View Controller, choose the **Map** image.

Now the tab bar looks a lot more appealing:



The tab bar with proper icons

The status bar

The status bar is currently invisible on the Tag screen and appears as black text on dark gray on the other two screens. It would look better if the status bar text was white instead.

To do this, you need to override the `preferredStatusBarStyle` property in your view controllers and make it return the value `.lightContent`.

The simplest way to make the status bar white for all your view controllers in the entire app is to replace the `UITabBarController` with your own subclass.

- Add a new source file to the project and name it **MyTabBarController.swift**.

- Replace the contents of **MyTabBarController.swift** with:

```
import UIKit

class MyTabBarController: UITabBarController {
    override var preferredStatusBarStyle: UIStatusBarStyle {
        return .lightContent
    }

    override var childForStatusBarStyle: UIViewController? {
        return nil
    }
}
```

By returning `nil` from `childForStatusBarStyle`, the tab bar controller will look at its own `preferredStatusBarStyle` property instead of those from the other view controllers.

- In the storyboard, select the Tab Bar Controller and in the **Identity inspector** change its **Class** to **MyTabBarController**. This tells the storyboard that it should now create an instance of your subclass when the app starts up.

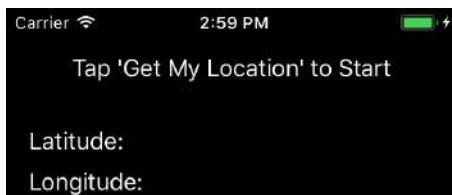
That's right, you can replace standard UIKit components with your own subclasses!

Subclassing lets you change what the built-in UIKit objects do — that's the power of object-oriented programming. But don't get carried away and alter their behavior *too* much — before you know it, your app ends up with an identity crisis!

`MyTabBarController` still does everything that the standard `UITabBarController` does. You only override `preferredStatusBarStyle` to change the status bar color.

You can plug this `MyTabBarController` class into any app that uses a tab bar controller, and from then on, all its view controllers will have a white status bar.

Now, the status bar is white everywhere:



The status bar is visible again

Well, almost everywhere... When you open the photo picker, the status bar fades to black again. Subclasses to the rescue again!

- Add a new file to the project and name it **MyImagePickerController.swift**.
(Getting a sense of déjà vu?)

- Replace the contents of **MyImagePickerController.swift** with:

```
import UIKit

class MyImagePickerController: UIImagePickerController {
    override var preferredStatusBarStyle: UIStatusBarStyle {
        return .lightContent
    }
}
```

Now, instead of instantiating the standard `UIImagePickerController` to pick a photo, you should use this new subclass.

- Go to **LocationDetailsViewController.swift**. In `takePhotoWithCamera()` and `choosePhotoFromLibrary()`, change the line that creates the image picker to:

```
let imagePicker = MyImagePickerController()
```

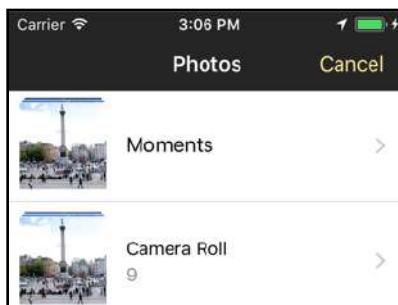
This is allowed because `MyImagePickerController` is a subclass of the standard `UIImagePickerController` — it has the same properties and methods. As far as `UIKit` is concerned, the two are interchangeable. So, you can use your subclass anywhere you'd use `UIImagePickerController`.

While you're at it, the photo picker still uses the standard blue tint color. That makes its navigation bar buttons hard to read. The fix is simple: set the tint color on the Image Picker Controller just before you present it.

- Add the following line to the two methods:

```
imagePicker.view.tintColor = view.tintColor
```

Now, the Cancel button appears in yellow instead of blue.

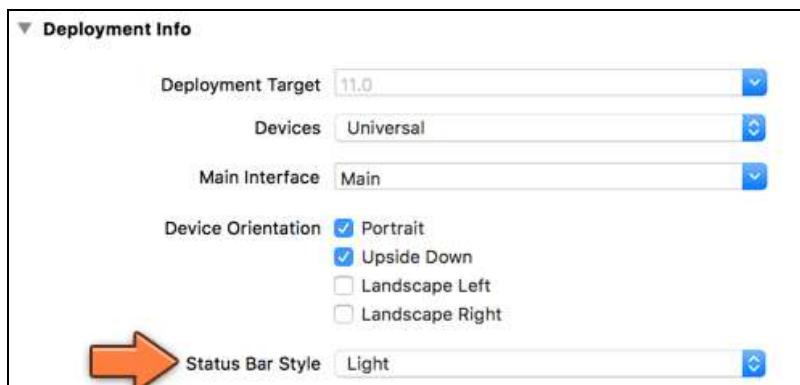


The photo picker with the new colors

There is one more thing to change. When the app starts up, iOS looks in the Info.plist file to determine whether it should show a status bar while the app launches, and if so, what color that status bar should be.

Right now, it's set to Default, which is the black status bar.

- Just to be thorough, go to the **Project Settings** screen. In the **General** tab, under **Deployment Info** is a **Status Bar Style** option. Change this to **Light**.

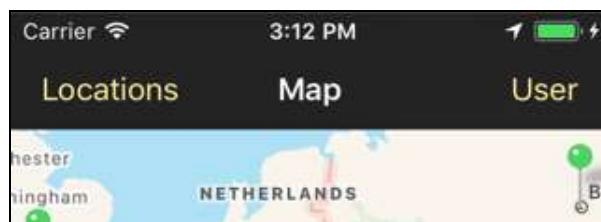


Changing the status bar style for app startup

And now the status bar really is white everywhere!

The map screen

The Map screen currently has a somewhat busy navigation bar with three pieces of text in it: the title and the two buttons.



The bar button items have text labels

The design advice that Apple gives is to prefer text to icons because icons tend to be harder to understand. The disadvantage of using text is that it makes your navigation bar more crowded.

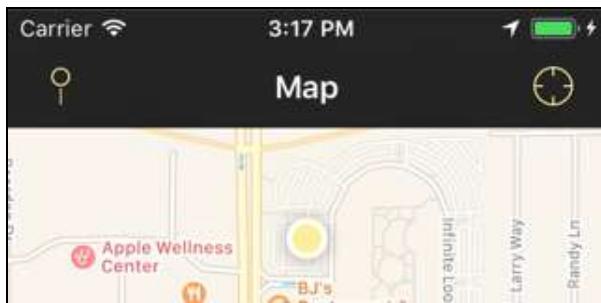
There are two possible solutions:

1. Remove the title. If the purpose of the screen is obvious, which it is in this case, then the title “Map” is superfluous. You might as well remove it.
2. Keep the title but replace the button labels with icons.

For this app, you’ll choose the second option.

- Go to the Map scene in the storyboard and select the **Locations** bar button item. In the **Attributes inspector**, under **Image** choose **Pin**. This will remove the text from the button.
- For the User bar button item, choose the **User** image.

The Map screen now looks like this:



Map screen with the button icons

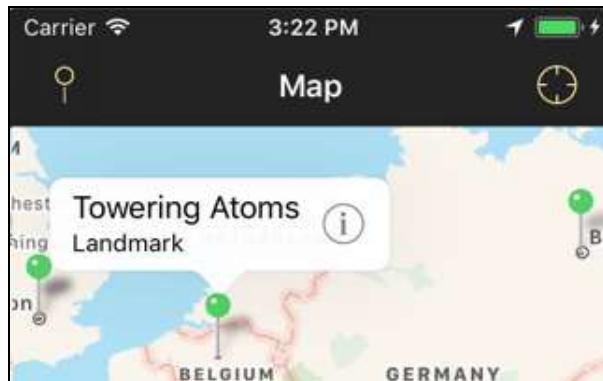
Notice that the dot for the user’s current location is drawn in the yellow tint color (it was a blue dot before).

The **i** button on the map annotations also appears in yellow, making it hard to see on the white callout. Fortunately, you can override the tint color on a per-view basis. There’s no rule that says the tint color has to be the same everywhere!

- In **MapViewController.swift**, in the method `mapView(_:viewFor:)`, add this below the line that sets `pinView.pinTintColor`:

```
pinView.tintColor = UIColor(white: 0.0, alpha: 0.5)
```

This sets the annotation's tint color to half-opaque black:



The callout button is now easier to see

Fixing the table views

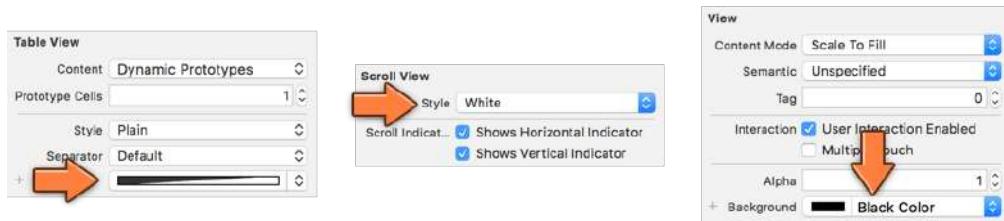
The app is starting to shape up, but there are still some details to take care of. The table views, for example, are still very white.

Unfortunately, what `UIAppearance` can do for table views is very limited. So, you'll have to customize each of the table views individually.

This can be done either via code, or via storyboard. The advantage of using storyboards is that you can see the actual changes such as color, spacing, font etc. and be sure how a change affects the rest of the UI. So, let's make these changes via storyboards as much as possible.

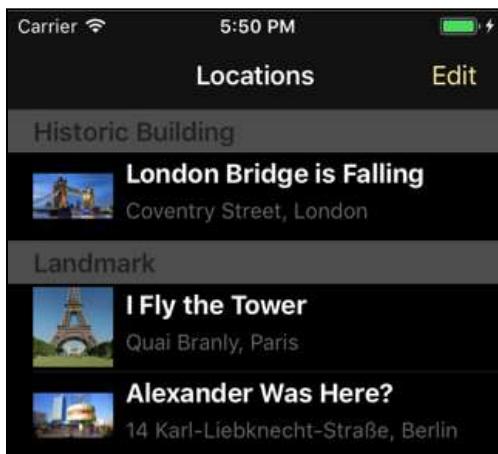
Storyboard changes for the Locations scene

- Open the storyboard and select the table view for the Locations scene. Set **Table View - Separator color to white with 20% Opacity**, **Scroll View - Indicators to white**, and **View - Background to black**.

*Table view color changes*

This makes the table view itself black but does not alter the cells.

- Select the prototype cell in the table view and set its **View - Background** to **black**.
- Next, select the Description label in the cell and set its **Label - Color** and **Label - Highlighted** color to **white**.
- Select the Address label and set its **Label - Color** and **Label - Highlighted** color to **white with 60% Opacity**.
- Run the app. That's starting to look pretty good already:

*The table view cells are now white-on-black*

That's as far as you can get with customization via storyboard. But there are a couple of small issues still.

Code changes for the Locations view

The first, when you tap a cell it still lights up in a bright color, which is a little jarring. It would look better if the selection color was more subdued.

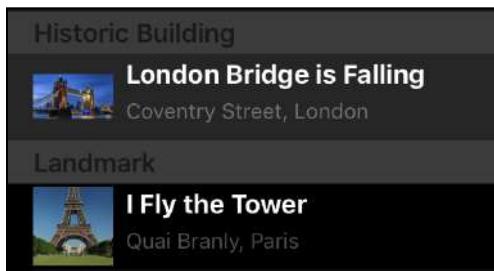
Unfortunately, there is no “`selectionColor`” property on `UITableViewCell`, but you can give it a different view to display when it is selected via a `UITableViewCell`’s `selectedBackgroundView` property.

- In `LocationCell.swift`, replace `awakeFromNib()` with the following:

```
override func awakeFromNib() {
    super.awakeFromNib()
    let selection = UIView(frame: CGRect.zero)
    selection.backgroundColor = UIColor(white: 1.0, alpha: 0.3)
    selectedBackgroundView = selection
}
```

Every object that comes from a storyboard has the `awakeFromNib()` method. This method is invoked when UIKit loads the object from the storyboard. It’s the ideal place to customize its looks.

Here, you create a new `UIView` filled with a dark gray color. This new view is placed on top of the cell’s background when the user taps on the cell. It will look like this:



The selected cell has a subtly different background color

The second issue is that the section headers are a bit on the heavy side. There is no easy way to customize the existing headers, but you can replace them with a view of your own.

- Go to `LocationsViewController.swift` and add the following table view delegate method:

```
override func tableView(_ tableView: UITableView,
    viewForHeaderInSection section: Int) -> UIView? {

    let labelRect = CGRect(x: 15,
        y: tableView.sectionHeaderHeight - 14,
        width: 300, height: 14)
    let label = UILabel(frame: labelRect)
    label.font = UIFont.boldSystemFont(ofSize: 11)

    label.text = tableView.dataSource!.tableView!(
```

```
        tableView, titleForHeaderInSection: section)

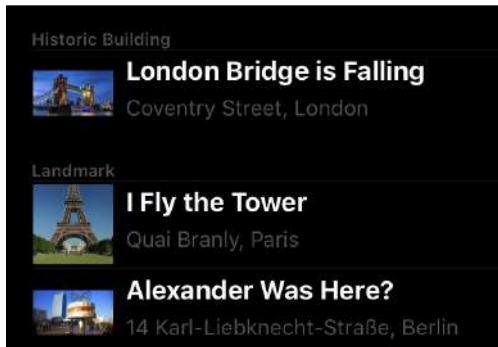
label.textColor = UIColor(white: 1.0, alpha: 0.6)
label.backgroundColor = UIColor.clear

let separatorRect = CGRect(
    x: 15, y: tableView.sectionHeaderHeight - 0.5,
    width: tableView.bounds.size.width - 15, height: 0.5)
let separator = UIView(frame: separatorRect)
separator.backgroundColor = tableView.separatorColor

let viewRect = CGRect(x: 0, y: 0,
                      width: tableView.bounds.size.width,
                      height: tableView.sectionHeaderHeight)
let view = UIView(frame: viewRect)
view.backgroundColor = UIColor(white: 0, alpha: 0.85)
view.addSubview(label)
view.addSubview(separator)
return view
}
```

This method gets called once for each section in the table view. Here, you create a label for the section name, a 1-pixel high view that functions as a separator line, and a container view to hold these two subviews.

It looks like this:



The section headers now draw much less attention to themselves

Note: Did you notice anything special about the following line?

```
label.text = tableView.dataSource!.tableView!(tableView,
titleForHeaderInSection: section)
```

This asks the table view's data source for the text to put in the header. The `dataSource` property is an optional so you're using `!` to unwrap it. But that's not the only `!` in this line...

You're calling the `tableView(_:titleForHeaderInSection:)` method on the table view's data source, which is of course the `LocationsViewController` itself.

But this method is an optional method — not all data sources need to implement it. Because of that you have to *unwrap the method* with the exclamation mark in order to use it. Unwrapping methods... does it get any crazier than that?

By the way, you can also write this as:

```
label.text = self.tableView(tableView, titleForHeaderInSection: section)
```

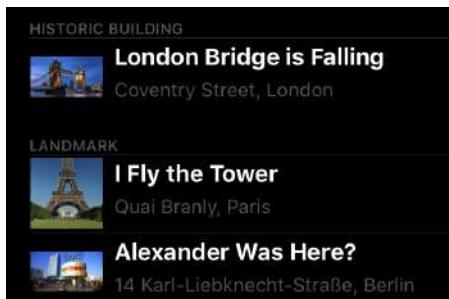
Here you use `self` to directly access that method on `LocationsViewController`. Both ways achieve exactly the same thing, since the view controller happens to be the table view's data source.

Another small improvement you can make is to always put the section headers in uppercase.

► Change `tableView(_:titleForHeaderInSection:)` to:

```
override func tableView(_ tableView: UITableView,
    titleForHeaderInSection section: Int) -> String? {
    let sectionInfo = fetchedResultsController.sections![section]
    return sectionInfo.name.uppercased()
}
```

Now the section headers look even better:



The section header text is in uppercase

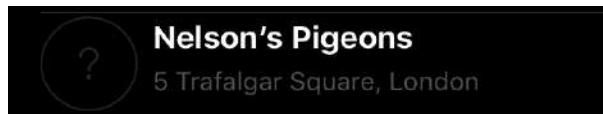
Currently, if a location does not have a photo, there is a black gap where the thumbnail is supposed to be. It's better to show a placeholder image. You already added one to the asset catalog when you imported the Images folder.

- In **LocationCell.swift**'s `thumbnail(for:)`, replace the last line that returns an empty `UIImage` with:

```
return UIImage(named: "No Photo")!
```

Recall that `UIImage(named:)` is a failable initializer, so it returns an optional. Don't forget the exclamation point at the end to unwrap the optional.

Now locations without photos appear like so:



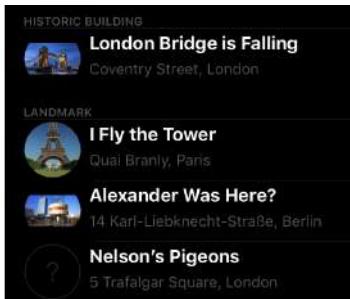
A location using the placeholder image

That makes it a lot clearer to the user that the photo is missing. (As opposed to, say, being a photo of a black hole.) The placeholder image is round. That's the fashion for thumbnail images on iOS these days, and it's pretty easy to make the other thumbnails rounded too.

- Still in **LocationCell.swift**, add the following lines to the end of `awakeFromNib()`:

```
// Rounded corners for images
photoImageView.layer.cornerRadius =
    photoImageView.bounds.size.width / 2
photoImageView.clipsToBounds = true
separatorInset = UIEdgeInsets(top: 0, left: 82, bottom: 0,
                             right: 0)
```

This gives the image view rounded corners with a radius that is equal to half the width of the image, which makes it a perfect circle. The `clipsToBounds` setting makes sure that the image view respects these rounded corners and does not draw outside them. The `separatorInset` moves the separator lines between the cells a bit to the right so there are no lines between the thumbnail images.



The thumbnails are now circular

Note: As you'll notice from the above image, the rounded thumbnails don't look very good if the original photo isn't square. You may want to change the Mode of the image view back to Aspect Fill or Scale to Fill so that the thumbnail always fills up the entire image view.

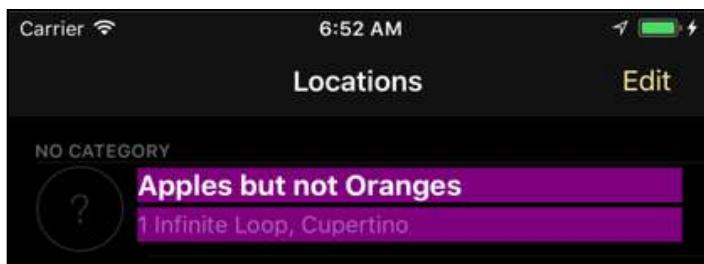
At this point, you probably want to make sure that the labels in this screen extend to cover the full width of larger screens — remember that while you've been designing for 320 point wide screens, you've been setting up Auto Layout constraints so that all screen sizes will be automatically supported.

Tip: To verify that the labels now take advantage of all the available screen space on larger screens, give them a non-transparent background color. Do you like bright purple?

► Add these lines to `awakeFromNib()` (in `LocationCell.swift`, of course) and run the app:

```
descriptionLabel.backgroundColor = UIColor.purple  
addressLabel.backgroundColor = UIColor.purple
```

This is how it looks on an iPhone 8 Plus screen:



The labels resize to fit the iPhone 8 Plus

When you're done testing, don't forget to remove the lines that set the background color. It's useful as a debugging tool, but not particularly pretty to look at.

There are two other table views in the app and they require similar changes.

Table view changes for Tag Location screen

- Open the storyboard and select the table view for the Tag Location scene. Set **Table View - Separator** color to **white with 20% Opacity**, **ScrollView - Indicators** to **white**, and **View - Background** to **black**.
- Select all the static cells in the table view and set their **View - Background** to **black**.
- Select the Description text view and set its **Text View - Color** to **white**, and **View - Background** to **black**.
- Select the Add Photo label and set its **Label - Color** and **Label - Highlighted** color to **white**.
- Select the main label from all the cells with the Right Detail style and set their **Label - Color** and **Label - Highlighted** color to **white**.
- Select the detail label from all the cells with the Right Detail style and set their **Label - Color** and **Label - Highlighted** color to **white with 60% Opacity**.
- Select the Address label and set its **Label - Color** and **Label - Highlighted** color to **white**.
- Select the Address detail label and set its **Label - Color** and **Label - Highlighted** color to **white with 60% Opacity**.

That completes all the storyboard changes but there are a few code changes left.

Previously, you modified the cell's subclasss to add the selection highlighting. However, you have static table view cells here and don't have a subclasss to modify. Don't despair yet though, the table view delegate has a handy method that comes in useful here.

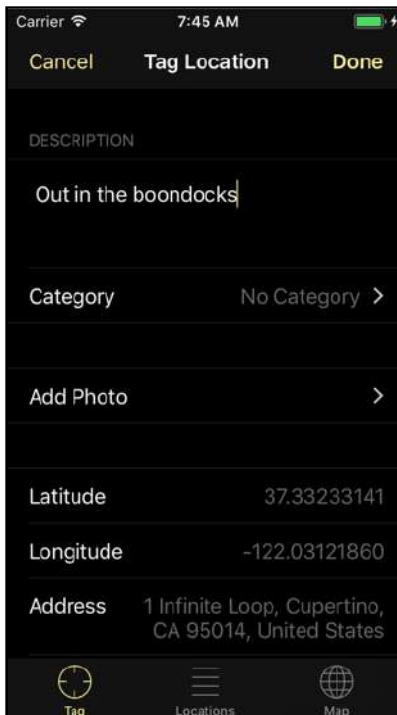
- Open **LocationDetailsViewController.swift** and add the following method:

```
override func tableView(_ tableView: UITableView,  
                      willDisplay cell: UITableViewCell,  
                      forRowAt indexPath: IndexPath) {  
    let selection = UIView(frame: CGRect.zero)  
    selection.backgroundColor = UIColor(white: 1.0, alpha: 0.3)
```

```
    cell.selectedBackgroundView = selection  
}
```

The `willDisplay` delegate method is called just before a cell becomes visible. So, you can do some last-minute customizations on the cell and its contents in this method.

- Run the app. The Tag Location screen should now look like this:



The Tag Location screen with styling applied

Table view changes for the Category Picker screen

The final table view is the category picker. There's nothing new here, the changes are basically the same as before.

- Open the storyboard and select the table view for the Category Picker view controller. Set **Table View - Separator** color to **white with 20% Opacity**, **ScrollView - Indicators** to **white**, and **View - Background** to **black**.

- Select the prototype cell in the table view and set its **View - Background** to **black**.
- Select the label in the prototype cell and set its **Label - Color** and **Label - Highlighted** color to **white**.

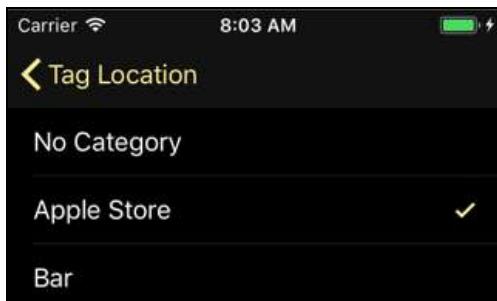
All that's left is to set the cell background for highlighted cells. Since there is no subclass for the cell, it's possible to use the table view delegate's `willDisplay` method again.

However, remember that you are dealing with a prototype cell here. That means that it already is being set up in code via `cellForRowAt`. So why not simply use the existing method to do the extra work? Remember, there's often multiple ways to do the same thing.

- Open **CategoryPickerController.swift** and add the following code to `cellForRowAt`, just before the `return`:

```
override func tableView(_ tableView: UITableView,  
                      cellForRowAt indexPath: IndexPath) ->  
    UITableViewCell {  
  
    let selection = UIView(frame: CGRect.zero)  
    selection.backgroundColor = UIColor(white: 1.0, alpha: 0.3)  
    cell.selectedBackgroundView = selection  
    // End new code  
    return cell  
}
```

Now the category picker is dressed in black as well. It's a bit of work to change the visuals of all these table views by hand, but it's worth it.



The category picker is lookin' sharp

Polishing the main screen

I'm pretty happy with all the other screens, but the main screen needs a bit more work to be presentable.

Here's what you'll do:

- Show a logo when the app starts up. Normally, such splash screens are bad for the user experience, but here you can get away with it.
- Make the logo disappear with an animation when the user taps Get My Location.
- While the app is fetching the coordinates, show an animated activity spinner to make it even clearer to the user that something is going on.
- Hide the **Latitude:** and **Longitude:** labels until the app has found coordinates.

You will first hide the text labels from the screen until the app actually has some coordinates to display. The only label that will be visible until then is the one at the top and it will say “Searching...” or give some kind of error message.

In order to do this, you must have outlets for the labels.

► Add the following properties to **CurrentLocationViewController.swift**:

```
@IBOutlet weak var latitudeTextLabel: UILabel!
@IBOutlet weak var longitudeTextLabel: UILabel!
```

You'll put the logic for updating these labels in a single place, `updateLabels()`, so that hiding and showing them is pretty straightforward.

► Change `updateLabels()` in **CurrentLocationViewController.swift**:

```
func updateLabels() {
    if let location = location {
        latitudeTextLabel.isHidden = false
        longitudeTextLabel.isHidden = false
    } else {
        latitudeTextLabel.isHidden = true
        longitudeTextLabel.isHidden = true
    }
}
```

► Connect the **Latitude:** and **Longitude:** labels in the storyboard to the `latitudeTextLabel` and `longitudeTextLabel` outlets.

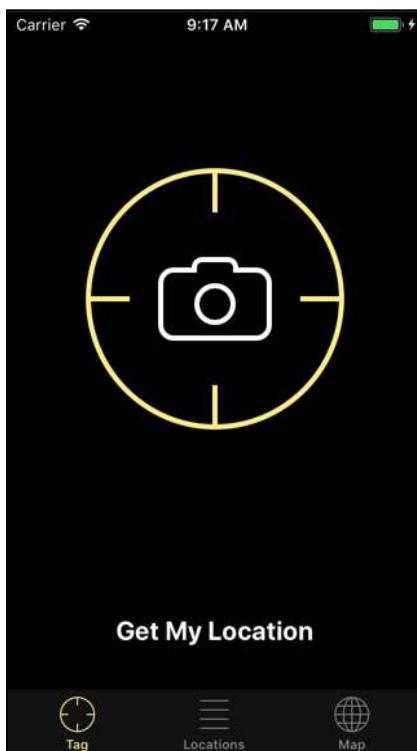
- Run the app and verify that the **Latitude:** and **Longitude:** labels only appear when you have obtained GPS coordinates.

The first impression

The main screen looks decent and is completely functional, but it could do with more pizzazz. It lacks the “Wow!” factor. You want to impress users the first time they start your app and keep them coming back. To pull this off, you’ll add a logo and a cool animation.

When the user hasn’t yet pressed the Get My Location button, there are no GPS coordinates and the Tag Location button is hidden.

Instead of showing a completely blank upper panel, you can show a large version of the app’s icon.



The welcome screen of MyLocations

When the user taps the Get My Location button, the icon rolls out of the screen — it’s round so that kinda makes sense — while a panel with the GPS status will slide in.

This is pretty easy to program thanks to the power of Core Animation and it makes the app a whole lot more impressive for first-time users.

First, you need to move the labels into a new container subview.

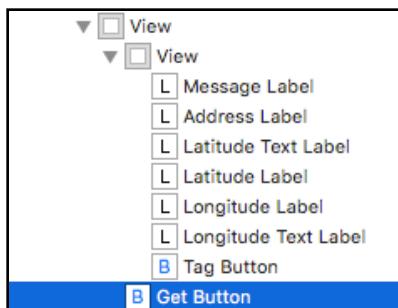
► Open the storyboard and go to the **Current Location View Controller**. In the Document Outline, select the six labels and the Tag Location button. With these seven views selected, choose **Editor** ➤ **Embed In** ➤ **View Without Inset** from the Xcode menu bar.

This creates a blank, white `UIView` and puts these labels and the button inside that new view.

Note: The "View Without Inset" option is new in Xcode 10. Previously, you only had the "View" option which created a view with some padding around the controls that you enclosed in the view. This new option does not add any extra padding and keeps your enclosed controls at their original locations.

- Change the **Background** color of this new container view to **Clear Color**, so that everything becomes visible again. The layout of the screen hasn't changed; you have simply reorganized the view hierarchy so that you can easily manipulate and animate this group of views as a whole. Grouping views in a container view is a common technique for building complex layouts.
- To avoid problems on smaller screens, make sure that the Get My Location button sits higher up in the view hierarchy than the container view. If the button sits under another view you cannot tap it anymore.

Non-intuitively, in the Document Outline, the button must sit below the container view. If it doesn't, drag to rearrange:



Get My Location must sit below the container view in the Document Outline

Note: When you drag the Get My Location button, make sure you're not dropping it into the container view. The view you just added and the Get My Location button should sit at the same level in the view hierarchy.

When you embedded the six labels and the button in the container view, the Auto Layout constraints that those seven controls had to the main view were broken. Makes sense, right? Because those controls are now inside a different view.

You have to fix a few Auto Layout constraints so that the controls are laid out correctly within the container view.

- Select the Container View and set its Auto Layout constraints as follows: **left=16**, **top=0**, and **right=16**.
- Select the (**Message Label**) at the top and set its Auto Layout Constraints to: **left=0**, **top=0**, and **right=0**.
- Select the (**Latitude**), (**Longitude**), and (**Address goes here**) labels and set their Auto Layout Constraints to: **left=0**.
- Select the (**Latitude goes here**), (**Longitude goes here**), and (**Address goes here**) labels and set their Auto Layout Constraints to: **right=0**.
- Finally, set the Tag Location button's Auto Layout Constraints to: **left=0**, **bottom=0**, and **right=0**.
- Add the following outlet to **CurrentLocationViewController.swift**:

```
@IBOutlet weak var containerView: UIView!
```

- In the storyboard, connect the new container **UIView** to the **containerView** outlet.

Now on to the good stuff!

- Add the following instance variables to **CurrentLocationViewController.swift**:

```
var logoVisible = false

lazy var logoButton: UIButton = {
    let button = UIButton(type: .custom)
    button.setBackgroundImage(UIImage(named: "Logo"),
                           for: .normal)
    button.sizeToFit()
    button.addTarget(self, action: #selector(getLocation),
```

```
        for: .touchUpInside)
button.center.x = self.view.bounds.midX
button.center.y = 220
return button
}()
```

The logo image is actually a button, so that you can tap the logo to get started. The app will show this button when it starts up, and when it doesn't have anything better to display — for example, after you press Stop and there are no coordinates and no error. To orchestrate this, you'll use the boolean `logoVisible`.

The button is a “custom” type `UIButton`, meaning that it has no title text or other frills. It draws the `Logo.png` image and calls the `getLocation()` method when tapped. This is another one of those lazily loaded properties; It's nicer because it keeps all the initialization logic inline with the declaration of the property.

► Add the following method:

```
func showLogoView() {
    if !logoVisible {
        logoVisible = true
        containerView.isHidden = true
        view.addSubview(logoButton)
    }
}
```

This hides the container view so the labels disappear, and puts the `logoButton` object on the screen. This is the first time `logoButton` is accessed, so at this point the lazy loading kicks in.

► In `updateLabels()`, change the line that says:

```
statusMessage = "Tap 'Get My Location' to Start"
```

To:

```
statusMessage = ""
showLogoView()
```

This new logic makes the logo appear when there are no coordinates or error messages to display. That's also the state at startup time, so when you run the app now, you should be greeted by the logo.

► Run the app to check it out.



When you tap the logo (or Get My Location), the logo should disappear and the panel with the labels ought to show up. That doesn't happen yet, so let's add some more code to do that.

- Add the following method:

```
func hideLogoView() {  
    logoVisible = false  
    containerView.isHidden = false  
    logoButton.removeFromSuperview()  
}
```

This is the counterpart to `showLogoView()`. For now, it simply removes the button with the logo and un-hides the container view with the GPS coordinates.

- Add the following to `getLocation()`, right after the authorization status checks:

```
if logoVisible {  
    hideLogoView()  
}
```

Before it starts the location manager, this first removes the logo from the screen if it was visible.

Currently, there is no animation code to be seen. When doing complicated layout stuff such as this, it's better to first make sure the basics work. If they do, you can make it look fancy with an animation afterwards.

- Run the app. You should see the screen with the logo. Press the Get My Location button and the logo is replaced by the coordinate labels.

Great! Now you can add the animation. The only method you have to change is `hideLogoView()`.

- First, give `CurrentLocationViewController` the ability to handle animation events by making it the `CAAnimationDelegate`:

```
class CurrentLocationViewController: UIViewController,  
    CLLocationManagerDelegate, CAAnimationDelegate {
```

- Then replace `hideLogoView()` with:

```
func hideLogoView() {  
    if !logoVisible { return }  
  
    logoVisible = false  
    containerView.isHidden = false
```

```
containerView.center.x = view.bounds.size.width * 2
containerView.center.y = 40 +
    containerView.bounds.size.height / 2

let centerX = view.bounds.midX

let panelMover = CABasicAnimation(keyPath: "position")
panelMover.isRemovedOnCompletion = false
panelMover.fillMode = CAMediaTimingFillMode.forwards
panelMover.duration = 0.6
panelMover.fromValue = NSValue(CGPoint: containerView.center)
panelMover.toValue = NSValue(CGPoint:
    CGPoint(x: centerX, y: containerView.center.y))
panelMover.timingFunction = CAMediaTimingFunction(
    name: CAMediaTimingFunctionName.easeOut)
panelMover.delegate = self
containerView.layer.add(panelMover, forKey: "panelMover")

let logoMover = CABasicAnimation(keyPath: "position")
logoMover.isRemovedOnCompletion = false
logoMover.fillMode = CAMediaTimingFillMode.forwards
logoMover.duration = 0.5
logoMover.fromValue = NSValue(CGPoint: logoButton.center)
logoMover.toValue = NSValue(CGPoint:
    CGPoint(x: -centerX, y: logoButton.center.y))
logoMover.timingFunction = CAMediaTimingFunction(
    name: CAMediaTimingFunctionName.easeIn)
logoButton.layer.add(logoMover, forKey: "logoMover")

let logoRotator = CABasicAnimation(keyPath:
    "transform.rotation.z")
logoRotator.isRemovedOnCompletion = false
logoRotator.fillMode = CAMediaTimingFillMode.forwards
logoRotator.duration = 0.5
logoRotator.fromValue = 0.0
logoRotator.toValue = -2 * Double.pi
logoRotator.timingFunction = CAMediaTimingFunction(
    name: CAMediaTimingFunctionName.easeIn)
logoButton.layer.add(logoRotator, forKey: "logoRotator")
}
```

This creates three animations that are played at the same time:

1. The containerView is placed outside the screen (somewhere on the right) and moved to the center.
2. The logo image slides out of the screen.
3. The logo image also rotates around its center, giving the impression that it's rolling away.

Because the “panelMover” animation takes longest, you set a delegate on it so that you will be notified when the entire animation is over.

- Now add the necessary CAAnimationDelegate method:

```
// MARK:- Animation Delegate Methods
func animationDidStop(_ anim: CAAnimation,
                      finished flag: Bool) {
    containerView.layer.removeAllAnimations()
    containerView.center.x = view.bounds.size.width / 2
    containerView.center.y = 40 +
        containerView.bounds.size.height / 2
    logoButton.layer.removeAllAnimations()
    logoButton.removeFromSuperview()
}
```

This cleans up after the animations and removes the logo button, as you no longer need it.

- Run the app. Tap on Get My Location to make the logo disappear.

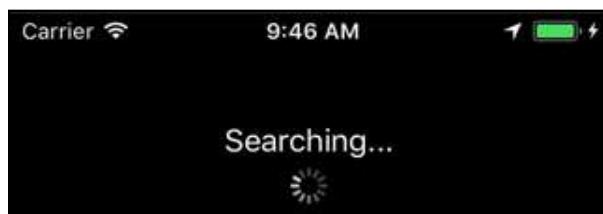
Tip: To get the logo back so you can try again, first choose **Location ▶ None** from the Simulator’s **Debug** menu. Then tap Get My Location followed by Stop to make the logo reappear.

Apple says that good apps should “surprise and delight,” and modest animations such as these really make your apps more interesting to use — as long as you don’t overdo it!

Adding an activity indicator

When the user taps the Get My Location button, you currently change the button’s text to say Stop to indicate the change of state. You can make it even clearer to the user that something is going on by adding an animated activity “spinner.”

It will look like this:



The animated activity spinner shows that the app is busy

UIKit comes with a standard control for this, `UIActivityIndicatorView`. You could add the spinner to the storyboard. However, it's good to learn different techniques and so you'll create the spinner in code this time.

The code to change the appearance of the Get My Location button sits in the `configureGetButton()` method. That's also a good place to show and hide the spinner.

► Replace `configureGetButton()` with the following:

```
func configureGetButton() {
    let spinnerTag = 1000

    if updatingLocation {
        getButton.setTitle("Stop", for: .normal)

        if view.viewWithTag(spinnerTag) == nil {
            let spinner = UIActivityIndicatorView(style: .white)
            spinner.center = messageLabel.center
            spinner.centerY += spinner.bounds.size.height/2 + 25
            spinner.startAnimating()
            spinner.tag = spinnerTag
            containerView.addSubview(spinner)
        }
    } else {
        getButton.setTitle("Get My Location", for: .normal)

        if let spinner = view.viewWithTag(spinnerTag) {
            spinner.removeFromSuperview()
        }
    }
}
```

In addition to changing the button text to "Stop," you create a new instance of `UIActivityIndicatorView`. Then you do some calculations to position the spinner view below the message label at the top of the screen. The call to `addSubview()` actually adds the spinner to the container view and makes it visible.

To keep track of this spinner view, you give it a tag of 1000. You could use an instance variable but this is just as easy and it keeps everything local to the `configureGetButton()` method. It's nice to have everything in one place.

When it's time to revert the button to its old state, you call `removeFromSuperview()` to remove the activity indicator view from the screen.

And that's all you need to do.

► Run the app. There should now be a cool little animation while the app is busy talking to the GPS satellites.

Making some noise

Visual feedback is important, but you can't expect users to keep their eyes glued to the screen all the time, especially if an operation might take a few seconds or more.

Emitting an unobtrusive sound is a good way to alert the user that a task is complete — for example, when your iPhone sends an email, you hear a soft “whoosh” sound.

You're going to add a sound effect to the app too, which is to be played when the first reverse geocoding successfully completes. That seems like a reasonable moment to alert the user that GPS and address information has been captured.

There are many ways to play sounds on iOS, but you're going to use one of the simplest: system sounds. The System Sound API is intended for short beeps and other notification sounds, which is exactly the type of sound that you want to play here.

► Add an import for `AudioToolbox`, the framework for playing system sounds, to the top of `CurrentLocationViewController.swift`:

```
import AudioToolbox
```

► Add a `soundID` instance variable:

```
var soundID: SystemSoundID = 0
```

Because writing just 0 would normally give you a variable of type `Int`, you explicitly mention the type that you want it to be: `SystemSoundID`. This is a numeric identifier — sometimes called a “handle” — that refers to a system sound object. 0 means no sound has been loaded yet.

► Add the following methods to the class:

```
// MARK:- Sound effects
func loadSoundEffect(_ name: String) {
    if let path = Bundle.main.path(forResource: name,
                                    ofType: nil) {
        let fileURL = URL(fileURLWithPath: path, isDirectory: false)
        let error = AudioServicesCreateSystemSoundID(
            fileURL as CFURL, &soundID)
        if error != kAudioServicesNoError {
            print("Error code \(error) loading sound: \(path)")
        }
    }
}
```

```
func unloadSoundEffect() {
    AudioServicesDisposeSystemSoundID(soundID)
    soundID = 0
}

func playSoundEffect() {
    AudioServicesPlaySystemSound(soundID)
}
```

The `loadSoundEffect()` method loads the sound file and puts it into a new sound object. The specifics don't really matter, but you end up with a reference to that object in the `soundID` instance variable.

- Call `loadSoundEffect()` from `viewDidLoad()`:

```
loadSoundEffect("Sound.caf")
```

- In `locationManager(_:_:didUpdateLocations:)`, in the geocoder's completion closure, change the following code:

```
if error == nil, let p = placemarks, !p.isEmpty {
    // New code block
    if self.placemark == nil {
        print("FIRST TIME!")
        self.playSoundEffect()
    }
    // End new code
    self.placemark = p.last!
} else {
    ...
}
```

The new `if` statement simply checks whether the `self.placemark` instance variable is `nil`, in which case this is the first time you've reverse geocoded an address. It then plays a sound using the `playSoundEffect()` method.

Of course, you shouldn't forget to add the actual sound effect to the project!

- Add the **Sound** folder from this app's Resources to the project. Make sure **Copy items if needed** is selected — click the Options button in the file open panel to reveal this option.
- Run the app and see if it makes some noise. The sound should only be played for the first address it finds — when you see the FIRST TIME! log message — even if more precise locations keep coming in afterwards.

Note: If you don't hear the sound on the Simulator, try the app on a device. Sometimes system sounds will not play on the simulators.



CAF audio files

The Sound folder contains a single file, **Sound.caf**. The **caf** extension stands for Core Audio Format, and it's the preferred file format for these kinds of short audio files on iOS.

If you want to use your own sound file but it is in a different format than CAF and your audio software can't save CAF files, then you can use the `afconvert` utility to convert the audio file. You need to run it from the Terminal:

```
$ /usr/bin/afconvert -f caff -d LEI16 Sound.wav Sound.caf
```

This converts the `Sound.wav` file into `Sound.caf`. You don't need to do this for the audio file from this app's Sound folder because that file is already in the correct format. But if you want to experiment with your own audio files, then knowing how to use `afconvert` might be useful.

By the way, iOS can play `.wav` files just fine, but `.caf` is more optimal.

The icon and launch images

The Resources folder for this app contains an **Icon** folder with the app icons.

- Import the icon images into the asset catalog — you can simply drag them from Finder into the **AppIcon** group. It's best to drag them one-by-one into their respective slots — if you drag the whole set of icons into the group at once, Xcode can get confused. You can see the icons of the asset catalog, here:

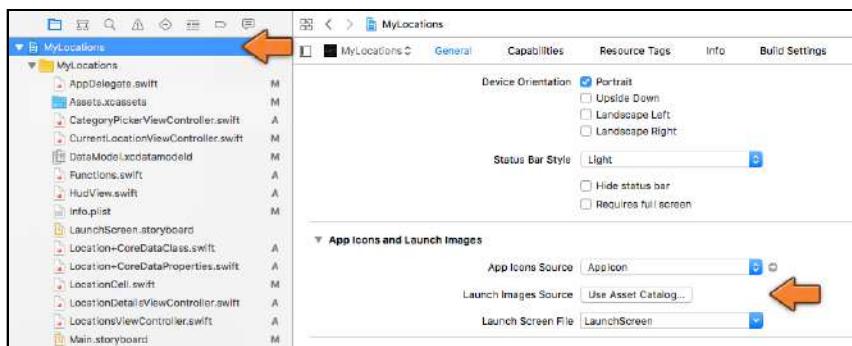


The icons in the asset catalog

The app currently also has a launch file, **LaunchScreen.storyboard**, that provides the splash image for when the app is still loading.

Instead of using a storyboard for the launch screen, you can also supply a set of images. Let's do that for this app.

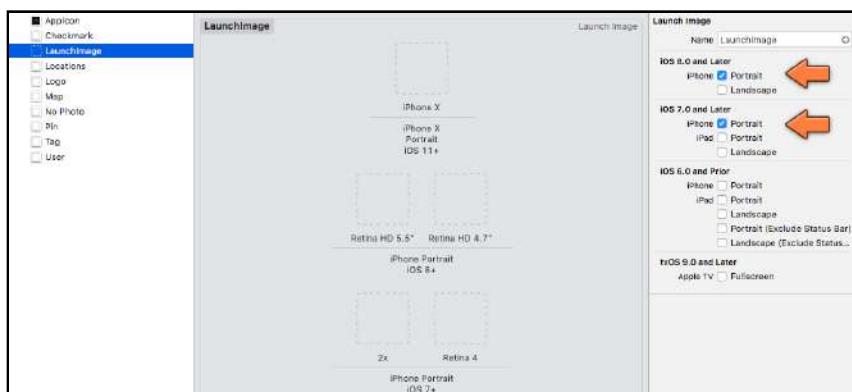
- In the **Project Settings** screen, in the **General** tab, find the **App Icons and Launch Images** section. Click the **Use Asset Catalog** button next to **Launch Images Source**:



Using the asset catalog for launch images

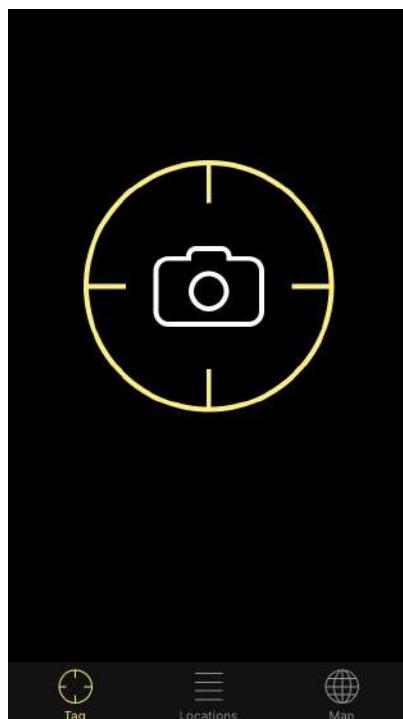
Xcode now asks if you want to migrate the launch images. Click **Migrate**.

- Clear the **Launch Screen File** text field.
- Also remove **LaunchScreen.storyboard** from the project. It's also a good idea to delete the app from the Simulator, or even reset it, so that there is no trace of the old launch screen.
- Open **Assets.xcassets**. There is now a **LaunchImage** item in the list. Select it and go to the Attributes inspector. Under both **iOS 8.0 and Later** and **iOS 7.0 and Later**, put checkmark by **iPhone Portrait**:



Enabling the launch images for iPhone portrait

You should now have five slots for dropping the launch images into — if you have any slots that say “Unassigned,” then select and remove them by pressing the delete key. The Resources folder for this app contains a **Launch Images** folder. Let’s take a look at one of those images, **Launch Image Retina 4.png**:



The launch image for this app

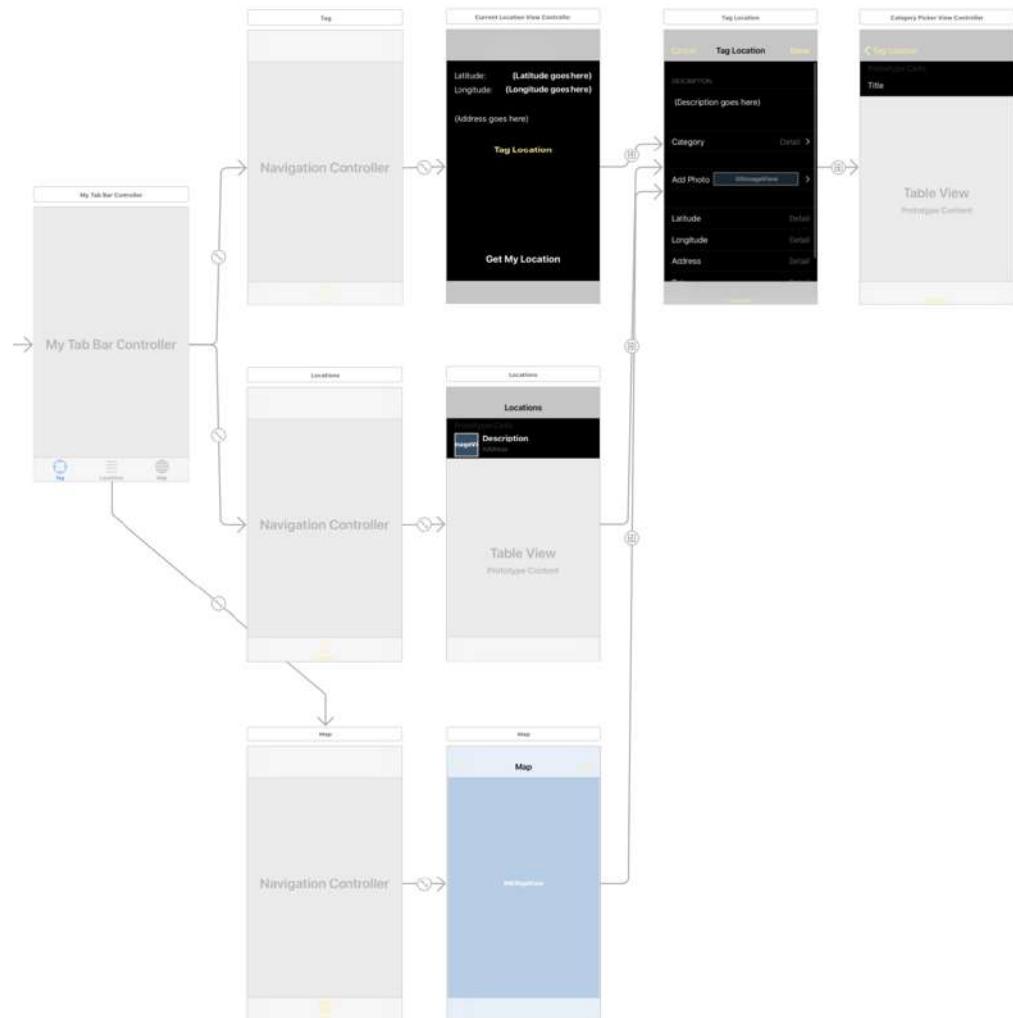
The launch image only has the tab bar and the logo button, but no status bar or any buttons. The reason it has no “Get My Location” button is that you don’t want users to try and tap it while the app is still loading since it’s not really a button! To make this launch image, you can run the app in the Simulator and choose **File ▶ Save Screen Shot**. This puts a new PNG file on the Desktop. You then open the image in Photoshop and blank out any text and the status bar portion of the image. The iPhone will draw its own status bar on top anyway.

- Drag the files from the **Launch Images** folder into the asset catalog, one at a time. The slot for each image should be pretty obvious.

Done. That was easy. And with that, *MyLocations* is complete! Woohoo! You can find the final project files for the app under **36 - Polishing the App** in the Source Code folder.

Congrats on making it this far! It has been a long and winding road with a lot of theory to boot.

The final storyboard for *MyLocations* looks like this:



Where to go from here?

In this section you took a more detailed look at Swift, but there's still plenty to discover. To learn more about the Swift programming language, you can read the following books:

- **The Swift Programming Language** by Apple. This is a free download on the iBooks Store. If you don't want to read the whole thing, at least take the Swift tour. It's a great introduction to the language.
- **Swift Apprentice** by the raywenderlich.com tutorial Team. This is a book that teaches you everything you need to know about Swift, from beginning to advanced topics. This is a sister book to the iOS Apprentice; the iOS Apprentice focuses more on making apps, while the Swift Apprentice focuses more on the Swift language itself. <https://store.raywenderlich.com/products/swift-apprentice>

There are several good Core Data beginner books on the market. Here are two recommendations:

- **Core Data by Tutorials** by the raywenderlich.com tutorial Team. One of the few Core Data books that is completely up-to-date with the lastest iOS and Swift versions. This book is for intermediate iOS developers who already know the basics of iOS and Swift development, but want to learn how to use Core Data to save data in their apps. <https://store.raywenderlich.com/products/core-data-by-tutorials>
- **Core Data Programming Guide** by Apple. If you want to get into the nitty gritty, then Apple's official guide is a must-read. You can learn a ton from this guide. <apple.co/2wNgiRu>

Credits for this tutorial:

- Sound effect based on a flute sample by elmomo, downloaded from The Freesound Project (<freesound.org>)
- Image resizing category is based on code by Trevor Harmon (<bit.ly/2wNGRX3>)
- HudView code is based on MBProgressHUD by Matej Bukovinski (<github.com/matej/MBProgressHUD>)

Are you ready for the final app? Then continue on to the next chapter, where you'll make an app that communicates with a web service over the network!

Section 5: Store Search

The final section of the book covers iPad support in more detail via the Store Search app.

Store Search shows you how to have separate custom screens both for specific orientations (landscape vs. portrait) as well as for specific platforms (iPhone vs. iPad). This section covers networking, working with remote API endpoints to fetch data needed by your app, and how to parse the fetched data. If that wasn't enough, this section also takes you through the full application life cycle — from developing the code, testing it, and all the way to submitting to Apple. So don't skip this section thinking that you know all about iOS development after the last few sections!

This section contains the following chapters:

37. Search Bar: One of the most common tasks for mobile apps is to talk to a server. In this final UIKit app you will build `StoreSearch`. In this chapter, you will build the first screens, add fake searches and create the data models.

38. Custom Table Cells: Before your app can search the iTunes store for real, we need to make the app look visually appealing. In this chapter, you will cover custom table view cells and nibs. Learn a little more about using git and the debugger right inside Xcode.

39. Networking: Networking you say? Start querying the iTunes web service by using HTTP requests. An introduction to JSON and best to convert them into data models and finally look at how best to sort results.



40. Asynchronous Networking: Phew! You will rarely want to block the main thread with a network request. In this chapter, we will explore asynchronous networking and finally showing an activity indicator to let the user know something is loading.

41. URLSession: The iOS toolbox and the Swift language has many tools for our disposal, including URLSession. In this chapter, we will explore URLSession and its many benefits. Downloading the iTunes artwork and how best to merge your git changes.

42. The Detail Pop-Up: In this chapter, we will create a detail pop-up view when a user taps a row in the TableView. We don't want to display too much information now, do we?

43. Polish the Pop-Up: We're about to get the polish back out again. The detail pop-up view is working well but we can display the information better. Learn about dynamic types, gradients for the background and let's explore adding some more animations.

44. Landscape: Users expect apps to work in both portrait and landscape. They also expect the app to look great in both orientations. In this chapter, we will learn about adding a completely different user interface for landscape vs. portrait.

45. Refactoring: The final app is looking great. You should put your feet up and grab a coffee! Programming is all about building new pretty features but when you join an existing company with an existing code-base you have to learn about the best ways to refactor existing code. Let's go!

46. Internationalization: So far our app works great in English. But if you want your app to go international you must support multiple languages and formats. In this chapter, you will explore adding support for a new language and look at regional settings.

47. The iPad: Even though the app works OK on the iPad, but it's not exactly optimized for the iPad. In this chapter, we're going to explore universal apps, the split view controller functionality, and dark mode support.

48. Distributing the App: Are you ready to ship to the App Store? Finally, you will learn the key fundamentals on how to ship the app to the App Store, including the Apple Developer Program, beta testing using TestFlight and finally submitting to the App Store.

37

Chapter 37: Search Bar

Eli Ganim

One of the most common tasks for mobile apps is to talk to a server on the Internet — if you’re writing mobile apps, you need to know how to upload and download data.

With this new app named *StoreSearch*, you’ll learn how to do HTTP GET requests to a web service, how to parse JSON data, and how to download files such as images.

You’re going to build an app that lets you search the iTunes store. Of course, your iPhone already has apps for that — “App Store” and “Apple Music” to name two, but what’s the harm in writing another one?

Apple has made a web service available for searching the entire iTunes store and you’ll be using that to learn about networking.

The finished app will look like this:



The finished StoreSearch app

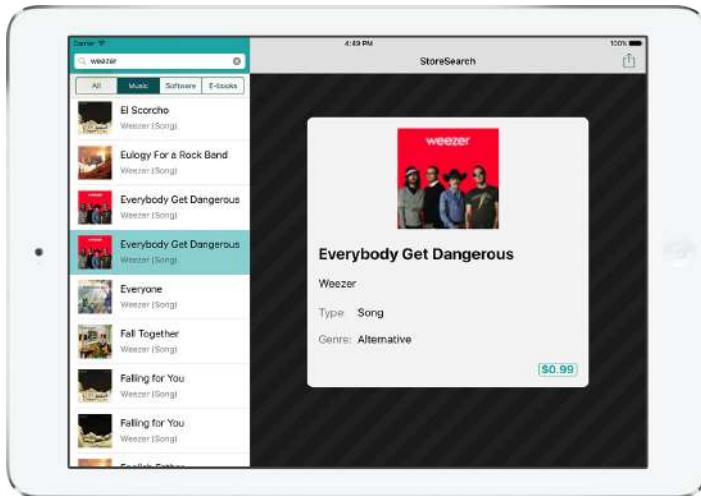
You will add search capability to your old friend, the table view. There is an animated pop-up with extra information when you tap an item in the table. And when you flip the iPhone over to landscape, the layout of the app completely changes to show the search results in a different way.

You will also add Dark Mode support, making the app look like this:



The finished StoreSearch app

Lastly, you'll create an iPad version of the app with a custom UI for the iPad:



The app on the iPad

StoreSearch fills in the missing pieces and rounds off the knowledge you have gained from developing the previous apps. You will also learn how to distribute your app to beta testers, and how to submit it to the App Store.

In this chapter, you will do the following:

- **Create the project:** Create a new project for your new app. Set up version control using Git.
- **Create the UI:** Create the user interface for *StoreSearch*.
- **Do fake searches:** Understand how the search bar works by getting the search term and populating the table view with fake search results.
- **Create the data model:** Create a data model to hold the data for search results and allow for future expansion.
- **No data found:** Handle "no data" situations when doing a search.

There's a lot of work ahead, so let's get started!

Creating the project

Fire up Xcode and create a new project. Choose the **Single View App** template and fill in the options as follows:

- Product Name: **StoreSearch**
- Team: Default value
- Organization Name: your name
- Organization Identifier: com.yourname
- Language: **Swift**
- Use SwiftUI, Use Core Data, Include Unit Tests, Include UI Tests: leave these unchecked

When you save the project Xcode gives you the option to create a **Git repository**. You've ignored this option thus far, but now you should enable it:



Creating a Git repository for the project

If you don't see this option, click the Options button at the bottom-left of the dialog.

Git and version control

Git is a **version control system** — it allows you to make snapshots of your work so you can always go back later and see a history of the changes made to the project. Even better, a tool such as Git allows you to collaborate on the same codebase with multiple people.

Imagine the chaos if two programmers changed the same source file at the same time. It's possible that your changes could accidentally be overwritten by a colleague's.

With a version control system such as Git, each programmer can work independently on the same files, without fear of undoing the work of another. Git is smart enough to automatically merge in all of the changes, and if there are any conflicting edits, it will let you resolve them manually.

Git is not the only version control system out there, but it's the most popular one for iOS. A lot of iOS developers share their source code on GitHub (github.com), a free collaboration site that uses Git as its engine. Another popular system is Subversion, often abbreviated as SVN. Xcode has built-in support for Git and while it used to support Subversion in past versions, that is no longer the case since Xcode 10.

For *StoreSearch*, you will use some basic Git functionality. Even if you work alone and don't have to worry about other programmers messing up your code, it still makes sense to use it. After all, you might be the one messing up your own code, and with Git, you'll always have a way to go back to your old — and working! — version of the code.

The first screen

The first screen in *StoreSearch* will have a table view with a search bar — let's create the view controller for that screen.

- In the Project navigator, select **ViewController.swift**, move your cursor over the `ViewController` class name and right-click to show the context menu. Select **Refactor > Rename...** from the menu and rename the class (and associated files and storyboard references) to `SearchViewController`.

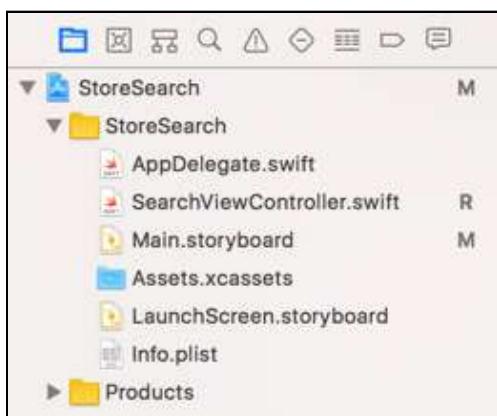
Note: Sometimes, the refactoring will do everything right except to rename your file correctly. If this happens, you'll see the new file name in red in the



Project navigator because Xcode expects the new file but the file actually has the old file name still. If this happens to you, simply go via Finder to your project folder and rename the file manually.

- Run the app to make sure everything works. You should see a white screen with the status bar at the top.

Notice that the project navigator now shows **M** and **R** icons next to some of the filenames in the list:



Xcode shows the files that are modified

If you don't see these icons, then choose the **Source Control ▶ Fetch and Refresh Status** option from the Xcode menu bar. If that gives an error message or still doesn't work, simply restart Xcode. That's a good tip in general: if Xcode is acting weird, restart it.

An **M** means the file has been modified since the last *commit* and an **R** means this is a file that has been renamed.

So what is a *commit*?

When you use a version control system such as Git, you're supposed to make a snapshot every so often. Usually you'll do that after you've added a new feature to your app or when you've fixed a bug, or whenever you feel like you've made changes that you want to keep. That is called a commit.

Git version control

When you created the project, Xcode made the initial commit. You can see that in the Project History window.

- Select the **Source Control navigator** from the Navigator pane and then click on the **project root** (the blue folder icon at the top) to see the project history:

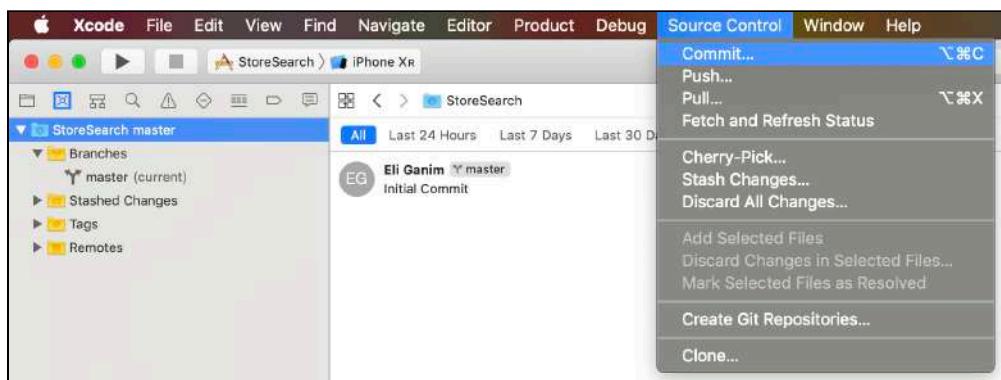


The history of commits for this project

You may get a pop-up at this point asking for permission to access your contacts. That allows Xcode to add contact information to the names in the commit history. This can be useful if you're collaborating with other developers.

You can always change this later under Security & Privacy in System Preferences.

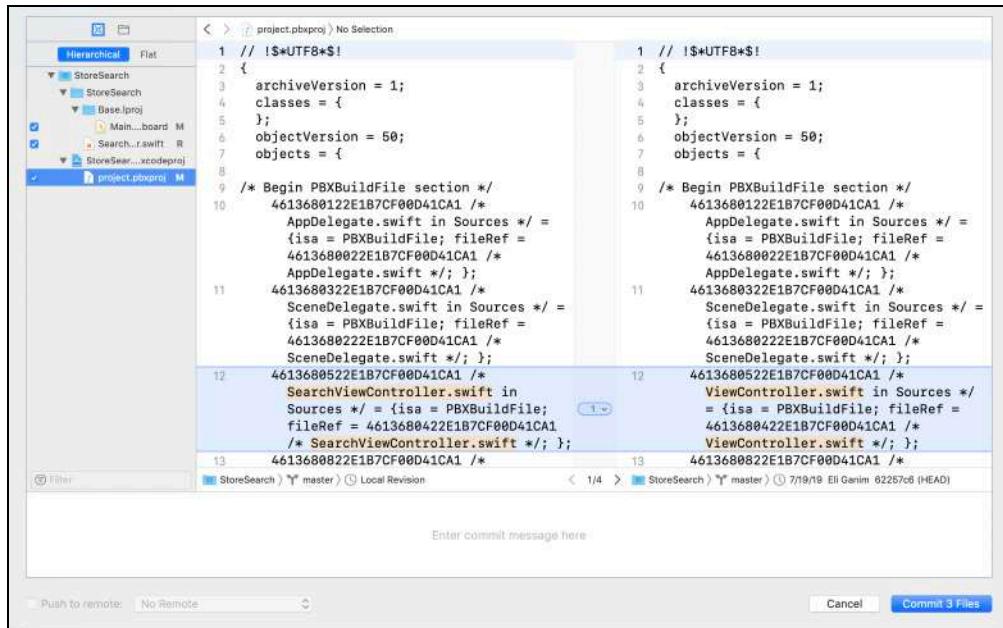
- Let's commit the change you just made. From the **Source Control** menu, choose **Commit...**:



The Commit menu option

This opens a new window that shows in detail what changes you made.

This is a good time to quickly review the code changes, just to make sure you're not committing anything you didn't intend to:



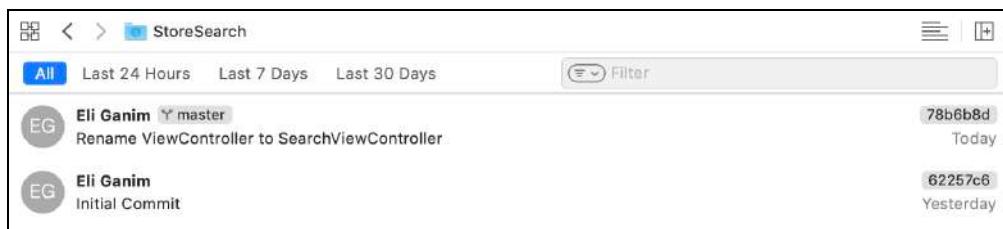
Xcode shows the changes you've made since the last commit

It's always a good idea to write a short but clear reason for the commit in the text box at the bottom. Having a good description here will help you later to find specific commits in your project's history.

► Write: **Rename ViewController to SearchViewController** as the commit message.

► Press the **Commit 3 Files** button. You'll see that in the Project navigator the M and R icons are gone — at least until you make your next change.

The Source Control navigator should now show two commits. If it doesn't, click on a different branch in the list and then click on the root folder again.

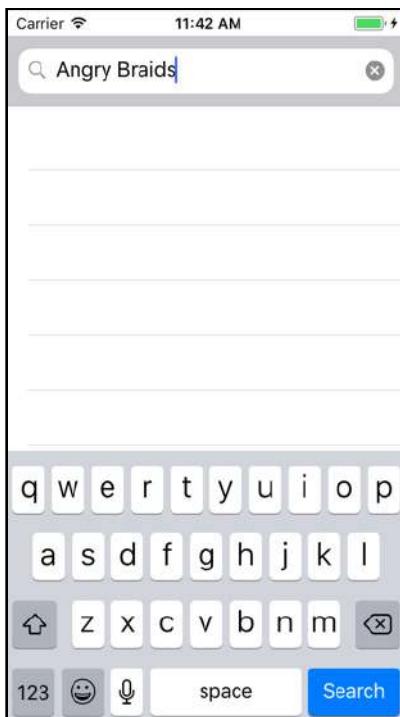


Your commit is listed in the project history

If you double-click a particular commit, Xcode will show you the changes for that commit. You'll be doing commits on a regular basis and by the end of the book you'll be a pro at it!

Creating the UI

StoreSearch still doesn't do much yet. In this section, you'll build the UI to look like this — a search bar on top of a table view:



The app with a search bar and table view

Even though this screen uses the familiar table view, it is not a *table view controller* but a regular `UIViewController` — check the class definition in `SearchViewController.swift`, if you are not sure.

You are not required to use a `UITableView` as the base class for your view controller just because you have a table view in your UI. You'll see how in this app.

UITableViewController vs. UIViewController

So what exactly is the difference between a *table* view controller and a regular view controller?

First off, `UITableViewController` is a subclass of `UIViewController` — it can do everything that a regular view controller can. However, it is optimized for use with table views and has some cool extra features.

For example, when a table cell contains a text field, tapping that text field will bring up the on-screen keyboard. `UITableViewController` automatically scrolls the cells out of the way of the keyboard so you can always see what you're typing.

You don't get that behavior for free with a plain `UIViewController` — if you want that feature, you'll have to program it yourself.

`UITableViewController` does have a big restriction: its main view must be a `UITableView` that takes up the entire screen space, except for a possible navigation bar at the top, and a toolbar or tab bar at the bottom.

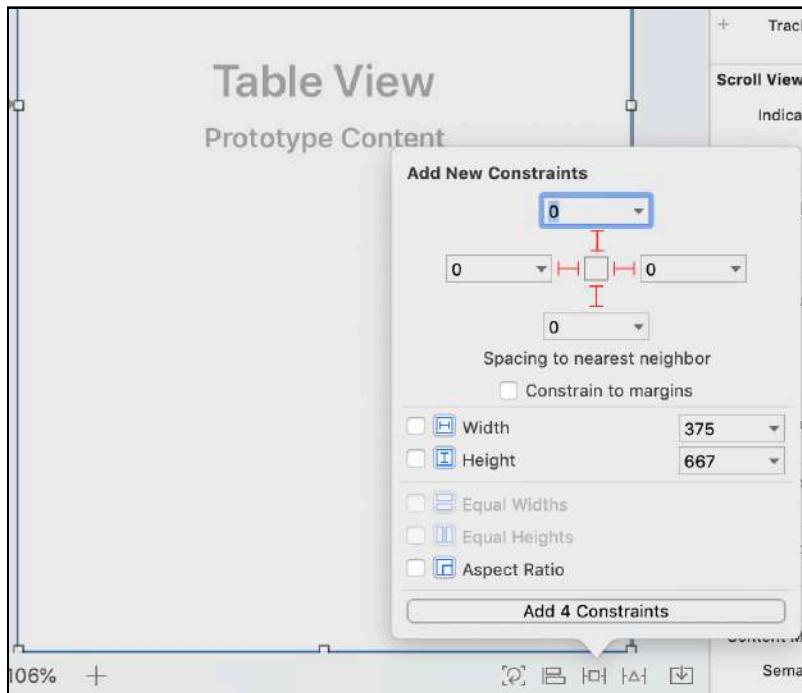
If your screen consists of just a `UITableView`, then it makes sense to make it a `UITableViewController`. But if you want to have other views as well, the more basic `UIViewController` is the option to go with.

That's the reason you're not using a `UITableViewController` in this app. Beside the table view, the app has another view, a `UISearchBar`. It is possible to put the search bar *inside* the table view as a special header view, or have the searchbar appear as part of the navigation bar, but for this app you will have it sitting above the table view.

Setting up the storyboard

- Open the storyboard and use the **View as:** panel to switch to the **iPhone 8** dimensions. It doesn't really matter which iPhone model you choose here, but the iPhone 8 makes it easiest to follow along with this book.
- Drag a new **Table View** — *not* a Table View Controller — into the existing view controller.

- Make the Table View as big as the main view (375 by 667 points) and then use the **Add New Constraints** menu at the bottom to attach the Table View to the edges of the screen:



Creating constraints to pin the Table View

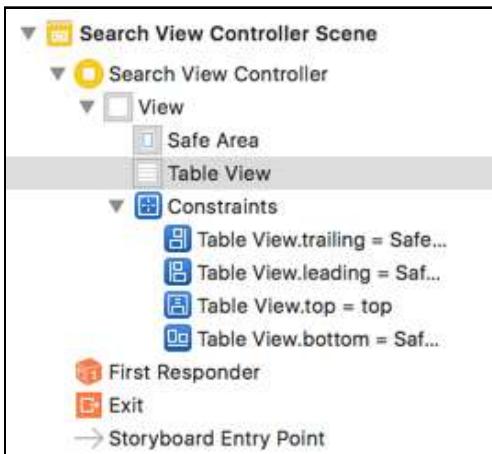
Remember how this works? This app uses Auto Layout, which you've used for the previous apps. With Auto Layout you create **constraints** that determine how big the views are and where they go on the screen.

- First, uncheck **Constrain to margins**, if it is checked. Each screen has 16-point margins on the left and right, but you can change their size. When “Constrain to margins” is enabled you’re pinning to these margins. That’s no good here; you want to pin the Table View to the edge of the screen instead.
- In the **Spacing to nearest neighbor** section, select the red I-beams to make four constraints, one on each side of the Table View. Keep the spacing values at 0.

This pins the Table View to the edges of its superview. Now the table will always fill up the entire screen, regardless of the size of the device screen.

- Click the **Add 4 Constraints** button to finish.

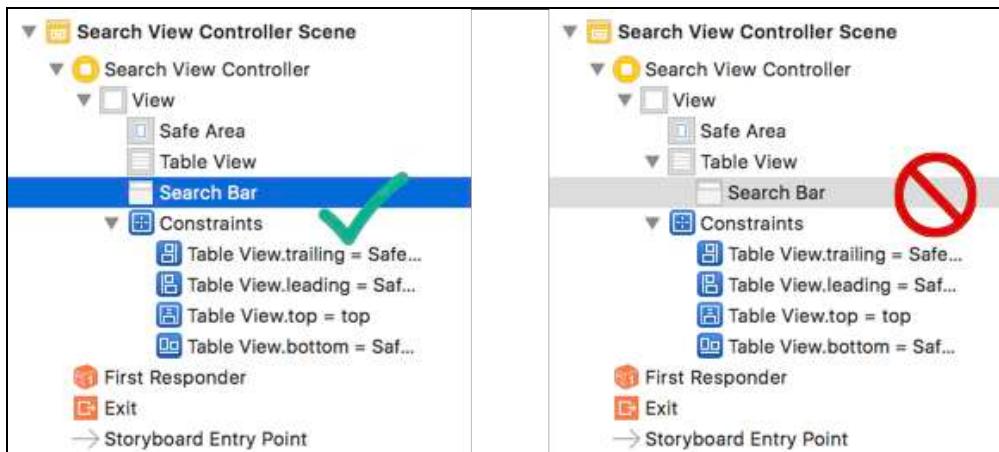
If you were successful, there should now be four blue bars surrounding the table view, one for each constraint. In the Document outline there should also be a new Constraints section.



The new constraints in the Document Outline

- From the Objects Library, drag a **Search Bar** on to the view – be careful to pick the Search Bar and not “Search Bar and Search Display Controller”. Place it at Y = 20 so it sits right under the status bar.

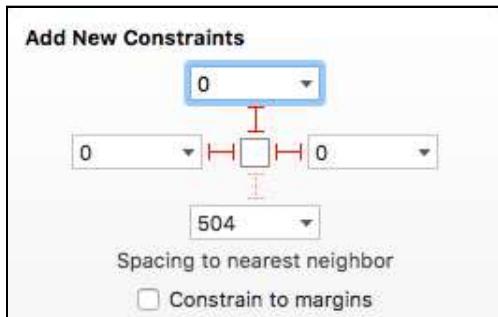
Make sure the Search Bar is not placed inside the table view. It should sit on the same level as the table view in the Document Outline:



Search Bar must be below of Table View (left), not inside (right)

If you did put the Search Bar inside the Table View, you can pick it up in the Document Outline and drag it below the Table View.

- Pin the Search Bar to the **top**, **left**, and **right** edges – 3 constraints in total.

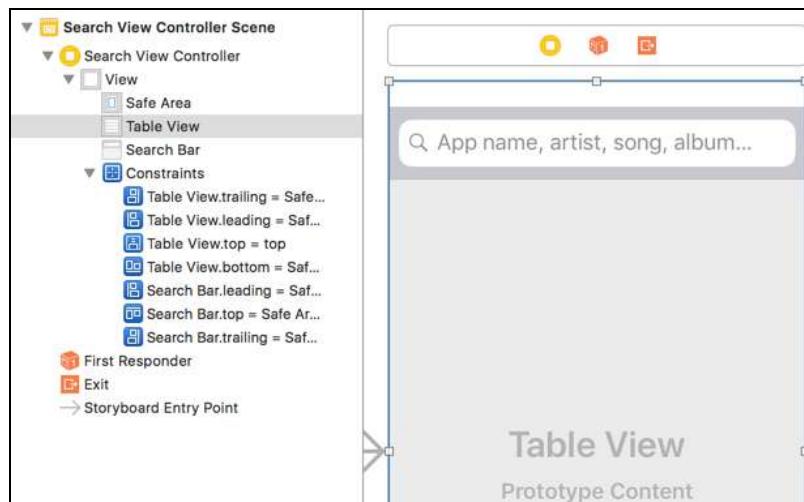


The constraints for the Search Bar

You don't need to pin the bottom of the Search Bar or give it a height constraint. Search Bars have an *intrinsic* height of 44 points.

- In the **Attributes inspector** for the Search Bar, change the **Placeholder** text to **App name, artist, song, album, e-book**.

The view controller's design should look like this:



The search view controller with Search Bar and Table View

Connecting to outlets

You know what's coming next: connecting the Search Bar and the Table View to outlets on the view controller.

- Add the following outlets to **SearchViewController.swift**:

```
@IBOutlet weak var searchBar: UISearchBar!
@IBOutlet weak var tableView: UITableView!
```

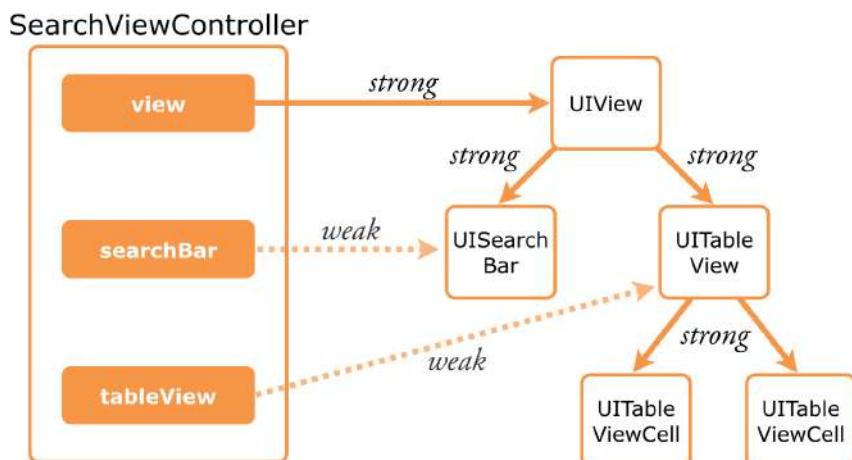
Recall that as soon as an object no longer has any strong references, it goes away — it is deallocated — and any weak references to it become `nil`.

Per Apple's recommendation, you've been making your outlets weak. You may be wondering, if the references to these view objects are weak, then won't the objects get deallocated too soon?

Exercise: What is keeping these views from being deallocated?

Answer: Views are always part of a view hierarchy and they will always have an owner with a strong reference — their superview.

The `SearchViewController`'s main view object holds a reference to both the search bar and the table view. This is done inside UIKit and you don't have to worry about it. As long as the view controller exists, so will these two outlets.



Outlets can be weak because the view hierarchy already has strong references

- Switch back to the storyboard and connect the Search Bar and the Table View to their respective outlets — Control-drag from the view controller to the object that you want to connect.

Table view content insets

If you run the app now, you'll notice a small problem: the first rows of the Table View are hidden beneath the Search Bar.



The first row is only partially visible

That's not so strange because you put the Search Bar on top of the table, obscuring part of the table view below.

You could fix this in several different ways:

1. Change the table view's top layout constraint to match the search bar's bottom edge.
2. Make the Search Bar partially translucent to let the contents of the table cells shine through.
3. Use the table view's **content inset** attribute to allow for the area covered by the search bar.

You will go with option #3. Unfortunately, the content inset attribute is unavailable via Interface Builder. So, this has to be done from code.

- Add the following line to the end of `viewDidLoad()` in `SearchViewController.swift`:

```
tableView.contentInset = UIEdgeInsets(top: 64, left: 0,  
bottom: 0, right: 0)
```

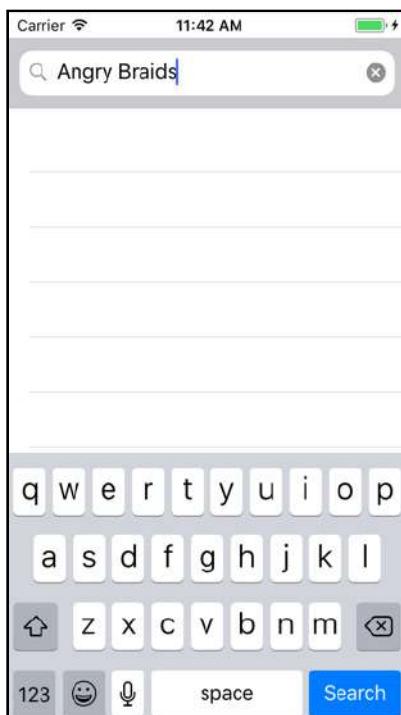
This tells the table view to add a 64-point margin at the top — 20 points for the status bar and 44 points for the Search Bar. Now the first row will always be visible, and when you scroll the table view, the cells still go under the search bar. Nice.

Doing fake searches

Before you implement the iTunes store searching, it's good to understand how the `UISearchBar` component works. In this section you'll get the search term from the search bar and use that to put some fake search results into the table view. Once you've got that working, you can build in the web service. Baby steps!

- Run the app. If you tap the search bar, the on-screen keyboard will appear — if you're on the simulator, you may need to press `⌘K` to bring up the keyboard, and `Shift+⌘K` to allow typing from your Mac keyboard.

However, it won't do anything when you type in a search term and tap the Search button.



Keyboard with Search button

Listening to the search bar is done — how else? — with a delegate. Let's put this delegate code into an extension.

Adding a search bar delegate

- Add the following to the bottom of **SearchViewController.swift**, after the final closing bracket:

```
extension SearchViewController: UISearchBarDelegate {  
    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
        print("The search text is: '\(searchBar.text!)'")  
    }  
}
```

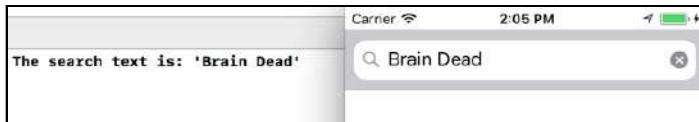
Recall that you can use extensions to organize your source code. By putting all the `UISearchBarDelegate` stuff into its own extension, you keep it together in one place and out of the way of the rest of the code.

The `UISearchBarDelegate` protocol has a method `searchBarSearchButtonClicked(_ :)` that is invoked when the user taps the Search button on the keyboard. You will implement this method to put some fake data into the table. Later, you'll make this method send a network request to the iTunes store to find songs, movies and e-books that match the search text that the user typed, but let's not do too many new things at once!

At the moment, all the new code does is to output the search term from the search bar to the Xcode Console.

Tip: It's recommended to put strings between single quotes when you use `print()`. That way you can easily see whether there are any trailing or leading spaces in the string. Also note that `searchBar.text` is an optional, so we need to unwrap it. It will never actually return `nil`, so a `!` will do just fine.

- In the storyboard, **Control-drag** from the Search Bar to Search View Controller, or the yellow circle at the top. Connect to **delegate**.
- Run the app, type something in the search bar and press the Search button. The Xcode Debug pane should now print the text you typed.



The search text in the Xcode Console

Showing fake results

- Add the following new (and empty) extension to **SearchViewController.swift**:

```
extension SearchViewController: UITableViewDelegate,  
    UITableViewDataSource {  
}
```

The above extension will handle all the table view related delegate methods. You could certainly have added them as two separate extensions if you liked, but some developers prefer to keep all the table view delegate related code in one place.

Adding the `UITableViewDataSource` and `UITableViewDelegate` protocols wasn't necessary for the previous apps because you used a `UITableViewController` in each case. `UITableViewController` already conforms to these protocols by necessity.

`SearchViewController` however, is a regular view controller and therefore you have to hook up the data source and delegate protocols yourself.

- Xcode should be complaining at this point that your code does not conform to the `UITableViewDataSource` protocol. Use the Xcode "Fix" option to add protocol stubs and then modify the code as follows to add the minimum code you need for the moment:

```
extension SearchViewController: UITableViewDelegate,  
    UITableViewDataSource {  
    func tableView(_ tableView: UITableView,  
                  numberOfRowsInSection section: Int) -> Int {  
        return 0  
    }  
  
    func tableView(_ tableView: UITableView,  
                  cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
        return UITableViewCell()  
    }  
}
```

This simply tells the table view that it has no rows yet. Soon you'll give it some fake data to display, but for now you just want to be able to compile the code without errors.

Often you can declare to conform to a protocol without implementing any of its methods — for example, this works fine for `UISearchBarDelegate`.

A protocol can have optional and required methods. If you forget a required method, you'll generally see Xcode complain, like you did above.



- In the storyboard, **Control-drag** from the Table View to Search View Controller. Connect to **dataSource**. Repeat to connect to **delegate**.

In case you’re wondering how you connected something to a **delegate** property in Search View Controller twice — first the Search Bar, and then the Table View — the way Interface Builder presents this is a little misleading: the delegate outlet is not from **SearchViewController**, but belongs to the thing that you Control-dragged from. So you connected the **SearchViewController** to the **delegate** outlet on the Search Bar and also to the **delegate** (and **dataSource**) outlets on the Table View:



The connections from Search View Controller to the other objects

- Build and run the app to make sure everything still works.

Note: Did you notice a difference between these data source methods and the ones from the previous apps? Look closely...

Answer: They don't have the `override` keyword.

In the previous apps, `override` was necessary because you were dealing with a subclass of `UITableViewController`, which already provides its own version of the `tableView(_:numberOfRowsInSection:)` and `tableView(_:cellForRowAt:)` methods.

In those apps, you were “overriding” or replacing those methods with your own versions, hence the need for the `override` keyword.

Here, however, your base class is not a table view controller but a regular `UIViewController`. Such a view controller doesn’t have any table view methods yet, so you’re not overriding anything here.

As you know by now, a table view needs some kind of data model. Let's start with a simple `Array`.

- Add an instance variable for the array —this goes inside the `class` brackets, not in any of the extensions:

```
var searchResults = [String]()
```

The search bar delegate method will put some fake data into this array and then display it using the table.

- Replace the `searchBarSearchButtonClicked(_:)` method with:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
    searchResults = []  
    for i in 0...2 {  
        searchResults.append(String(format:  
            "Fake Result %d for '%@'", i, searchBar.text!))  
    }  
    tableView.reloadData()  
}
```

Here the notation `[]` means you instantiate a new `String` array and replace the contents of `searchResults` property with it.

This is done each time the user performs a search. If there was already a previous array of results, then that is thrown away and deallocated. You could also have written `searchResults = [String]()` to do the same thing.

You add a string with some text into the array. Just for fun, that is repeated 3 times so your data model will have three rows in it.

When you write `for i in 0...2`, it creates a loop that repeats three times because the *closed range* `0...2` contains the numbers 0, 1, and 2. Note that this is different from the *half-open range* `0..2`, which only contains 0 and 1. You could also have written `1...3` but as you've discovered by now, programmers like to start counting at 0.

You've seen format strings before. The format specifier `%d` is a placeholder for integer numbers. Likewise, `%f` is for floating-point numbers. The placeholder `%@` is for all other kinds of objects, such as strings.

The last statement in the method reloads the table view to make the new rows visible, which means you have to adapt the data source methods to read from this array as well.

- Replace the methods in the table view delegate extension with the following:

```
func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return searchResults.count
}

func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "SearchResultCell"

    var cell:UITableViewCell! = tableView.dequeueReusableCell(
        forCellReuseIdentifier: cellIdentifier)
    if cell == nil {
        cell = UITableViewCell(style: .default,
            reuseIdentifier: cellIdentifier)
    }

    cell.textLabel!.text = searchResults[indexPath.row]
    return cell
}
```

All of the above code should be pretty familiar to you by now. You simply return the number of rows to display based on the contents of the `searchResults` array and you create a `UITableViewCell` by hand to display the table rows.

- Run the app. If you search for anything, a few fake results get added to the data model and are shown in the table.

Search for something else and the table view updates with new fake results.



The app shows fake results when you search

UI Improvements

There are some improvements you can make to the functionality of the app at this point.

Dismissing keyboard on search

It's not very nice that the keyboard stays on screen after you press the Search button. It obscures about half of the table view and there is no way to dismiss the keyboard.

- Add the following line to the top of `searchBarSearchButtonClicked(_:)`:

```
searchBar.resignFirstResponder()
```

This tells the `UISearchBar` that it should no longer listen for keyboard input. As a result, the keyboard will hide itself until you tap on the search bar again.

You can also configure the table view to dismiss the keyboard with a gesture.

- In the storyboard, select the Table View. Go to the **Attributes inspector** and set **Keyboard** to **Dismiss interactively**.

Extending search bar to status area

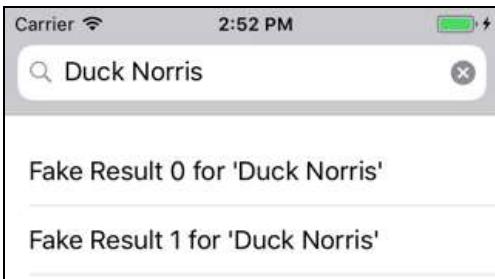
The search bar still has an ugly white gap above it for the status area. It would look a lot better if the status bar area was unified with the search bar. There's a delegate method for `UINavigationBar` and `UISearchBar` items which allows the item to indicate its top position.

- Add the following method to the `earchBarDelegate` extension:

```
func position(for bar: UIBarPositioning) -> UIBarPosition {  
    return .topAttached  
}
```



Now the app looks way better:



The search bar is “attached” to the top of the screen

If you were to look in the API documentation for `UISearchBarDelegate` you wouldn't find this `position(for:)` method. Instead, it is part of the `UIBarPositioningDelegate` protocol, which the `UISearchBarDelegate` protocol extends — like classes, protocols can inherit from other protocols.

The API documentation

Xcode comes with a big library of documentation for developing iOS apps. Basically everything you need to know is in here. Learn to use the Xcode documentation browser — it will become your best friend!

There are a few ways to get to the documentation for an item in Xcode.

There is Quick Help, which shows info about the item under the text cursor:

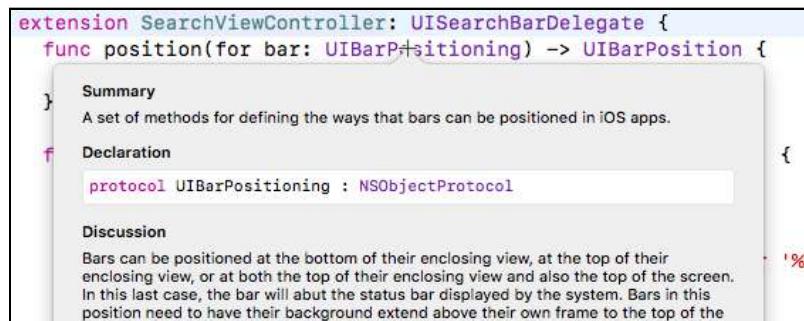
Quick Help
Summary
Asks the delegate for the position of the specified bar in its new window.

Declaration
optional func position(for:
bar: UIBarPositioning) ->
UIBarPosition

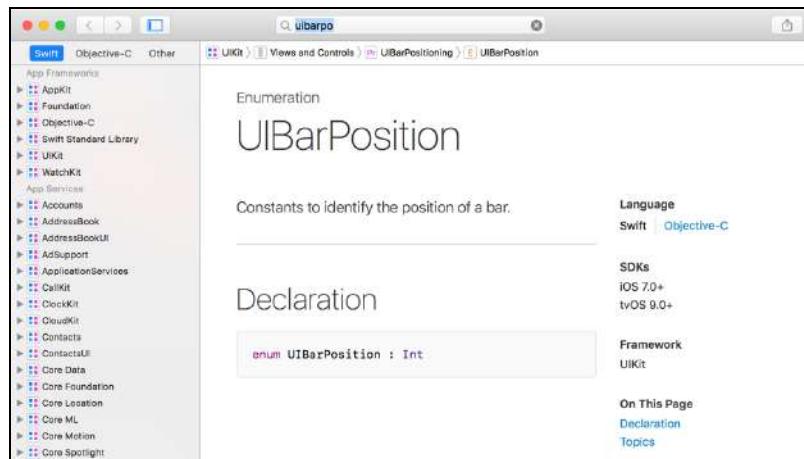
Discussion
If your interface has a custom bar with a delegate, that delegate can implement this method and use it to specify the position of the bar that has been added to a window.

Simply have the **Quick Help inspector** — the second tab in the inspector pane — open and it will show context-sensitive help. Put the text cursor on the item you want to know more about and the inspector will provide a summary. You can click any of the blue text links in the summary to jump to the full documentation.

You can also get pop-up help. Hold down the **Option** (Alt) key and hover over the item that you want to learn more about. Then click the mouse:



And of course, there is the full-fledged documentation window. You can access it from the **Help** menu, under **Developer Documentation**. Use the bar at the top to search for the item that you want to know more about:



Creating the data model

So far you've added `String` objects to the `searchResults` array, but that's a bit limited. The search results that you'll get back from the iTunes store include the product name, the name of the artist, a link to an image, the purchase price, and much more.

You can't fit all of that in a single string, so let's create a new class to hold this data.

The `SearchResult` class

► Add a new file to the project using the **Swift File** template. Name the new class `SearchResult`.

► Add the following to `SearchResult.swift`:

```
class SearchResult {
    var name = ""
    var artistName = ""
}
```

This adds two properties to the new `SearchResult` class. You'll add several others in a bit.

In `SearchViewController` you need to modify the `searchResults` array to hold instances of `SearchResult`.

► In `SearchViewController.swift`, change the declaration of the property:

```
var searchResults = [SearchResult]()
```

► Next, change the `for` `in` loop in the search bar delegate method to:

```
for i in 0...2 {
    let searchResult = SearchResult()
    searchResult.name = String(format: "Fake Result %d for", i)
    searchResult.artistName = searchBar.text!
    searchResults.append(searchResult)
}
```

This creates an instance of the `SearchResult` object and simply puts some fake text into its `name` and `artistName` properties. Again, you do this in a loop because just having one search result by itself is a bit sad.

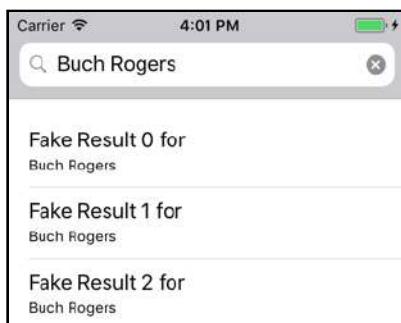


- At this point, `tableView(_:cellForRowAt:)` still expects the array to contain strings. So, update that method:

```
func tableView(_ tableView: UITableView,  
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
  
    if cell == nil {  
        cell = UITableViewCell(style: .subtitle, // change  
            reuseIdentifier: cellIdentifier)  
    }  
    // Replace all the code below this point  
    let searchResult = searchResults[indexPath.row]  
    cell.textLabel!.text = searchResult.name  
    cell.detailTextLabel!.text = searchResult.artistName  
    return cell  
}
```

Instead of a regular table view cell, the code now uses a “subtitle” cell style. You put the contents of the `artistName` property into the subtitle text label.

- Run the app; it should look like this:



Fake results in a subtitle cell

No results found

When you add search functionality to your apps, you have to handle the following situations:

1. The user did not perform a search yet.
2. The user performed the search and received one or more results. That's what happens in the current version of the app: for every search you'll get back a handful of `SearchResult` objects.

3. The user performed the search and there were no results. It's usually a good idea to explicitly tell the user there were no results. If you display nothing at all, the user may wonder whether the search was actually performed or not.

Even though the app doesn't do any actual searching yet, there is no reason why you cannot fake the last scenario as well.

Handling not getting any results

In defense of good taste, the app will return 0 results when a user searches for "justin bieber", just so you know the app can handle this kind of situation.

- In `searchBarSearchButtonClicked(_:)`, put the following `if` statement around the `for` in loop:

```
if searchBar.text! != "justin bieber" {  
    for i in 0...2 {  
        . . .  
    }  
    . . .
```

The change here is pretty simple — you've added an `if` statement that prevents the creation of any `SearchResult` objects if the text is equal to "justin bieber".

- Run the app and do a search for "justin bieber" — note the all lowercase. The table should remain empty.

At this point, you don't know if the search failed, or if there were no results. You can improve the user experience by showing the text "(Nothing found)" instead, so the user knows beyond a shadow of a doubt that there were no search results.

- Change the last part of `tableView(_:cellForRowAt:)` to:

```
if cell == nil {  
    . . .  
}  
// New code  
if searchResults.count == 0 {  
    cell.textLabel!.text = "(Nothing found)"  
    cell.detailTextLabel!.text = ""  
} else {  
    let searchResult = searchResults[indexPath.row]  
    cell.textLabel!.text = searchResult.name  
    cell.detailTextLabel!.text = searchResult.artistName  
}
```

```
// End of new code  
return cell
```

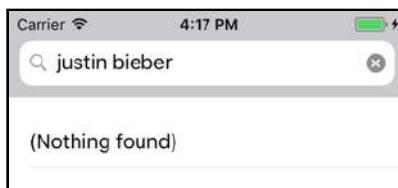
That alone is not enough. When there is nothing in the array, `searchResults.count` is 0, right? But that also means that `numberOfRowsInSection` will return 0 and the table view will stay empty — this “Nothing found” row will never show up.

► Change `tableView(_:numberOfRowsInSection:)` to:

```
func tableView(_ tableView: UITableView,  
    numberOfRowsInSection section: Int) -> Int {  
    if searchResults.count == 0 {  
        return 1  
    } else {  
        return searchResults.count  
    }  
}
```

Now, if there are no results, the method returns 1, for the row with the text “(Nothing Found)”. This works because both `numberOfRowsInSection` and `cellForRowAtIndex` check for this special situation.

► Try it out:



One can hope...

Handling no results when app starts

Unfortunately, the text “Nothing found” also appears initially when the user has not searched for anything yet. That’s just silly.

The problem is that you have no way to distinguish between “not searched yet” and “nothing found”. Right now, you can only tell whether the `searchResults` array is empty, but not what caused this.

Exercise: How would you solve this little problem?

There are two obvious solutions that come to mind:

- Change `searchResults` to an optional. If it is `nil`, i.e. it has no value, then the user hasn't searched yet. That's different from the case where the user did search and no matches were found.
- Use a separate boolean variable to keep track of whether a search has been done yet or not.

It may be tempting to choose the optional, but it's best to avoid optionals if you can. They complicate the logic, they can cause the app to crash if you don't unwrap them properly, and they require `if let` statements everywhere. Optionals certainly have their uses, but here they are not really necessary.

So, we'll opt for the boolean. But do feel free to come back and try the optional on your own, and compare the differences. It'll be a great exercise!

► Still in **SearchViewController.swift**, add a new instance variable:

```
var hasSearched = false
```

► In the search bar delegate method, set this variable to `true`. It doesn't really matter where you do this, as long as it happens before the table view is reloaded.

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    .
    .
    hasSearched = true // Add this line
    tableView.reloadData()
}
```

► And finally, change `tableView(_:numberOfRowsInSection:)` to look at the value of this new variable:

```
func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    if !hasSearched {
        return 0
    } else if searchResults.count == 0 {
        return 1
    } else {
        return searchResults.count
    }
}
```

Now, the table view remains empty until you first search for something. Try it out! Later on, you'll see a much better way to handle this using an enum and it will blow your mind!



Selection handling

One more thing, if you currently tap on a row it will become selected and stay selected.

- To fix that, add the following methods to the table view delegate extension:

```
func tableView(_ tableView: UITableView,  
              didSelectRowAt indexPath: IndexPath) {  
    tableView.deselectRow(at: indexPath, animated: true)  
}  
  
func tableView(_ tableView: UITableView,  
              willSelectRowAt indexPath: IndexPath) -> IndexPath? {  
    if searchResults.count == 0 {  
        return nil  
    } else {  
        return indexPath  
    }  
}
```

The `tableView(_:didSelectRowAt:)` method will simply deselect the row with an animation, while `willSelectRowAt` makes sure that you can only select rows when you have actual search results.

If you tap on the **(Nothing Found)** row now, you will notice that it does not turn gray at all. Actually, the row may still turn gray if you press down on it for a short while. That happens because you did not change the `selectionStyle` property of the cell. You'll fix that in a bit.

- This is a good time to commit your changes. Go to **Source Control ▶ Commit** — or press the **⌘+Option+C** keyboard shortcut.

Make sure all the modified files are selected/checked in the list on the left, review your changes, and type a good commit message — something like “Add a search bar and table view. The search puts fake results in the table for now”. Press the **Commit** button to finish.

Note: It is customary to write commit messages in the present tense. That's why the message says “Add a search bar” instead of “Added a search bar”.

Versions editor

If you ever want to look back through your commit history, you can do that from the Source Control navigator — as you learned how at the beginning of this chapter — or from the **Version editor**, pictured below:



Viewing revisions in the Version editor

You switch to the Version editor using the relevant toolbar button on the top right of the Xcode window.

In the screenshot above, the previous version is shown on the left and the current version on the right. You can switch between versions using the jump bar at the bottom of each pane. The Version editor is a very handy tool for viewing the history of changes in your source files.

The app isn't very impressive yet, but you've laid the foundation for what is to come. You have a search bar and know how to take action when the user presses the Search button. The app also has a simple data model that consists of an array with `SearchResult` objects, and it can display these search results in a table view.

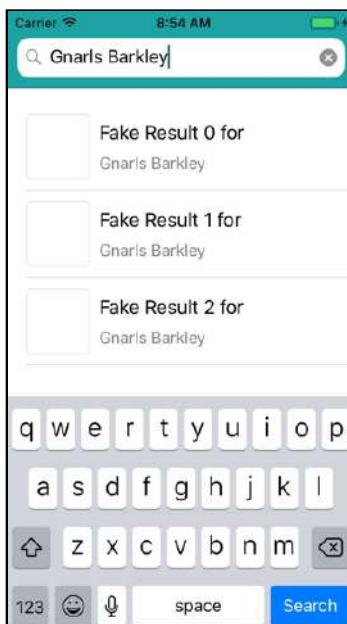
You can find the project files for this chapter under **37 – Search Bar** in the Source Code folder.

Chapter 38: Custom Table Cells

Eli Ganim

Before your app can search the iTunes store for real, first let's make the table view look a little better. Appearance does matter when it comes to apps!

Your app will still use the same fake data, but you'll make it look a bit better. This is what you'll have by the end of this chapter:



The app with better looks

In the process, you will learn the following:

- **Custom table cells and nibs:** How to create, configure and use a custom table cell via nib file.
- **Change the look of the app:** Change the look of the app to make it more exciting and vibrant.
- **Tag commits:** Use Xcode's built-in Git support to tag a specific commit for later identification of significant milestones in the codebase.
- **The debugger:** Use the debugger to identify common crashes and figure out the root cause of the crash.

Custom table cells and nibs

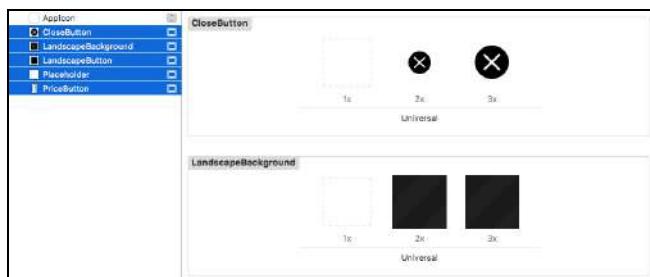
For the previous apps, you used prototype cells to create your own table view cell layouts. That works great, but there's another way. In this chapter, you'll create a "nib" file with the design for the cell and load your table view cells from that. The principle is very similar to prototype cells.

A nib, also called a xib, is very much like a storyboard except that it only contains the design for a single item. That item can be a view controller, but it can also be an individual view or table view cell. A nib is really nothing more than a container for a "freeze dried" object that you can edit in Interface Builder.

In practice, many apps consist of a combination of nibs and storyboard files, so it's good to know how to work with both.

Adding assets

► First, add the contents of the **Images** folder from this app's resources into the project's asset catalog, **Assets.xcassets**.

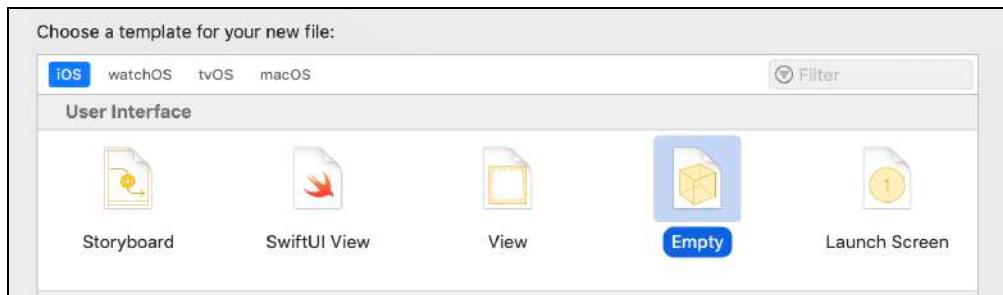


Imported images in the asset catalog

Each of the images comes in two versions: 2x and 3x. There are no low-resolution 1x devices that can run the latest version of iOS. So there's no point in including 1x images.

Adding a nib file

- Add a new file to the project. Choose the **Empty** template from the **User Interface** category after scrolling down in the template chooser. This will create a new empty nib.



Adding an empty nib to the project

- Click **Next** and save the new file as **SearchResultCell**.

Open **SearchResultCell.xib** and you will see an empty canvas.

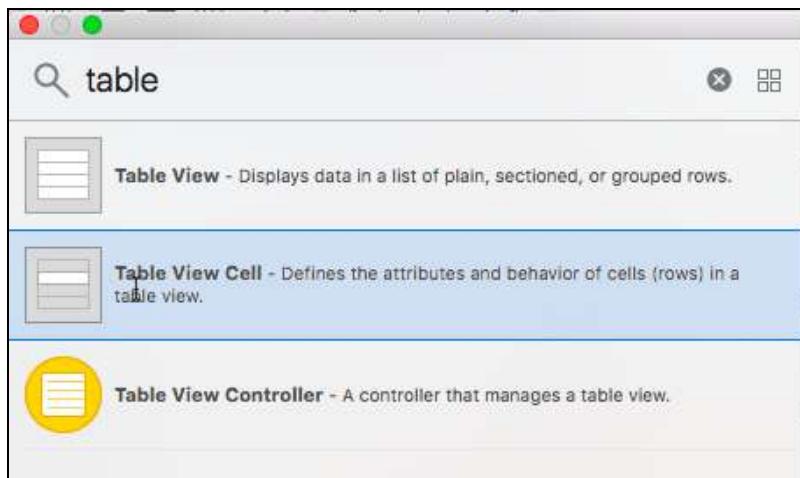
Xib or nib

I've been calling it a nib but the file extension is **.xib**. So what is the difference? In practice, these terms are used interchangeably. Technically speaking, a xib file is compiled into a nib file that is put into your application bundle. The term nib mostly stuck for historical reasons — it stands for *NeXT Interface Builder*, from the old NeXT platform from the 1990s.

You can consider the terms “xib file” and “nib file” to be equivalent. The preferred term seems to be nib, so that is what will be used from now on. This won't be the last time computer terminology is confusing, ambiguous or inconsistent. The world of programming is full of jargon.

- Use the **View as:** panel to switch to **iPhone 8** dimensions. As usual, you'll design for this device but use Auto Layout to make the user interface adapt to larger devices/screens.

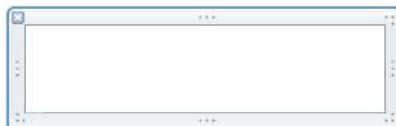
- From the Objects Library, drag a new **Table View Cell** on to the canvas:



The Table View Cell in the Objects Library

- Select the new Table View Cell and go to the **Size inspector**. Type 80 in the **Height** field (not Row Height). Make sure **Width** is 375, the width of the iPhone 8 screen.

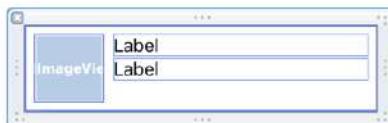
The cell now looks like this:



An empty table view cell

Note: Sometimes, you might have a blue bounding rectangle for the cell which is slightly offset from the actual cell's location. This is an Interface Builder bug. If this happens to you, simply switch to some other file and then switch back to the SearchResultCell.xib — all should be well at this point.

- Drag an **Image View** and two **Labels** into the cell, like this:



The design of the cell

Note: If you get blue rectangles around each item like above — or would like to get the rectangles to see the full bounds of each item — then use the **Editor ▶ Canvas ▶ Show Bounds Rectangles** menu item to toggle the bounds rectangles on/off.

- Position the Image View at **X:16, Y:10, Width:60, Height:60**.
- Set the **Text** of the first label to **Name**, **Font** to **System 18**, **X:84, Y:16, Width:220, Height:22**.
- Set the **Text** for the second label to **Artist Name**, **Font** to **System 15**, **Color** to **black with 50% opacity**, **X:84, Y:44, Width:220, Height:18..**

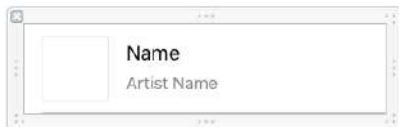
As you can see, editing a nib is just like editing a storyboard. The difference is that the canvas is a lot smaller because you’re only editing a single table view cell, not an entire view controller.

- The Table View Cell itself needs to have a reuse identifier. You can set this in the **Attributes inspector** to **SearchResultCell**.

The image view will hold the artwork for the found item, such as an album cover, book cover, or an app icon. It may take a few seconds for these images to be loaded, so until then, it’s a good idea to show a placeholder image. That placeholder is part of the image files you just added to the project.

- Select the Image View. In the **Attributes inspector**, set **Image** to **Placeholder**.

The cell design should now look like this:



The cell design with placeholder image

You’re not done yet. The design for the cell is only 320 points wide but there are iOS devices with screens wider than that. The cell itself will resize to accommodate those larger screens, but the labels won’t, potentially causing their text to be cut off. You’ll have to add some Auto Layout constraints to make the labels resize along with the cell.

Setting up Auto Layout constraints

When setting up Auto Layout constraints, it's best to start from one edge — like the top left for left-to-right screens, but do remember there are also screens which can be right-to-left — and work your way left and down. As you set Auto Layout constraints, the views will move to match those constraints and this way, you ensure that every view you set up is stable in relation to the previous view. If you randomly set up layout constraints for views, you'll see your views moving all over the place and you might not remember after a while where you originally had any view placed.

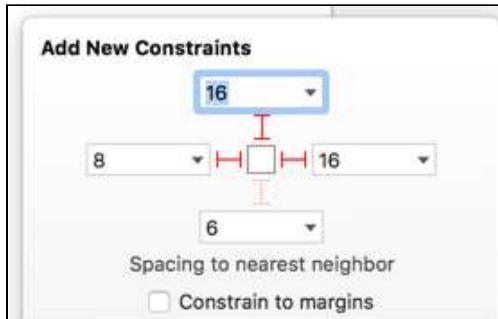
- Select the **Image View** and open the **Add New Constraints** menu. Uncheck **Constrain to margins** and pin the Image View to the **top** and **left** sides of the cell. Also give it **Width** and **Height** constraints so that its size is always fixed at 60 by 60 points:



The constraints for the Image View

- Click **Add 4 Constraints** to actually add the constraints.

- Select the **Name** label and again use the **Add New Constraints** menu. Uncheck **Constrain to margins** and select the **top**, **left**, and **right** pins (but not the bottom one):



The constraints for the Name label

- Click **Add 3 Constraints**.

- Finally, pin the **Artist Name** label to the **left**, **top**, **right** and **bottom** — again without constraining to margins — as above by adding 4 new constraints.

That concludes the design for this cell. Now you have to tell the app to use this nib.

Registering nib file for use in code

- In **SearchViewController.swift**, add these lines to the end of `viewDidLoad()`:

```
let cellNib = UINib(nibName: "SearchResultCell", bundle: nil)
tableView.register(cellNib, forCellReuseIdentifier:
    "SearchResultCell")
```

The `UINib` class is used to load nibs. Here, you tell it to load the nib you just created — note that you don't specify the `.xib` file extension. Then you ask the table view to register this nib for the reuse identifier “`SearchResultCell`.”

From now on, when you call `dequeueReusableCell(withIdentifier:)` for the identifier “`SearchResultCell`,” `UITableView` will automatically make a new cell from the nib — or reuse an existing cell if one is available, of course. And that's all you need to do.

- In `tableView(_:cellForRowAt:)` change this bit of code:

```
let cellIdentifier = "SearchResultCell"
var cell: UITableViewCell! = tableView.dequeueReusableCell(
```

```
        withIdentifier: cellIdentifier)
if cell == nil {
    cell = UITableViewCell(style: .subtitle,
    reuseIdentifier: cellIdentifier)
}
```

So that the final method looks like this:

```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "SearchResultCell", for: indexPath)
    if searchResults.count == 0 {
        .
        .
    }
    return cell
}
```

You were able to replace a chunk of code with just one statement. Now, it's almost exactly like using prototype cells, except that you have to create your own nib object and you need to register it with the table view beforehand.

Note: The call to `dequeueReusableCell(withIdentifier:)` now takes a second parameter, `for:`, that takes an `IndexPath` value. This variant of the `dequeue` method lets the table view be a bit smarter, but it only works when you have registered a nib with the table view — or when you use a prototype cell.

- Run the app and do a (fake) search. Yikes, the app crashes.

Exercise: Any ideas why?

Answer: Because you made your own custom cell design, you cannot use the `textLabel` and `detailTextLabel` properties of `UITableViewCell`.

Every table view cell — even a custom cell that you load from a nib — has a few labels and an image view of its own, but you should only employ these when you're using one of the standard cell styles: `.default`, `.subtitle`, etc. If you use them on custom cells, then these built-in labels get in the way of your own labels.



In this case, you shouldn't use `textLabel` and `detailTextLabel` to put text into the cell — you need to make your own properties for your labels.

Where do you put these properties? In a new class, of course. You're going to make a new class named `SearchResultCell` which extends `UITableViewCell` and has properties — and logic — for displaying the search results in this app.

Adding a custom `UITableViewCell` subclass

- Add a new file to the project using the **Cocoa Touch Class** template. Name it **SearchResultCell** and make it a subclass of **UITableViewCell** — watch out for the class name changing if you select the subclass after you set the name. “Also create XIB file” should be unchecked as you already have one.

This creates the Swift file to accompany the nib file you created earlier.

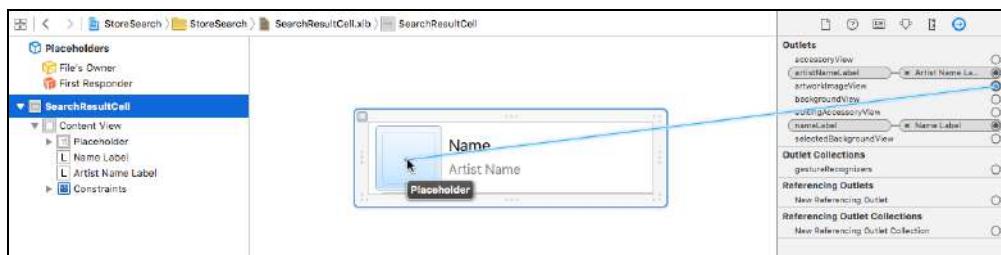
- Open **SearchResultCell.xib** and select the Table View Cell — make sure you select the actual Table View Cell object, not its Content View.
- In the **Identity inspector**, change its class from “`UITableViewCell`” to **SearchResultCell**.

You do this to tell the nib that the top-level view object it contains is no longer a `UITableViewCell` but your own `SearchResultCell` subclass. From now on, whenever you call `dequeueReusableCell()`, the table view will return an object of type `SearchResultCell`.

- Add the following outlet properties to **SearchResultCell.swift**:

```
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var artistNameLabel: UILabel!
@IBOutlet weak var artworkImageView: UIImageView!
```

- Hook these outlets up to the respective labels and image view in the nib. It is easiest to do this from the Connections inspector for `SearchResultCell`:



Connect the labels and image view to Search Result Cell

You can also open the Assistant editor and Control-drag from the labels and image view to their respective outlet definitions. If you've used nib files before you might be tempted to connect the outlets to File's Owner but that won't work in this case; they must be connected to the table view cell. Now that this is all set up, you can tell the SearchViewController to use these new SearchResultCell objects.

Using custom table view cell in app

- In `SearchViewController.swift`, change `cellForRowAt` to:

```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
        "SearchResultCell", for: indexPath)
    as! SearchResultCell
    if searchResults.count == 0 {
        cell.nameLabel.text = "(Nothing found)"
        cell.artistNameLabel.text = ""
    } else {
        let searchResult = searchResults[indexPath.row]
        cell.nameLabel.text = searchResult.name
        cell.artistNameLabel.text = searchResult.artistName
    }
    return cell
}
```

Notice the change in the first line. Previously this returned a `UITableViewCell` object, but now that you've changed the class name in the nib, you're guaranteed to always receive a `SearchResultCell` — you still need to cast it with `as!`, though.

Given that cell, you can put the name and artist name from the search result into the proper labels. You're now using the cell's `nameLabel` and `artistNameLabel` outlets instead of `textLabel` and `detailTextLabel`. You also no longer need to write `!` to unwrap because the outlets are implicitly unwrapped optionals.

- Run the app and it should look something like this:



Much better!

There are a few more things to improve. Notice that you've been using the string literal "SearchResultCell" in a few different places? It's generally better to create a constant for such occasions.

Using a constant for table cell identifier

Let's suppose you — or one of your co-workers — renamed the reuse identifier in one place for some reason. Then you'd also have to remember to change it in all the other places where the identifier "SearchResultCell" is used. It's better to limit those changes to one single spot by using a symbolic name instead.

- Add the following to **SearchViewController.swift**, somewhere within the class definition:

```
struct TableView {  
    struct CellIdentifiers {  
        static let searchResultCell = "SearchResultCell"  
    }  
}
```

This defines a new struct, `TableView`, containing a secondary struct named `CellIdentifiers` which contains a constant named `searchResultCell` with the value "SearchResultCell".

Should you want to change this value, then you only have to do it here and any code that uses `TableView.CellIdentifiers.searchResultCell` will be automatically updated. There is another reason for using a symbolic name rather than the actual value: it gives extra meaning. Just seeing the text "SearchResultCell" says less about its intended purpose than the symbol `TableView.CellIdentifiers.searchResultCell`.

Note: Putting symbolic constants as `static let` members inside a `struct` — or a series of `structs` — is a common trick in Swift. A static value can be used without an instance so you don't need to instantiate `TableView.CellIdentifiers` before you can use it — like you would need to do with a class.

It's allowed in Swift to place a `struct` *inside* a class, which permits different classes to all have their own `TableView.CellIdentifier` structs. This wouldn't work if you placed the `struct` outside the class — then you'd have multiple `structs` with the same name in the global namespace, which is not allowed.



- In **SearchViewController.swift**, replace the string "SearchResultCell" with `TableView.CellIdentifiers.searchResultCell`.

For example, `viewDidLoad()` will now look like this:

```
override func viewDidLoad() {  
    . . .  
    let cellNib = UINib(nibName:  
        TableView.CellIdentifiers.searchResultCell, bundle: nil)  
    tableView.register(cellNib, forCellReuseIdentifier:  
        TableView.CellIdentifiers.searchResultCell)  
}
```

The other change is in `tableView(_:cellForRowAt:)`.

- Run the app to make sure everything still works.

A new “No results” cell

Remember our friend Justin Bieber? Searching for him now looks like this:

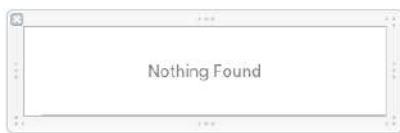


The Nothing Found label now looks like this

That's not very pretty — not to mention slightly off. It would be nicer if you gave this its own look. That's not too hard: you can simply make another nib for it.

- Add another nib file to the project. Again this will be an **Empty** nib. Name it **NothingFoundCell.xib**.
- Drag a new **Table View Cell** on to the canvas. Set its **Width** to 375, its **Height** to 80 and give it the reuse identifier **NothingFoundCell**.
- Drag a **Label** into the cell and give it the text **Nothing Found**. Make the text color 50% opaque black and the font **System 15**.
- Use **Editor ▶ Size to Fit Content** to make the label fit the text exactly — you may have to deselect and select the label again to enable the menu option.
- Center the label in the cell, using the blue guides to snap it exactly to the center.

It should look like this:



Design of the Nothing Found cell

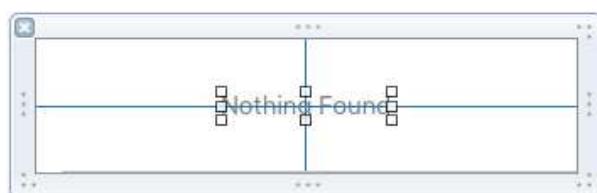
In order to keep the text centered on all devices, you can use the Auto Layout **Align menu**:



Creating the alignment constraints

- Choose **Horizontally in Container** and **Vertically in Container** and click **Add 2 Constraints**.

The constraints should look like this:



The constraints for the label

One more thing to fix. Remember that in `willSelectRowAt` you return `nil` if there are no search results to prevent the row from being selected? Well, if you are persistent enough you can still make the row appear gray as if it were selected.

For some reason, UIKit draws the selected background if you press down on the cell for long enough, even though this doesn't count as a real selection. To prevent this, you have to tell the cell not to use a selection color.

- Select the cell itself. In the **Attributes inspector**, set **Selection** to **None**. Now tapping or holding down on the Nothing Found row will no longer show any sort of selection.

You don't have to make a `UITableViewCell` subclass for this cell because there is no text to change or properties to set. All you need to do is register this nib with the table view.

- Add a new reuse identifier to the struct in `SearchViewController.swift`:

```
struct TableView {
    struct CellIdentifiers {
        static let searchResultCell = "SearchResultCell"
        static let nothingFoundCell = "NothingFoundCell" // New
    }
}
```

- Add these lines to `viewDidLoad()`, below the other code registering the nib:

```
cellNib = UINib(nibName:
    TableView.CellIdentifiers.nothingFoundCell, bundle: nil)
tableView.register(cellNib, forCellReuseIdentifier:
    TableView.CellIdentifiers.nothingFoundCell)
```

This also requires you to change `let cellNib` two lines up to `var` because you're re-using the `cellNib` local variable.

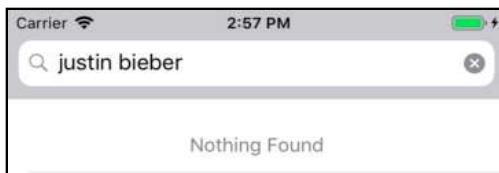
- And finally, change `tableView(_:cellForRowAt:)` to:

```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if searchResults.count == 0 {
        return tableView.dequeueReusableCell(withIdentifier:
            TableView.CellIdentifiers.nothingFoundCell,
            for: indexPath)
    } else {
        let cell = tableView.dequeueReusableCell(withIdentifier:
            TableView.CellIdentifiers.searchResultCell,
            for: indexPath) as! SearchResultCell
```

```
let searchResult = searchResults[indexPath.row]
cell.nameLabel.text = searchResult.name
cell.artistNameLabel.text = searchResult.artistName
return cell
}
}
```

The logic here has been restructured a little. You only make a `SearchResultCell` if there are actually any results. If the array is empty, you'll simply dequeue the cell for the `nothingFoundCell` identifier and return it since there is nothing to configure for that cell.

- Run the app. The search results for Justin Bieber now look like this:



The new Nothing Found cell in action

Also try it out on larger screen devices. The label should always be centered in the cell.

Sweet. It has been a while since your last commit, so this seems like a good time to secure your work.

Source Control changes

But before you commit your changes, take a look at `SearchViewController.swift` in your editor view. You might notice some blue lines along the gutter like this:

```
36  var hasSearched = false
37
38  struct TableView {
39      struct CellIdentifiers {
40          static let searchResultCell = "SearchResultCell"
41          static let nothingFoundCell = "NothingFoundCell"
42      }
43  }
44
45  override func viewDidLoad() {
46      super.viewDidLoad()
47      tableView.contentInset = UIEdgeInsets(top: 64, left: 0, bottom: 0, right: 0)
}
```



Source control change indicator in editor view

Whatever could those blue lines mean?

This is actually something new in Xcode 10 — those blue lines appear in projects which have source control enabled and they indicate the changes made by the developer since the last commit.

But it goes beyond that, if you work with other developers and somebody else made a change to the file you are working on and committed their change to Git, Xcode will even show these pending changes so that you are aware of changes made by somebody else that might impact the work you’re doing. Very handy!

- Commit the changes to the repository. You can use the message “Use custom cells for search results.”

Changing the look of the app

The app too looks quite gray and dull. Let’s cheer it up a little by giving it more vibrant colors.

- Add the following method to `AppDelegate.swift`:

```
// MARK:- Helper Methods
func customizeAppearance() {
    let barTintColor = UIColor(red: 20/255, green: 160/255,
                               blue: 160/255, alpha: 1)
    UISearchBar.appearance().barTintColor = barTintColor
}
```

This changes the appearance of the `UISearchBar` — in fact, it changes *all* search bars in the application. You only have one, but if you had several then this changes the whole lot in one fell swoop.

The `UIColor(red:green:blue:alpha:)` method makes a new `UIColor` object based on the RGB and alpha color components that you specify.

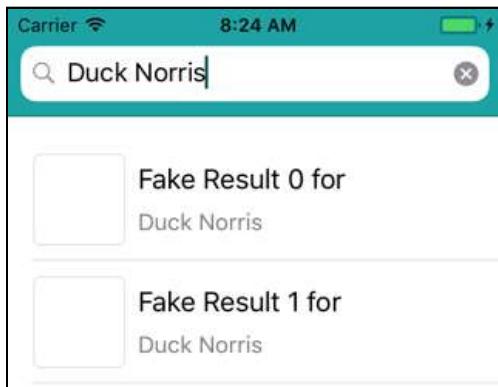
Many painting programs let you pick RGB values going from 0 to 255 so that’s the range of color values that many programmers are accustomed to thinking in. The `UIColor` initializer, however, accepts values between 0.0 and 1.0, so you have to divide these numbers by 255 to scale them down to that range.

- Call this new method from `application(_:didFinishLaunchingWithOptions:)`:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
```

```
customizeAppearance() // Add this line  
    return true  
}
```

- Run the app and notice the difference:



The search bar in the new teal-colored theme

The search bar is bluish-green, but still slightly translucent. The overall tint color is now a dark shade of green instead of the default blue — you can currently only see the tint color in the text field's cursor but it will become more obvious later on.

The role of App Delegate

The poor AppDelegate is often abused. People give it too many responsibilities. Really, there isn't that much for the app delegate to do.

It gets a number of callbacks about the state of the app — whether the app is about to be closed, for example — and handling those events should be its primary responsibility. The app delegate also owns the main window and the top-level view controller. Other than that, it shouldn't do much.

Some developers use the app delegate as their data model. That is just bad design. You should really have a separate class — or several — for that. Others make the app delegate their main control hub. Wrong again! Put that stuff in your top-level view controller.

If you ever see the following type of thing in someone's source code, it's a pretty good indication that the application delegate is being used the wrong way:

```
let appDelegate = UIApplication.shared.delegate as! AppDelegate  
appDelegate.someProperty = . . .
```

This happens when an object wants to get something from the app delegate. It works but it's not good architecture.

In my opinion, it's better to design your code the other way around: the app delegate may do a certain amount of initialization, but then it gives any data model objects to the root view controller, and hands over control. The root view controller passes these data model objects to any other controller that needs them, and so on.

This is also called *dependency injection*. This principle was described in Chapter 32 in the “Pass the context” section for the *MyLocations* app.

Changing the row selection color

Currently, tapping a row highlights it in gray. This doesn't go so well with the teal-colored theme. So, you'll give the row selection the same bluish-green tint.

As you learned with *MyLocations*, that's very easy to do because all table view cells have a `selectedBackgroundView` property. The view from that property is placed on top of the cell's background, but below the other content, when the cell is selected.

► Add the following code to `awakeFromNib()` in **SearchResultCell.swift**:

```
override func awakeFromNib() {
    super.awakeFromNib()
    // New code below
    let selectedView = UIView(frame: CGRect.zero)
    selectedView.backgroundColor = UIColor(red: 20/255,
                                           green: 160/255, blue: 160/255, alpha: 0.5)
    selectedBackgroundView = selectedView
}
```

The `awakeFromNib()` method is called after the cell object has been loaded from the nib but before the cell is added to the table view. You can use this method to do additional work to prepare the object for use. That's perfect for creating the view with the selection color.

Why don't you do that in an init method, such as `init?(coder)`? To be fair, in this case you could. But it's worth noting that `awakeFromNib()` is called some time after `init?(coder)` and also after the objects from the nib have been connected to their outlets.

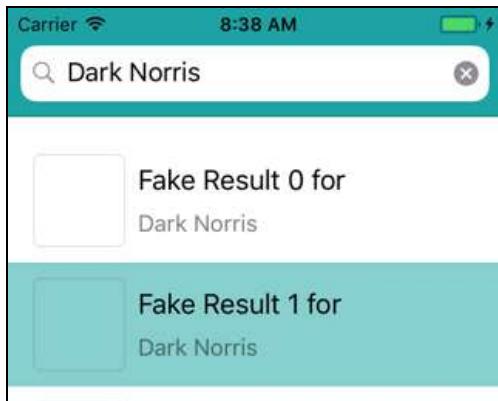
For example, in `init?(coder)` the `nameLabel` and `artistNameLabel` outlets will still be `nil` but in `awakeFromNib()` they will be properly hooked up to their `UILabel` objects. So, if you wanted to do something with those outlets in code, you'd need to do that in `awakeFromNib()`, not in `init?(coder)`.

That's why `awakeFromNib()` is the ideal place for this kind of thing — it's similar to how you use `viewDidLoad()` in a view controller.

Don't forget to first call `super.awakeFromNib()` — it is required. If you forget, then the superclass `UITableViewCell` — or any of the other superclasses — may not get a chance to initialize themselves.

Tip: It's always a good idea to call `super.methodName(...)` in methods that you're overriding — such as `viewDidLoad()`, `viewWillAppear()`, `awakeFromNib()`, and so on — unless the documentation says otherwise.

When you run the app, do a search and tap a row, it should look like this:



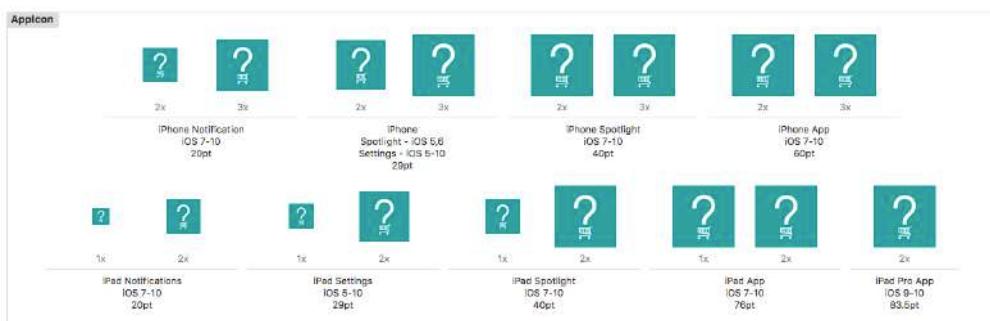
The selection color is now green

Adding app icons

While you're at it, you might as well give the app an icon.

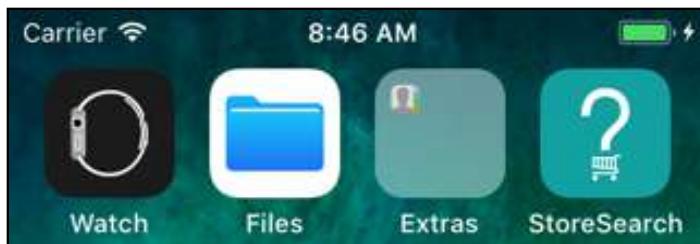
- Open the asset catalog (`Assets.xcassets`) and select the **AppIcon** group.
- Drag the images from the **Icon** folder from the Resources folder into the matching slots.

Keep in mind that for the 2x slots you need to use the image with twice the size in pixels. For example, you drag the **Icon-152.png** file into **iPad App 76pt, 2x**. For 3x you need to multiply the image size by 3.



All the icons in the asset catalog

- Run the app and notice that it now has a nice new icon:



The app icon

Showing keyboard on app launch

One final user interface tweak I'd like to make is that the keyboard should be immediately visible when you start the app so the user can start typing right away.

- Add the following line to `viewDidLoad()` in `SearchViewController.swift`:

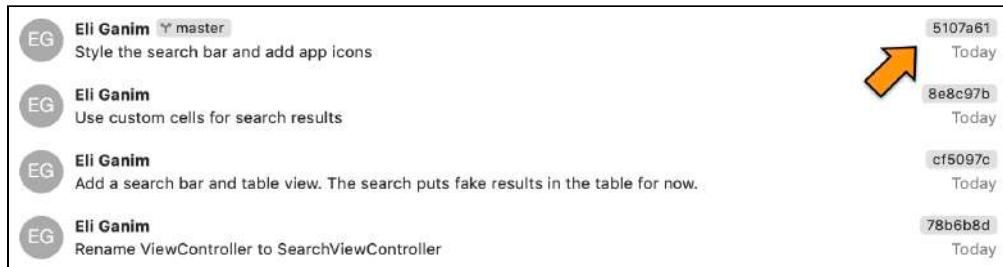
```
searchBar.becomeFirstResponder()
```

As you are aware from the *Checklists* app, `becomeFirstResponder()` will give `searchBar` the "focus" and show the keyboard. Anything you type will end up in the search bar.

- Try it out and commit your changes. You styled the search bar and added app icons.

Tagging commits

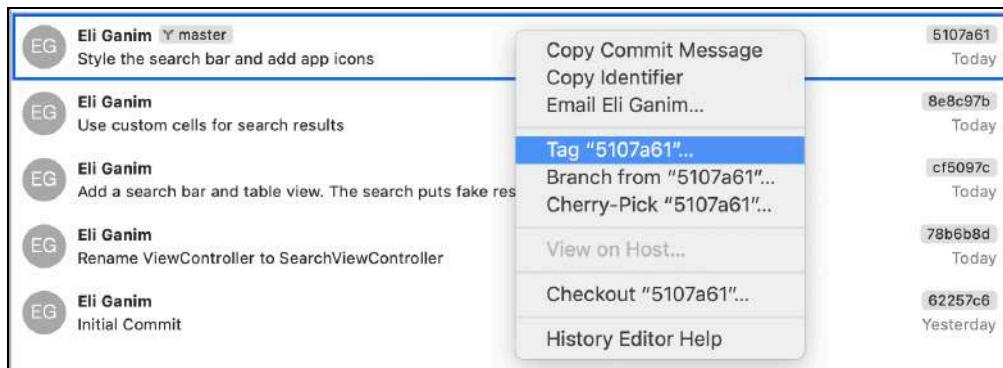
If you look through the various commits you've made so far, you'll notice a bunch of strange numbers, such as "5107a61":



The commits listed in the history window have weird numbers

Those are internal numbers — known as the *hash* — that Git uses to uniquely identify commits. Such numbers aren't very memorable, or useful, for us humans, so Git also allows you to "tag" a certain commit with a more friendly label.

- Tagging a commit in Xcode is as simple as selecting the commit in the Source Control navigator view, right-clicking to get the context menu and selecting the **Tag** option.



Tagging a commit in Xcode

- Enter "v0.1" as the **Tag**, and an optional message describing what this particular tag encompasses. Then click **Create** to create the tag.

You can see the new tag in the Source Control navigator view:

 Eli Ganim	Y master	v0.1	5107a61	Today
Style the search bar and add app icons				
 Eli Ganim	8e8c97b		Today	
Use custom cells for search results				

The new tag in Xcode

Xcode works quite well with Git, but you might want more power to do complex Git operations. If you do, you'll probably need to learn how to use the Terminal or get a tool such as SourceTree, which is available for free on the Mac App Store.

The debugger

Xcode has a built-in debugger. Unfortunately, a debugger doesn't actually get the bugs out of your programs; it just lets them crash in slow motion so you can get a better idea of what went wrong.

Like a detective, the debugger lets you dig through the evidence after the damage has been done, in order to find the scoundrel who did it. Thanks to the debugger, you don't have to stumble in the dark with no idea what just happened. Instead, you can use it to quickly pinpoint what went wrong and where. Once you know those two things, figuring out *why* it went wrong becomes a lot easier.

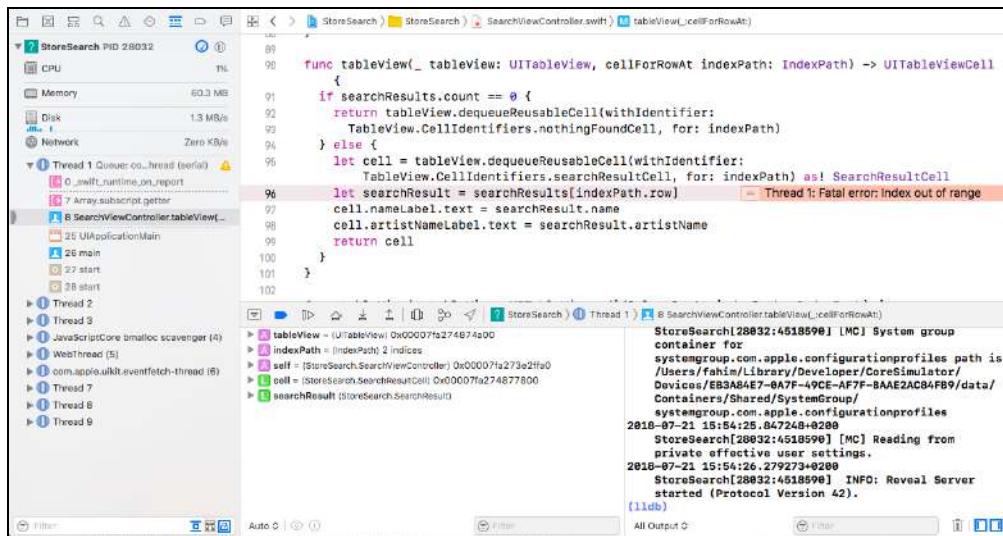
Indexing out of range bug

Let's introduce a bug into the app so that it crashes — knowing what to do when your app crashes is very important.

► Change `SearchViewController.swift`'s `numberOfRowsInSection` method to:

```
func tableView(_ tableView: UITableView,
              numberOfRowsInSection section: Int) -> Int {
    if !hasSearched {
        .
        .
    } else if searchResults.count == 0 {
        .
        .
    } else {
        return searchResults.count + 1 // This line changes
    }
}
```

- Now run the app and search for something. The app crashes and the Xcode window changes to something like this:



The Xcode debugger appears when the app crashes

The crash is: **Thread 1: Fatal error: Index out of range.** Sounds nasty!

According to the error message, the index that was used to access some array is larger than the number of items inside the array. In other words, the index is “out of range.” That is a common error with arrays and you’re likely to make this mistake more than once in your programming career.

Now that you know what went wrong, the big question is: *where did it go wrong?* You may have many calls to `array[index]` in your app, and you don’t want to have to dig through the entire code to find the culprit.

Thankfully, you have the debugger to help you out. In the source code editor it already points out the offending line:

```

95     let cell = tableView.dequeueReusableCell(withIdentifier:
        TableView.CellIdentifiers.searchResultCell, for: indexPath) as! SearchResultCell
96 |     let searchResult = searchResults[indexPath.row] // Thread 1: Fatal error: Index out of range
97     cell.nameLabel.text = searchResult.name
98     cell.artistNameLabel.text = searchResult.artistName

```

The debugger points at the line that crashed

Important: This line isn't necessarily the *cause* of the crash — after all, you didn't change anything in this method — but it is where the crash happens. From here you can trace backwards to the cause.

The array is `searchResults` and the index is given by `indexPath.row`. It would be great to get some insight into the row number and there are several ways to do this.

The one we'll look at here is to use the debugger's command line interface, like a hacker whiz kid from the movies.

- In the Xcode Console, after the **(lldb)** prompt, type **p indexPath.row** and press enter:



```
(lldb) p indexPath.row
(Int) $R1 = 3
(lldb)
```

Printing the value of indexPath.row

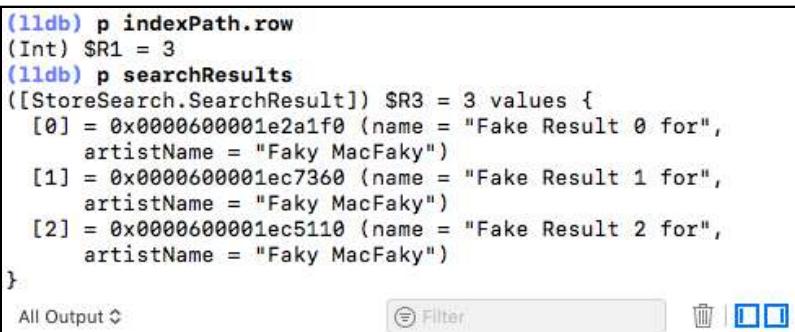
The output should be something like:

```
(Int) $R1 = 3
```

This means the value of `indexPath.row` is 3 and the type is `Int` — you can ignore the `$R1` bit.

Let's also find out how many items are in the array.

- Type **p searchResults** and press enter. If you use the auto complete functionality, do note that both `searchResult` — without the "s" at the end — and `searchResults` are choices. Be sure to select the correct one.:



```
(lldb) p indexPath.row
(Int) $R1 = 3
(lldb) p searchResults
([StoreSearch.SearchResult]) $R3 = 3 values {
    [0] = 0x0000600001e2a1f0 (name = "Fake Result 0 for",
        artistName = "Faky MacFaky")
    [1] = 0x0000600001ec7360 (name = "Fake Result 1 for",
        artistName = "Faky MacFaky")
    [2] = 0x0000600001ec5110 (name = "Fake Result 2 for",
        artistName = "Faky MacFaky")
}
All Output ◊
```

Printing the searchResults array

The output shows an array with three items.

You can now reason about the problem: the table view is asking for a cell for the fourth row — i.e. the one at index 3 — but apparently there are only three rows in the data model — rows 0 through 2.

The table view knows how many rows there are from the value that is returned from `numberOfRowsInSection`, so maybe that method is returning the wrong number of rows? That is indeed the cause, of course, as you intentionally introduced the bug in that method.

Hopefully this illustrates how you should deal with crashes: first find out where the crash happens and what the actual error is, then reason your way backwards until you find the cause.

Storyboard outlet bug

- Restore `numberOfRowsInSection` to its previous state and then add a new outlet property to `SearchViewController.swift`:

```
@IBOutlet weak var searchBar2: UISearchBar!
```

- Open the storyboard and **Control-drag** from Search View Controller to the Search Bar. Select `searchBar2` from the pop-up.

Now the search bar is also connected to this new `searchBar2` outlet — it's perfectly fine for an object to be connected to more than one outlet at a time.

- Delete the `searchBar2` outlet property from `SearchViewController.swift` in the source code, not the storyboard.

This is a dirty trick on my part to create another crash. The storyboard contains a connection to a property that no longer exists. If you think this a convoluted example, then wait until you make this mistake in one of your own apps. It happens more often than you may think!

- Run the app and it immediately crashes. The crash is “Thread 1: signal SIGABRT.”

Scrolling up the Xcode Console output in the Debug pane you should come across:

```
*** Terminating app due to uncaught exception
'NSUnknownKeyException', reason:
'<StoreSearch.SearchViewController 0x7fb83ec09bf0>
setValue:forUndefinedKey:]: this class is not key value coding-
compliant for the key searchBar2.'
*** First throw call stack:
(
    0   CoreFoundation                      0x0000000111da1c7b
__exceptionPreprocess + 171
    ...

```

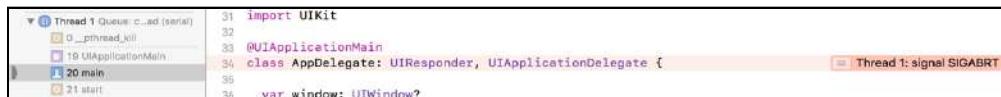
The first part of this message is very important: it tells you that the app was terminated because of an “`NSUnknownKeyException`.” On some platforms, exceptions are a commonly used error handling mechanism, but on iOS this is always a fatal error and the app is forced to halt.

The bit that should pique your interest is this:

```
this class is not key value coding-compliant for the key
searchBar2
```

Hmm, that is a bit cryptic. It does mention `searchBar2` but what does “key value-coding compliant” mean? I’ve seen this error enough times to know what is wrong, but if you’re new to this game, a message like that isn’t very enlightening.

So let’s see where Xcode thinks the crash happened:



Crash in AppDelegate?

That also isn’t very useful. Xcode says the app crashed in `AppDelegate`, but that’s not really true.

Xcode goes through the *call stack* until it finds a method that it has source code for and that’s the one it shows. The call stack is the list of methods that have been called most recently. You can see it on the left of the Debugger window.

- Click the left-most icon at the bottom of the Debug navigator to see more info.

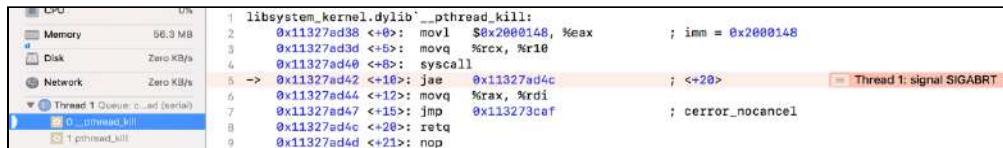


A more detailed call stack

The method at the top, `__pthread_kill`, was the last method that was called – it's actually a function, not a method.

It got called from `pthread_kill`, which was called from `abort`, which was called from `abort_message`, and so on, all the way back to the `main` function, which is the entry point of the app and the very first function that was called when the app started.

All of the methods and functions that are listed in this call stack are from system libraries, which is why they are grayed out. If you click on one, you'll get a bunch of unintelligible assembly code:



```

CPU          0%
Memory      56.3 MB
Disk        Zero KB/s
Network     Zero KB/s
Thread 1 Queue: c...ad (serial)
  0 __pthread_kill
    1 pthread_kill
      2 pthread_kill
        3 pthread_kill
          4 pthread_kill
            5 -> 0x11327ad42 <+8>: jae 0x11327ad4c ; <+20>
              6 0x11327ad44 <+12>: movq %rax, %rdi
              7 0x11327ad47 <+15>: jmp 0x113273cef ; perror_nocancel
              8 0x11327ad4c <+28>: retq
              9 0x11327ad4d <+23>; nop

```

You cannot look inside the source code of system libraries

Clearly, this approach is not getting you anywhere. However, there is another thing you can try — set an **Exception Breakpoint**.

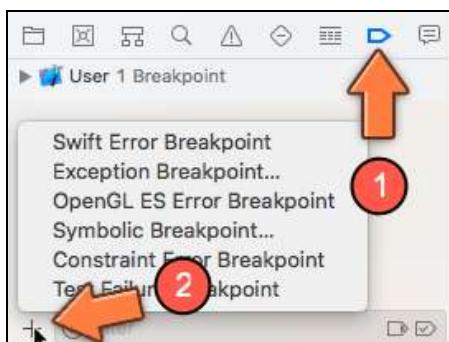
A **breakpoint** is a special marker in your code that will pause the app execution and launch the debugger.

When your app hits a breakpoint, the app will pause at that exact spot. Then you can use the debugger to step line-by-line through your code in order to run it in slow motion. That can be a handy tool if you really cannot figure out why something crashes.

You're not going to step through code in this book, but you can read more about it in the Debugging section of Apple's developer support site: developer.apple.com/support/debugging. Or, you can check the **Debug your app** topic under Xcode's **Help ▶ Xcode Help** menu option.

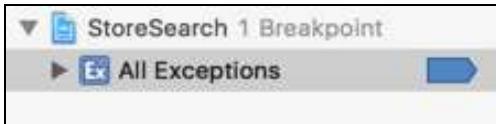
You are going to set a special breakpoint that is triggered whenever a fatal exception occurs. This will halt the program just as it is about to crash, which should give you more insight into what is going on.

- Switch to the **Breakpoint navigator** and click the + button at the bottom to add an **Exception Breakpoint**:



Adding an Exception Breakpoint

This will add a new breakpoint:



After adding the Exception Breakpoint

- Now run the app again. It will still crash, but Xcode shows a lot more info:



Xcode now halts the app at the point the exception occurs

There are many more methods in the call stack now. Let's see if we can find some clues as to what is going on.

What catches my attention is the call to something called `[UIViewController _loadViewFromNibNamed:bundle:]`. That's a pretty good hint that this error occurs when loading a nib file, or the storyboard in this case.

Using these hints and clues, and the somewhat cryptic error message that you got without the Exception Breakpoint, you can usually figure out what is making your app crash.

In this case, we've established that the app crashes when it's loading the storyboard, and the error message mentioned "searchBar2." Put two and two together and you've got your answer.

A quick peek in the source code confirms that the `searchBar2` outlet no longer exists in the view controller but the storyboard still refers to it.

- Open the storyboard and in the **Connections inspector** disconnect Search View Controller from `searchBar2` to fix the crash. That's another bug squashed!

Note: Enabling the Exception Breakpoint means that you no longer get a useful error message in the Console if the app crashes — the breakpoint stops the app just before the exception happens. If sometime later during development your app crashes on another bug, you may want to disable this breakpoint to actually see the error message. You can do that from the Breakpoint navigator by simply selecting the breakpoint and clicking on the dark blue arrow. If the arrow goes from dark blue to a pale blue, it is disabled.

To summarize:

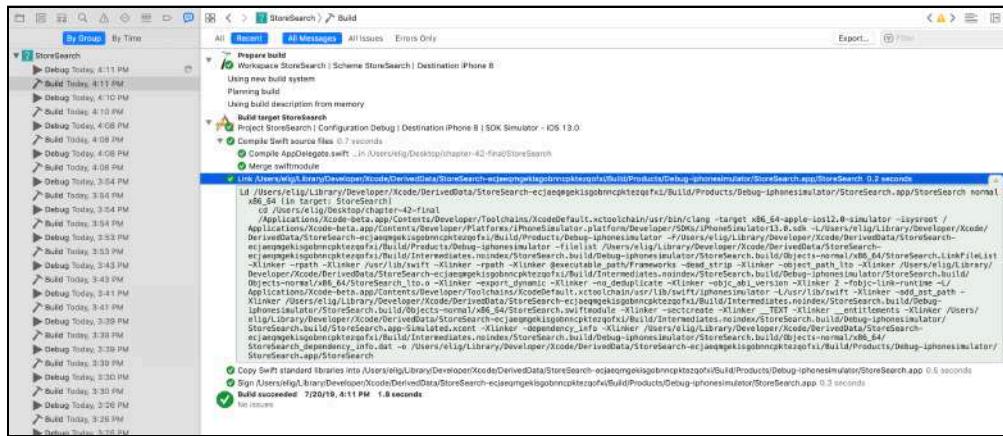
- If your app crashes while running in Xcode, the Xcode debugger will often show you an error message and where in the code the crash happened.
- If Xcode thinks the crash happened on **AppDelegate** — not very useful! — add an Exception Breakpoint to get more info.
- If the app crashes with a SIGABRT but there is no error message in the Console, disable any Exception Breakpoints you may have and make the app crash again. Alternatively, click the **Continue program execution** button from the debugger toolbar a few times. That will also show the error message... eventually.
- An EXC_BAD_ACCESS error usually means something went wrong with your memory management. An object may have been “released” one time too many or not “retained” enough. With Swift these problems are mostly a thing of the past because the compiler will usually make sure to do the right thing. However, it’s still possible to mess up if you’re talking to Objective-C code or low-level APIs.
- EXC_BREAKPOINT is not an error. The app has stopped on a breakpoint, the blue arrow points at the line where the app is paused. You set breakpoints to pause your app at specific places in the code, so you can examine the state of the app inside the debugger. The “Continue program execution” button resumes the app.

This should help you get to the bottom of most of your crashes!



The build log

If you’re wondering what Xcode actually does when it builds your app, then take a peek at the **Report navigator**. It’s the last tab in the navigator pane.



The Report navigator keeps track of your builds and debug sessions so you can look back at what happened. It even remembers the debug output of previous runs of the app.

Make sure **All Messages** is selected. To get more information about a particular log item, select the item and click the little detail icon that appears on the right. The line will expand and you’ll see exactly which commands Xcode executed and what the result was.

Should you run into some weird compilation problem, then this is the place for troubleshooting. Besides, it’s interesting to see what Xcode is up to from time to time.

You can find the project files for this chapter under **38 – Custom Table Cells** in the Source Code folder.

39

Chapter 39: Networking

Eli Ganim

Now that the preliminaries are out of the way, you can finally get to the good stuff: adding networking to the app so that you can download actual data from the iTunes Store!

The iTunes Store sells a lot of products: songs, e-books, movies, software, TV episodes... you name it. You can sign up as an affiliate and earn a commission on each sale that happens because you recommended a product — it can be even your own apps!

To make it easier for affiliates to find products, Apple made available a web service that queries the iTunes store. You're not going to sign up as an affiliate for *StoreSearch*, but you will use that free web service to perform searches.

In this chapter you will learn the following:

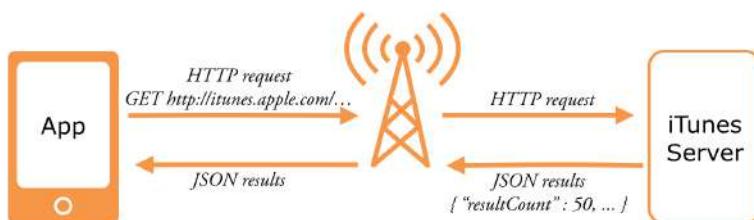
- **Query the iTunes web service:** An introduction to web services and the specifics about querying Apple's iTunes Store web service.
- **Send an HTTP request:** How to create a proper URL for querying a web service and how to send a request to the server.
- **Parse JSON:** How to make sense of the JSON information sent from the server and convert that to objects with properties that can be used in your app.
- **Sort the search results:** Explore different ways to sort the search results alphabetically so as to write the most concise and compact code.



Query the iTunes web service

So what is a *web service*? Your app — also known as the “client” — will send a message over the network to the iTunes store — the “server” — using the HTTP protocol.

Because the iPhone can be connected to different types of networks — Wi-Fi or a cellular network such as LTE, 3G, or GPRS — the app has to “speak” a variety of networking protocols to communicate with other computers on the Internet.



The HTTP requests fly over the network

Fortunately you don’t have to worry about any of that as the iPhone firmware will take care of this complicated process. All you need to know is that you’re using HTTP.

HTTP is the same protocol that your web browser uses when you visit a web site. In fact, you can play with the iTunes web service using a web browser. That’s a great way to figure out how this web service works.

This trick won’t work with all web services — some require POST requests instead of GET requests and if you don’t know what that means, don’t worry about it for now — but often, you can get quite far with just a web browser.

Open your favorite web browser — I’m using Safari — and go to the following URL:

```
http://itunes.apple.com/search?term=metallica
```

The browser will download a file. If you open the file in a text editor, it should contain something like this:

```
{
  "resultCount":50,
  "results": [
    {"wrapperType":"track", "kind":"song", "artistId":3996865,
     "collectionId":579372950, "trackId":579373079,
     "artistName":"Metallica", "collectionName":"Metallica",
     "trackName":"Enter Sandman",
     "collectionCensoredName":"Metallica", "trackCensoredName":"Enter
```

```

Sandman", "artistViewUrl": "https://itunes.apple.com/us/artist/
metallica/id3996865?uo=4", "collectionViewUrl": "https://
itunes.apple.com/us/album/enter-sandman/id579372950?
i=579373079&uo=4", "trackViewUrl": "https://itunes.apple.com/us/
album/enter-sandman/id579372950?i=579373079&uo=4",
"previewUrl": "http://a38.phobos.apple.com/us/r30/Music7/v4/bd/
fd/e4/bdfde4e4-5407-9bb0-e632-edbf079bed21/
mzaf_907706799096684396.plus.aac.p.m4a", "artworkUrl30": "http://
is1.mzstatic.com/image/thumb/Music/v4/0b/9c/
d2/0b9cd2e7-6e76-8912-0357-14780cc2616a/source/30x30bb.jpg",
"artworkUrl60": "http://is1.mzstatic.com/image/thumb/Music/v4/0b/
9c/d2/0b9cd2e7-6e76-8912-0357-14780cc2616a/source/60x60bb.jpg",
"artworkUrl100": "http://is1.mzstatic.com/image/thumb/Music/
v4/0b/9c/d2/0b9cd2e7-6e76-8912-0357-14780cc2616a/source/
100x100bb.jpg", "collectionPrice": 9.99, "trackPrice": 1.29,
"releaseDate": "1991-07-29T07:00:00Z",
"collectionExplicitness": "notExplicit",
"trackExplicitness": "notExplicit", "discCount": 1,
"discNumber": 1, "trackCount": 12, "trackNumber": 1,
"trackTimeMillis": 331560, "country": "USA", "currency": "USD",
"primaryGenreName": "Metal", "isStreamable": true},
...

```

Those are the search results that the iTunes web service gives you. The data is in a format named **JSON**, which stands for **JavaScript Object Notation**.

JSON is commonly used to send structured data back-and-forth between servers and clients, i.e. apps. Another data format that you may have heard of is XML, but that's being fast replaced by JSON.

There are a variety of tools that you can use to make the JSON output more readable for mere humans. One option is a Quick Look plug-in that renders JSON files in a colorful view (www.sagtau.com/quicklookjson.html).

You do need to save the output from the server to a file with a **.json** extension first, and then open it from Finder by pressing the space bar:

```

{
  "resultCount": 50,
  "results": [
    {
      "wrapperType": "track",
      "kind": "song",
      "artistId": 3996865,
      "collectionId": 579372950,
      "trackId": 579373079,
      "artistName": "Metallica",
      "collectionName": "Metallica",
      "trackName": "Enter Sandman",
      "collectionCensoredName": "Metallica",
      "trackCensoredName": "Enter Sandman",
      "artistViewUrl": "https://itunes.apple.com/us/artist/metallica/id3996865?uo=4",
      ...
    }
  ]
}

```

A more readable version of the output from the web service

That makes a lot more sense.

Note: You can find extensions for Safari (and most other browsers) that can prettify JSON directly inside the browser. github.com/rfletcher/safari-json-formatter is a good one.

There are also dedicated tools on the Mac App Store, for example Visual JSON, that let you directly perform the request on the server and show the output in a structured and readable format.

A great online tool is codebeautify.org/jsonviewer.

Browse through the JSON text for a bit. You'll see that the server gave back a list of items, some of which are songs; others are audiobooks, or music videos.

Each item has a bunch of data associated with it, such as an artist name — “Metallica,” which is what you searched for —, a track name, a genre, a price, a release date, and so on.

You'll store some of these fields in the `SearchResult` class so you can display them on the screen.

The results you get from the iTunes store might be different from mine. By default, the search returns at most 50 items and since the store has quite a bit more than fifty entries that match “metallica,” each time you do the search you may get back a different set of 50 results.

Also notice that some of these fields, such as `artistViewUrl` and `artworkUrl100` and `previewUrl` are links/URLs. Go ahead and copy-paste these URLs in your browser and see what happens.

The `artistViewUrl` will open an iTunes Preview page for the artist, the `artworkUrl100` loads a thumbnail image, and the `previewUrl` opens a 30-second audio preview.

This is how the server tells you about additional resources. The images and so on are not embedded directly into the search results, but you're given a URL that allows you to download each item separately. Try some of the other URLs from the JSON data and see what they do!



Back to the original HTTP request. You made the web browser go to the following URL:

```
http://itunes.apple.com/search?term=the search term
```

You can add other parameters as well to make the search more specific. For example:

```
http://itunes.apple.com/search?term=metallica&entity=song
```

Now the results won't contain any music videos or podcasts, only songs.

If the search term has a space in it you should replace it with a + sign, as in:

```
http://itunes.apple.com/search?term=pokemon+go&entity=software
```

This searches for all apps that have something to do with Pokemon Go — you may have heard of some of them.

The fields in the JSON results for this particular query are slightly different than before. There is no `previewUrl` but there are several screenshot URLs per entry. Different kinds of products — songs, movies, software — return different types of data.

That's all there is to it. You construct a URL to `itunes.apple.com` with the search parameters and then use that URL to make an HTTP request. The server will send some JSON gobbledegook back to the app and you'll have to somehow turn that into `SearchResult` objects and put them in the table view. Let's get on it!

Synchronous networking = bad

Before you begin you should know that there is a bad way to do networking in your apps and a good way.

The bad way is to perform the HTTP requests on your app's **main thread** — it is simple to program, but it will block the user interface and make your app unresponsive while the networking is taking place. Because it blocks the rest of the app, this is called synchronous networking.

Unfortunately, many programmers insist on doing networking the wrong way in their apps, which makes for apps that are slow and prone to crashing.

You'll start with the easy-but-bad way, just to see how *not* to do this. It's important that you realize the consequences of synchronous networking, so you will avoid it in your own apps.



After you're convinced of the evilness of this approach, you'll see how to do it the right way — it only requires a small modification to the code, but may require a big change in how you think about these problems.

Asynchronous networking — the right kind, with an “a” — makes your apps much more responsive, but also brings with it additional complexity that you need to deal with.

Sending an HTTP request

In order to query the iTunes Store web service, the very first thing you must do is send an HTTP request to the iTunes server. This involves several steps such as creating a URL with the correct search parameters, sending the request to the server, getting a response back etc. You'll take these step-by-step.

Creating the URL for the request

- Add a new method to **SearchViewController.swift**:

```
// MARK:- Helper Methods
func iTunesURL(searchText: String) -> URL {
    let urlString = String(format:
        "https://itunes.apple.com/search?term=%@", searchText)
    let url = URL(string: urlString)
    return url!
}
```

This first builds a URL string by placing the search text behind the “term=” parameter, and then turns this string into a URL object.

Because `URL(string:)` is a failable initializer, it returns an optional. You force unwrap that using `url!` to return an actual URL object.

HTTPS vs. HTTP Previously you used `http://` but here you're using `https://`. The difference is that HTTPS is the secure, encrypted version of HTTP. It protects your users from eavesdropping. The underlying protocol is the same, but any bytes that you're sending or receiving are encrypted before they go out on the network. As of iOS 9, Apple recommends that apps should always use HTTPS. In fact, even if you specify an unprotected `http://` URL, iOS will still try to connect using HTTPS. If the server isn't configured to use HTTPS, then the network connection will fail. You can ask to be exempt from

this behavior in your Info.plist file, but that is generally not recommended. Later on, you'll learn how to do this because the artwork images are hosted on a server that does not support HTTPS.

- Change searchBarSearchButtonClicked(_:) to:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()

        hasSearched = true
        searchResults = []

        let url = iTunesURL(searchText: searchBar.text!)
        print("URL: '\(url)'")

        tableView.reloadData()
    }
}
```

You've removed the code that created fake SearchResult items, and instead, call the new iTunesURL(searchText:) method. For testing purposes, you log the URL object that this method returns.

This logic sits inside an `if` statement so that none of this happens unless the user actually typed text into the search bar — it doesn't make much sense to search the iTunes store for “nothing.”

Note: Don't get confused by all the exclamation points in the line,

```
if !searchBar.text!.isEmpty
```

The first one is the “logical not” operator because you want to go inside the `if` statement only if the text is not empty. The second exclamation point is for force unwrapping the value of `searchBar.text`, which is an optional — it will never actually be `nil`, so it being an optional is a bit silly, but whaddya gonna do?

- Run the app and type in some search text that is a single word, for example “metallica,” or one of your other favorite metal bands, and press the Search button.



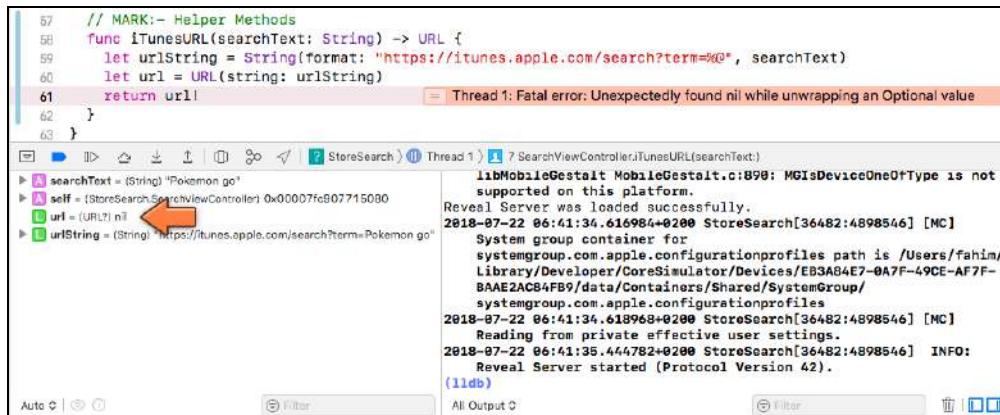
Xcode should now show this in its Debug pane:

```
URL: 'https://itunes.apple.com/search?term=metallica'
```

That looks good.

- Now type in a search term with one or more spaces, like “pokemon go,” into the search box.

Whoops, the app crashes!



The crash after searching for pokemon-go

Look at the left-hand pane, the **Variables view**, of the Xcode debugger and you’ll see that the value of the `url` constant is `nil` — this may also show up as `0x0000...` followed by a whole bunch of zeros.

The app apparently did not create a valid URL object. But why?

A space is not a valid character in a URL. Many other characters aren’t valid either — such as the < or > signs — and therefore must be **escaped**. Another term for this is **URL encoding**.

A space, for example, can be encoded as the + sign — you did that earlier when you typed the URL into the web browser — or as the character sequence %20.

- Fortunately, `String` can do this encoding already. So, you only have to add one extra statement to the app to make this work:

```
func iTunesURL(searchText: String) -> URL {
    let encodedText = searchText.addingPercentEncoding(
        withAllowedCharacters: CharacterSet.urlQueryAllowed)!
    let urlString = String(format:
```

```
"https://itunes.apple.com/search?term=%@", encodedText)
let url = URL(string: urlString)
return url!
}
```

This calls the `addingPercentEncoding(withAllowedCharacters:)` method to create a new string where all the special characters are escaped, and you use that string for the search term.

UTF-8 string encoding

This new string treats the special characters as being “UTF-8 encoded.” It’s important to know what that means because you’ll run into this UTF-8 thing every once in a while when dealing with text.

There are many different ways to encode text. You’ve probably heard of ASCII and Unicode, the two most common encodings.

UTF-8 is a version of Unicode that is very efficient for storing regular text, but less so for special symbols or non-Western alphabets. Still, it’s the most popular way to deal with Unicode text today.

Normally, you don’t have to worry about how your strings are encoded. But when sending requests to a web service you need to transmit the text with the proper encoding. Tip: When in doubt, use UTF-8, it will almost always work.

- Run the app and search for “pokemon go” again. This time a valid URL object can be created, and it looks like this:

```
URL: 'https://itunes.apple.com/search?term=pokemon%20go'
```

The space has been turned into the character sequence `%20`. The `%` indicates an escaped character and `20` is the UTF-8 value for a space. Also try searching for terms with other special characters, such as `#` and `*` or even Emoji, and see what happens.



Performing the search request

Now that you have a valid URL object, you can do some actual networking!

- Add a new method to **SearchViewController.swift**:

```
func performStoreRequest(with url: URL) -> String? {
    do {
        return try String(contentsOf: url, encoding: .utf8)
    } catch {
        print("Download Error: \(error.localizedDescription)")
        return nil
    }
}
```

The meat of this method is the call to `String(contentsOf:encoding:)` which returns a new string object with the data it receives from the server pointed to by the URL.

Note that you're telling the app to interpret the data as UTF-8 text. Should the server send back the text in a different encoding, then it will look like a garbled mess to your app.

It's important that the sending and receiving sides agree on the encoding they are using!

Because things can go wrong — for example, the network may be down and the server cannot be reached — you enclose this in a do-try-catch block. If there is a problem, the code jumps to the catch branch and the `error` variable will contain more details about the error. If this happens, you print out a user-understandable form of the error and return `nil` to signal that the request failed.

- Add the following lines to `searchBarSearchButtonClicked(_:)`, after the `print()` line:

```
if let jsonString = performStoreRequest(with: url) {
    print("Received JSON string '\(jsonString)'")
}
```

This invokes `performStoreRequest(with:)` with the URL object as a parameter and returns the JSON data that is received from the server. If everything goes according to plan, this method returns a new string containing the JSON data that you're after. Let's try it out!

- Run the app and search for your favorite band. After a second or so, a whole bunch of data will be dumped to the Xcode Console:

```
URL: 'http://itunes.apple.com/search?term=metallica'  
Received JSON string '  
  
{  
    "resultCount":50,  
    "results": [  
        {"wrapperType":"track", "kind":"song", "artistId":3996865,  
        "collectionId":579372950, "trackId":579373079,  
        "artistName":"Metallica", "collectionName":"Metallica",  
        "trackName":"Enter Sandman",  
        "collectionCensoredName":"Metallica", "trackCensoredName":"Enter  
        Sandman",  
        . . . and so on . . .
```

Congratulations, your app has successfully talked to a web service!

This prints the same stuff that you saw in the web browser earlier. Right now it's all contained in a single `String` object, which isn't really useful for our purposes, but you'll convert it to a more useful format in a minute.

Of course, it's possible that you received an error. In that case, the output should be something like this:

```
URL: 'https://itunes.apple.com/search?term=Metallica'  
HTTP load failed (error code: -1009 [1:50]) for Task  
<F5199AB7-5011-42FB-91B5-656244861482>. <0>  
NSURLSession finished with error - code -1009  
Download Error: The file "search" couldn't be opened.
```

You'll add better error handling to the app later, but if you get such an error at this point, then make sure your computer — or your iPhone in case you're running the app on a device and not in the Simulator — is connected to the Internet. Also try the URL directly in your web browser and see if that works.

Parsing JSON

Now that you have managed to download a chunk of JSON data from the server, what do you do with it?



JSON is a *structured* data format. It typically consists of arrays and dictionaries that contain other arrays and dictionaries, as well as regular data such as strings and numbers.

An overview of the JSON data

The JSON from the iTunes store roughly looks like this:

```
{  
    "resultCount": 50,  
    "results": [ . . . a bunch of other stuff . . . ]  
}
```

The { } brackets surround a dictionary. This particular dictionary has two keys: `resultCount` and `results`. The first one, `resultCount`, has a numeric value. This is the number of items that matched your search query. By default the limit is a maximum of 50 items, but as you will see later, you can increase this upper limit.

The `results` key contains an array, which is indicated by the [] brackets. Inside that array are more dictionaries, each of which describes a single product from the store. You can tell these things are dictionaries because they have the { } brackets again.

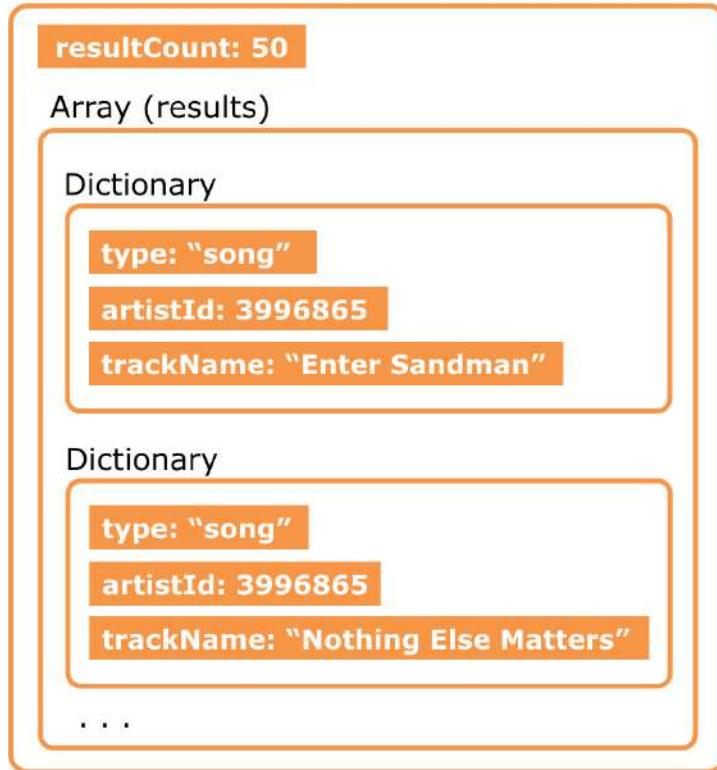
Here are two of these items from the array:

```
{  
    "wrapperType": "track",  
    "kind": "song",  
    "artistId": 3996865,  
    "artistName": "Metallica",  
    "trackName": "Enter Sandman",  
    . . . and so on . . .  
},  
{  
    "wrapperType": "track",  
    "kind": "song",  
    "artistId": 3996865,  
    "artistName": "Metallica",  
    "trackName": "Nothing Else Matters",  
    . . . and so on . . .  
},
```

Each product is represented by a dictionary made up of several keys. The values of the `kind` and `wrapperType` keys determine what sort of product this is: a song, a music video, an audiobook, and so on. The other keys describe the artist and the song itself.



Dictionary



The structure of the JSON data

To summarize, the JSON data represents a dictionary and inside that dictionary is an array of more dictionaries. Each of the dictionaries from the array represents one search result.

Currently, all of this sits in a `String`, which isn't very handy, but using a **JSON parser** you can turn this data into Swift `Dictionary` and `Array` objects.

JSON or XML?

JSON is not the only structured data format out there. XML, which stands for EXtensible Markup Language, is a slightly more formal standard. Both formats serve the same purpose, but they look a bit different.



If the iTunes store returned its results as XML, the output would look more like this:

```
<?xml version="1.0" encoding="utf-8"?>
<iTunesSearch>
<resultCount>5</resultCount>
<results>
  <song>
    <artistName>Metallica</artistName>
    <trackName>Enter Sandman</trackName>
  </song>
  <song>
    <artistName>Metallica</artistName>
    <trackName>Nothing Else Matters</trackName>
  </song>
  . . . and so on . . .
</results>
</iTunesSearch>
```

These days, most developers prefer JSON because it's simpler than XML and easier to parse. But it's certainly possible that if you want your app to talk to a particular web service, you'll be expected to deal with XML data.

Preparing to parse JSON data

In the past, if you wanted to parse JSON, it used to be necessary to include a third-party framework into your apps, or to manually walk through the data structure using the built-in iOS JSON parser. But with Swift 4, there's a new way to do things — your old pal Codable.

Remember how you used a `PropertyListDecoder` to decode plist data that supported the `Codable` protocol for reading — and saving — data in *Checklists*? Well, property lists aren't the only format supported by `Codable` out of the box — JSON is supported too!

All you need to do in order to allow your app to read JSON data directly into the relevant data structures is to set them up to conform to `Codable`!

You might ask how does `Codable` know how an arbitrary data structure from the Internet is set up in order to correctly extract the right bits of data. It's all in how you set your data structures up. You'll understand as you proceed to parse the data you received from the iTunes server.

The trick to using `Codable` to parse JSON data is to set up your classes — or structs — to reflect the structure of the data that you'll parse.



As you noticed above, there are two parts to the JSON response received from the iTunes server:

1. The response wrapper which contains the number of results and an array of results.
2. The array itself which is made up of individual search result items.

We need to model both of the above in order to parse the JSON data correctly. We've already made some headway in terms of modeling the search results by way of the `SearchResult` object, but we need to do some modifications in order to get the object ready for JSON parsing.

But first, let's add a new data model for the results wrapper.

► Open `SearchResult.swift` and replace its contents with the following:

```
class ResultArray: Codable {
    var resultCount = 0
    var results = [SearchResult]()
}

class SearchResult: Codable {
    var artistName: String? = ""
    var trackName: String? = ""

    var name: String {
        return trackName ?? ""
    }
}
```

There are a few changes here:

1. The `ResultArray` class models the response wrapper by containing a `results` count and an array of `SearchResult` objects. Note that this class supports the `Codable` protocol.

If you are wondering why this class is within the same file as `SearchResult`, it is simply for the sake of expediency. This class is not used anywhere else except as a temporary holder during the JSON parsing process, so it makes sense to put it in the same file as `SearchResult`, which is the actual class you'll be using. But if you prefer, you can put this class in a separate Swift file by itself — it doesn't make any difference to the app functionality.

2. The `SearchResult` class now supports the `Codable` protocol, too.
3. It also has a new property named `trackName` and the `artistName` property has been changed to an optional one — the optional properties are to make `Codable`'s

work easier since `Codable` expects non-optional values to be always present in the JSON data. Unfortunately, since the response from the iTunes server might not always have these properties and you have to allow for that.

4. The existing property for `name` has been converted to a computed property which returns the value of the `trackName` property, or an empty string if `trackName` is `nil`.

The reason for changes #3 and #4 might not be obvious immediately. Take a look at the response data you received from the server. Did you notice the "kind" key?

The search results from iTunes can be for multiple types of items – songs, videos, movies, tv shows, books etc. That key indicates the type of item the search result is for.

And depending on the item type, you might want to vary how you display an item name. For example, you might not always want to use the "trackName" key as the item name – in fact, as we mention above, "trackName" might not even be there in the returned data.

The computed `name` property is simply preparation for the future in case you want to display different names depending on the result type.

Also, notice that now all the property names in the class match actual keys in the JSON data – you can parse JSON even without the property names matching the key names, but that's a bit more complicated. So let's take the easy route here.

Remember, baby steps... And that's all you need in order to prepare for JSON parsing. Onwards!

Parsing the JSON data

You will be using the `JSONDecoder` class, appropriately enough, to parse JSON data. Only trouble is, `JSONDecoder` needs its input to be a `Data` object. You currently have the JSON response from the server as a `String`. You can convert the `String` to `Data` pretty easily, but it would be better to get the response from the server as `Data` in the first place – you got the response from the server as `String` initially only to ensure that the response was correct.

- Switch to `SearchViewController.swift` and modify `performStoreRequest(with:)` as follows:

```
func performStoreRequest(with url: URL) -> Data? { // Change to
    Data?
    do {
```

```
        return try Data(contentsOf:url)    // Change this line
    } catch {
        ...
    }
}
```

You simply change the request method to fetch the response from the server as `Data` instead of a `String` — the method now returns the value as an optional `Data` value instead of an optional `String` value.

► Add the following method to `SearchViewController.swift`:

```
func parse(data: Data) -> [SearchResult] {
    do {
        let decoder = JSONDecoder()
        let result = try decoder.decode(ResultArray.self, from:data)
        return result.results
    } catch {
        print("JSON Error: \(error)")
        return []
    }
}
```

You use a `JSONDecoder` object to convert the response data from the server to a temporary `ResultArray` object from which you extract the `results` property. Or at least, you *hope* you can convert the data without any issues...

Assumptions cause trouble

When you write apps that talk to other computers on the Internet, one thing to keep in mind is that your conversational partners may not always say the things you expect them to say.

There could be an error on the server and instead of valid JSON data, it may send back some error message. In that case, `JSONDecoder` will not be able to parse the data and the app will return an empty array from `parse(data:)`.

Another thing that could happen is that the owner of the server changes the format of the data they send back. Usually, this is done in a new version of the web service that is accessible via a different URL. Or, they might require you to send along a “version” parameter. But not everyone is careful like that, and by changing what the server does, they may break apps that depend on the data coming back in a specific format.

In the case of the iTunes store web service, the top-level object *should* be a dictionary with two keys — one for the count, the other for the array of results — but you can’t



control what happens on the server. If for some reason the server programmers decide to put [] brackets around the JSON data, then the top-level object will no longer be a Dictionary but an Array. This in turn will cause `JSONDecoder` to fail parsing the data since it is no longer in the expected format.

Being paranoid about these kinds of things and showing an error message in the unlikely event this happens is a lot better than your application suddenly crashing when something changes on a server that is outside of your control.

Just to be sure, you're using the do-try-catch block to check that the JSON parsing goes through fine. Should the conversion fail, then the app doesn't burst into flames but simply returns an empty results array.

It's good to add checks like these to the app to make sure you get back what you expect. If you don't own the servers you're talking to, it's best to program defensively.

► Modify `searchBarSearchButtonClicked(_:)` as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        .
        .
        .
        print("URL: '\(url)'")
        if let data = performStoreRequest(with: url) { // Modified
            let results = parse(data: data) // New line
            print("Got results: \(results)") // New line
        }
        tableView.reloadData()
    }
}
```

You simply change the constant for the result from the call to `performStoreRequest(with:)` from `jsonString` to `data`, call the new `parse(data:)` method, and print the return value.

► Run the app and search for something. The Xcode Console now prints the following:

```
URL: 'https://itunes.apple.com/search?term=Metallica'
Got results: [StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
.
.
.
]
```

Hmm ... that certainly *looks* like an array of 50 items, but it doesn't really tell you anything much about the actual data — just that the array consists of `SearchResult` objects. That's not much good to you, is it?

Printing object contents

- Modify the `SearchResult` class in `SearchResult.swift` to conform to the `CustomStringConvertible` protocol:

```
class SearchResult: Codable, CustomStringConvertible {
```

The `CustomStringConvertible` protocol allows an object to have a custom string representation. Or, to put it another way, the protocol allows objects to have a custom string describing the object, or its contents.

So, how does the protocol provide this string description? That is done via the protocol's `description` property.

- Add the following code to the `SearchResult` class:

```
var description: String {
    return "Name: \(name), Artist Name: \(artistName ?? "None")"
}
```

The above is your implementation of the `description` property to conform to the `CustomStringConvertible`. For your `SearchResult` class, the description consists of the values of the `name` and `artistName` properties — but since `artistName` is an optional value, you have to account for when it might be `nil` and output "None" when that happens.

Notice the `??` operator in the above code — it's called the *nil-coalescing operator* and you probably remember it from previous chapters. The nil-coalescing operator unwraps the variable to the left of the operator if it has a value, if not, it returns the value to the right of the operator as the default value.

- Run the app again and search for something. The Xcode Console should now print something like the following:

```
URL: 'https://itunes.apple.com/search?term=Metallica'
Got results: [Name: Enter Sandman, Artist Name: Metallica, Name:
Nothing Else Matters, Artist Name: Metallica, Name: The
Unforgiven, Artist Name: Metallica, Name: One, Artist Name:
Metallica, Name: Wherever I May Roam, Artist Name: Metallica,
...]
```

Yep, that looks more like it! You have converted a bunch of JSON that didn't make a lot of sense, into actual objects that you can use.

Error handling

Let's add an alert to handle potential errors. It's inevitable that something goes wrong somewhere and it's best to be prepared.

- Add the following method to **SearchViewController.swift**:

```
func showNetworkError() {  
    let alert = UIAlertController(title: "Whoops...",  
        message: "There was an error accessing the iTunes Store." +  
            " Please try again.", preferredStyle: .alert)  
  
    let action = UIAlertAction(title: "OK", style: .default,  
        handler: nil)  
    alert.addAction(action)  
    present(alert, animated: true, completion: nil)  
}
```

Nothing you haven't seen before; it simply presents an alert controller with an error message.

Note: The `message` variable is split into two separate strings and concatenated, or added together, using the plus (+) operator just so that the string would display nicely for this book. You can feel free to type out the whole string as a single string instead.

- Add the following line to `performStoreRequest(with:)` just before the `return nil`:

```
showNetworkError()
```

Simply put, if something goes wrong with the request to the iTunes store, you call `showNetworkError()` to show an alert box.

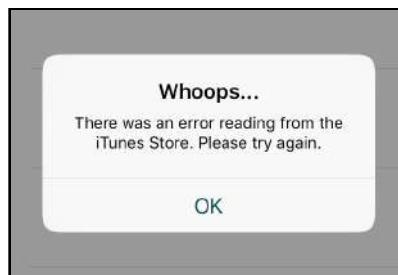
If you did everything correctly up to this point, then the web service should always have worked. Still it's a good idea to test a few error situations, just to make sure the error handling is working for those unlucky users with bad network connections.

- Try this: In `iTunesURL(searchText:)` method, temporarily change the "itunes.apple.com" part of the URL to "NOMOREitunes.apple.com."

You should now get an error alert when you try a search because no such server exists at that address. This simulates the iTunes server being down. Don't forget to change the URL back when you're done testing.

Tip: To simulate no network connection you can pull the network cable and/or disable Wi-Fi on your Mac, or run the app on your device in Airplane Mode.

It should be obvious that when you're doing networking, things can — and will! — go wrong, often in unexpected ways. So, it's always good to be prepared for surprises.



The app shows an alert when there is a network error

Working with the JSON results

So far you've managed to send a request to the iTunes web service and you parsed the JSON data into an array of `SearchResult` objects. However, we are not quite done.

The iTunes Store sells different kinds of products — songs, e-books, software, movies, and so on — and each of these has its own structure in the JSON data. A software product will have screenshots but a movie will have a video preview. The app will have to handle these different kinds of data.

You're not going to support everything the iTunes store has to offer, only these items:

- Songs, music videos, movies, TV shows, podcasts
- Audio books
- Software (apps)
- E-books

The reason to split them up like this is because that's how the iTunes store does it. Songs and music videos, for example, share the same set of fields, but audiobooks

and software have different data structures. The JSON data makes this distinction using the `kind` field.

Let's modify our data model to load the value for the above key.

- Add the following property to `SearchResult` (**SearchResult.swift**):

```
var kind: String? = ""
```

You might think that the "kind" property would always be there in the iTunes data and so it need not be an optional. Unfortunately that's not the case, so you'll need to go with an optional value there.

- Also modify the `return` line for `description` to:

```
return "Kind: \(kind ?? "None"), Name: \(name), Artist Name: \
(artistName ?? "None")\n"
```

That makes sense given that `kind` is optional, right?

- Run the app and do a search. Look at the Xcode output.

When you do this, Xcode shows three different types of products, with the majority of the results being songs. What you see may vary, depending on what you search for.

```
URL: 'https://itunes.apple.com/search?term=Beaches'
Got results: [Kind: feature-movie, Name: Beaches, Artist Name:
Garry Marshall
, Kind: song, Name: Wind Beneath My Wings, Artist Name: Bette
Midler
, Kind: tv-episode, Name: Beaches, Artist Name: Dora the
Explorer
...

```

Now, let's add some new properties to the `SearchResult` object.

Always check the documentation

If you were wondering how you're supposed to know how to interpret the data from the iTunes web service, or even how to set up the URLs to use the service in the first place, then you should realize there is no way you can be expected to use a web service if there is no documentation.

Fortunately, for the iTunes store web service, there is some good documentation here:

affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api

Just reading the docs is often not enough though. You have to play with the web service a bit to know what you can and cannot do.

There are some things that the *StoreSearch* app needs to do with the search results that were not clear from reading the documentation. So, first read the docs and then play with it. That goes for any API, really, whether it's something from the iOS SDK or a web service.

Loading more properties

The current `SearchResult` class only has a few properties. As you've seen, the iTunes store returns a lot more information than that, so you'll need to add a few new properties.

► Add the following properties to `SearchResult.swift`:

```
var trackPrice: Double? = 0.0
var currency = ""
var artworkUrl60 = ""
var artworkUrl100 = ""
var trackViewUrl: String? = ""
var primaryGenreName = ""
```

You're not including *everything* that the iTunes store returns, only the fields that are interesting to this app. Also, note that you've named the properties to match the keys in the JSON data exactly and that only some have been marked as optional.

Note: The optionality of the properties was based on my own results. It is possible that with the above code, you still find that the app barfs all over the place with an error like this:

```
URL: 'https://itunes.apple.com/search?term=Macky'
JSON Error: keyNotFound(CodingKeys(stringValue: "trackViewUrl",
intValue: nil), Swift.DecodingError.Context(codingPath:
[CodingKeys(stringValue: "results", intValue: nil),
JSONKey(stringValue: "Index 1", intValue: 1)],
debugDescription: "No value associated with key")
```

```
CodingKeys(stringValue: \"trackViewUrl\", intValue: nil)
(\\"trackViewUrl\").", underlyingError: nil))
```

If this happens to you, look at the error message to figure out the property in `SearchResult` which is missing and then mark it as optional — problem solved!

`SearchResult` stores the item's price and the currency — US dollar, Euro, British Pounds, etc. It also stores two artwork URLs, one for a 60×60 pixel image and the other for a 100×100 pixel image, a link to the product's page on the iTunes store, and the genre of the item.

Provided the class supports `Codable`, with just the simple addition of new properties — as long as they are named the same as the JSON keys and have the right optionality — you are now able to load these new values into your class.

But what if you don't want to use the not-quite-user-friendly names from the JSON data such as `artworkUrl60` or `artworkUrl100` but instead want to use more descriptive names such as `artworkSmall` and `artworkLarge`?

Never fear, `Codable` has support for that, too.

But before we get to that, you should run your app once to make sure that the above code changes didn't break anything. So, run your app, make a search, and verify that you still get output in the Xcode Console indicating that the search was successful.

All working fine? Great! Let's move on to naming the `SearchResults` properties to be as you want them and not as the JSON data sets them ...

Supporting better property names

► Replace the following lines of code in `SearchResult.swift`:

```
var artworkUrl60 = ""
var artworkUrl100 = ""
var trackViewUrl: String? = ""
var primaryGenreName = ""
```

With this:

```
var imageSmall = ""
var imageLarge = ""
var storeURL: String? = ""
```

```
var genre = ""

enum CodingKeys: String, CodingKey {
    case imageSmall = "artworkUrl60"
    case imageLarge = "artworkUrl100"
    case storeURL = "trackViewUrl"
    case genre = "primaryGenreName"
    case kind, artistName, trackName
    case trackPrice, currency
}
```

As you'll notice, you've changed the property names to be more descriptive, but what does the enum do?

As you've seen previously, an enum (or enumeration), is a way to have a list of values and names for those values. Here, you use the `Codable` protocol to let the `Codable` protocol know how you want the `SearchResult` properties matched to the JSON data.

Do note that if you do use the `CodingKeys` enumeration, it has to provide a case for all your properties in the class — the ones which map to a JSON key with the same name are the last two cases in the enum, you'll notice that they don't have a value specified.

That's all there is to it! Run your app again (and maybe change the `description` property to return one of the new values to test they display correctly) and verify that the code still works with the new properties.

Using the results

With these latest changes, `searchBarSearchButtonClicked(_:)` retrieves an array of `SearchResult` objects populated with useful information, but you're not doing anything with that array yet.

- Switch to `SearchViewController.swift` and in `searchBarSearchButtonClicked(_:)`, replace the following lines:

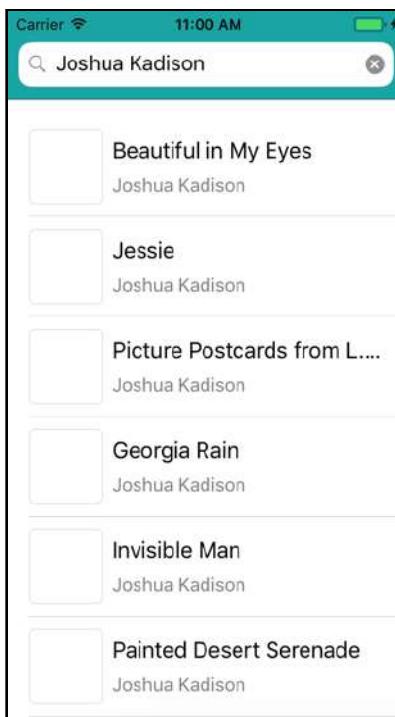
```
let results = parse(data: data)
print("Got results: \(results)")
```

With:

```
searchResults = parse(data: data)
```

Instead of placing the results in a local variable and printing them out, you now place the returned array into the `searchResults` instance variable so that the table view can show the actual search result objects.

- Run the app and search for your favorite musician. After a second or so, you should see a whole bunch of results appear in the table. Cool!



The results from the search now show up in the table

Differing data structures

Remember that some items, such as audiobooks have different data structures? Let's talk about that a bit more in detail...

The biggest differences currently between the other item types and audiobooks is that audiobooks do not have certain JSON keys that are present for other items. Here's a breakdown:

1. **kind:** This value is not present at all.
2. **trackName:** Instead of "trackName", you get "collectionName".

3. **trackviewUrl**: Instead of this value, you have "collectionViewUrl" — which provides the iTunes link to the item.
4. **trackPrice**: Instead of "trackPrice", you get "collectionPrice".

Interestingly enough, you'll notice that in `SearchResult` these are all properties that we had marked as optional. You grok now why we had to mark them optional, right? If your search results included an audiobook item, those properties would not have been there and so `Codable` would have had a fit!

Additionally, there are a few other JSON differences for a couple of item types:

1. Software and e-book items do not have "trackPrice" key, instead they have a "price" key.
2. E-books don't have a "primaryGenreName" key — they have an array of genres.

So how can you fix things so that the `JSONDecoder` can correctly decode the JSON data from the iTunes Store server no matter the type of item? How do you handle the situations where the same property — for example, "trackPrice" — can be present as a different property — like "collectionPrice" or "price" — depending on the type of item?

Remember how you added a computed variable called `name` which returns the `trackName`? This is where that comes into play ... If you add another variable to store `collectionName` — the name of the item when it is an audiobook — then you can return the correct value from `name` depending on the case. You can do something similar for the store URL and price as well.

Let's make the necessary changes.

- Remove the `storeURL` property from `SearchResult` — you'll add two separate optional properties for the audiobook and non-audiobook types. Also remove the `storeURL` case from `CodingKeys`.
- Remove the `genre` property from `SearchResult` — you'll add two separate optional properties for the e-book and non-e-book types. Also remove the `genre` case from `CodingKeys`.
- Add new optional properties for the variant keys present in the special items mentioned above:

```
var trackViewUrl: String?  
var collectionName: String?  
var collectionViewUrl: String?  
var collectionPrice: Double?
```



```
var itemPrice: Double?  
var itemGenre: String?  
var bookGenre: [String]?
```

- Replace the name computed property with the following:

```
var name: String {  
    return trackName ?? collectionName ?? ""  
}
```

The change is simple enough, except for the *chaining* of the nil-coalescing operator. You check to see if `trackName` is `nil` — if not, you return the unwrapped value of `trackName`. If `trackName` is `nil`, you move on to `collectionName` and do the same check. If both values are `nil`, you return an empty string.

- Add the following three new computed properties:

```
var storeURL: String {  
    return trackViewUrl ?? collectionViewUrl ?? ""  
}  
  
var price: Double {  
    return trackPrice ?? collectionPrice ?? itemPrice ?? 0.0  
}  
  
var genre: String {  
    if let genre = itemGenre {  
        return genre  
    } else if let genres = bookGenre {  
        return genres.joined(separator: ", ")  
    }  
    return ""  
}
```

The first two computed properties work similar to how the `name` computed property works. So nothing new there. The `genre` property simply returns the genre for items which are not e-books. For e-books, the method combines all the genre values in the array separated by commas and then returns the combined string.

All that remains is to add all the new properties to the `CodingKeys` enumeration — if you don't, some of the values might not be populated correctly during JSON decoding. Once you're done, `CodingKeys` should look like this:

```
enum CodingKeys: String, CodingKey {  
    case imageSmall = "artworkUrl60"  
    case imageLarge = "artworkUrl100"  
    case itemGenre = "primaryGenreName"  
    case bookGenre = "genres"
```

```
    case itemPrice = "price"
    case kind, artistName, currency
    case trackName, trackPrice, trackViewUrl
    case collectionName, collectionViewUrl, collectionPrice
}
```

- Run the app again, and search for something like "Stephen King" to be sure to get some results which include audiobooks for the master of horror! In case you wonder why that specific search term, we are looking for audiobooks specifically because that is one of the item types with variations in the data structures...

Showing the product type

The search results may include podcasts, songs, or other related products. It would be useful to make the table view display what type of product it is showing.

- Still in **SearchResult.swift**, add the following computed properties:

```
var type: String {
    return kind ?? "audiobook"
}

var artist: String {
    return artistName ?? ""
}
```

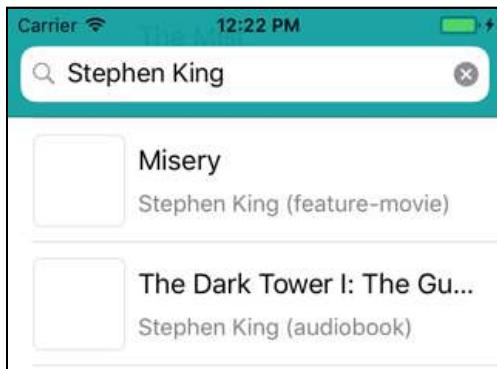
Remember that `kind` could be `nil` if the item type is an audiobook and that we've marked `artistName` as an optional. You hedge against that with these new computed properties.

- Open **SearchViewController.swift** and in `tableView(_:cellForRowAt:)`, change the line that sets `cell.artistNameLabel` to the following:

```
if searchResult.artist.isEmpty {
    cell.artistNameLabel.text = "Unknown"
} else {
    cell.artistNameLabel.text = String(format: "%@ (%@)", searchResult.artist, searchResult.type)
}
```

The first change is that you now check that the `SearchResult`'s `artist` is not empty. You might notice that sometimes a search result doesn't include an artist name. In that case you make the cell say "Unknown."

You also add the value of the new type property to the artist name label, which should tell the user what kind of product they’re looking at:



They're not books...

There is one problem with this. The value of kind comes straight from the server and it is more of an internal name than something you’d want to show directly to the user.

What if you want it to say “Movie” instead, or maybe you want to translate the app to another language — something you’ll do later for *StoreSearch*. It’s better to convert this internal identifier, “feature-movie,” into the text that you want to show to the user, “Movie.”

► Replace the type computed property in **SearchResult.swift** with this one:

```
var type:String {  
    let kind = self.kind ?? "audiobook"  
    switch kind {  
        case "album": return "Album"  
        case "audiobook": return "Audio Book"  
        case "book": return "Book"  
        case "ebook": return "E-Book"  
        case "feature-movie": return "Movie"  
        case "music-video": return "Music Video"  
        case "podcast": return "Podcast"  
        case "software": return "App"  
        case "song": return "Song"  
        case "tv-episode": return "TV Episode"  
        default: break  
    }  
    return "Unknown"  
}
```

These are the types of products that this app understands.

It's possible one was missed or that the iTunes Store adds a new product type at some point. If that happens, the switch jumps to the default: case and you'll simply return a string saying "Unknown" — and hopefully help identify and fix the unknown type in an update of the app.

Default and break

Switch statements often have a default: case at the end that just says break.

In Swift, a switch must be exhaustive, meaning that it must have a case for all possible values of the thing that you're looking at.

Here you're looking at kind. Swift needs to know what to do when kind is not any of the known values. That's why you're required to include the default: case, as a catchall for any other possible values of kind.

By the way: unlike in other languages, the case statements in Swift do not need to say break at the end. They do not automatically "fall through" from one case to the other as they do in Objective-C.

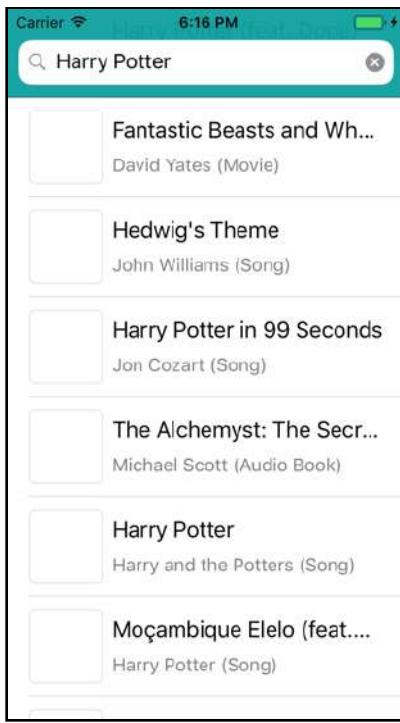
Now the item type should display not as a value from the web service, but instead, as the value you set for each item type:



The product type is a bit more human-friendly

- Run the app and search for software, audio books or e-books to see that the parsing code works. It can take a few tries before you find some because of the enormous quantity of products on the store.

Later on, you'll add a control that lets you pick the type of products that you want to search for, which makes it a bit easier to find just e-books or audiobooks.



The app shows a varied range of products now

Sorting the search results

It'd be nice to sort the search results alphabetically. That's actually quite easy. A Swift Array already has a method to sort itself. All you have to do is tell it what to sort on.

► In **SearchViewController.swift**, in `searchBarSearchButtonClicked(_:)`, right after the call to `parse(data:)` add the following:

```
searchResults.sort(by: { result1, result2 in
    return result1.name.localizedStandardCompare(
        result2.name) == .orderedAscending
})
```

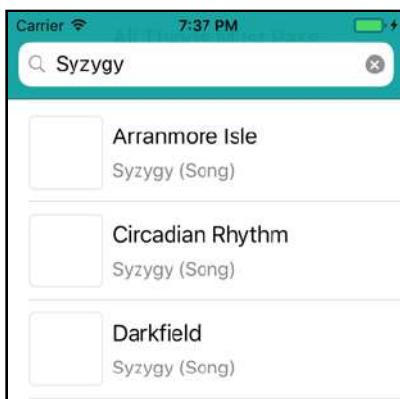
After the results array is fetched, you call `sort(by:)` on the `searchResults` array with a closure that determines the sorting rules. This is identical to what you did in *Checklists* to sort the to-do lists.

In order to sort the contents of the `searchResults` array, the closure will compare

the `SearchResult` objects with each other and return `true` if `result1` comes before `result2`. The closure is called repeatedly on different pairs of `SearchResult` objects until the array is completely sorted.

The comparison of the two objects uses `localizedStandardCompare()` to compare the names of the `SearchResult` objects. Because you used `.orderedAscending`, the closure returns `true` only if `result1.name` comes before `result2.name` — in other words, the array gets sorted from A to Z.

- Run the app and verify that the search results are sorted alphabetically.



The search results are sorted by name

Sorting was pretty easy to add, but there is an even easier way to write this.

Improving the sorting code

- Change the sorting code you just added to:

```
searchResults.sort { $0.localizedStandardCompare($1.name)
    == .orderedAscending }
```

This uses the *trailing* closure syntax to put the closure after the method name, rather than inside the traditional () parentheses as a parameter. It's a small improvement in readability.

More importantly, inside the closure you no longer refer to the two `SearchResult` objects by name but as the special `$0` and `$1` variables. Using this shorthand instead of full parameter names is common in Swift closures. There is also no longer a `return` statement.

- Verify that this works.

Believe it or not, you can do even better. Swift has a very cool feature called **operator overloading**. It allows you to take the standard operators such as + or * and apply them to your own objects. You can even create completely new operator symbols.

It's not a good idea to go overboard with this feature and make operators do something completely unexpected — don't overload / to do multiplications, eh? — but it comes in very handy for sorting.

- Open **SearchResult.swift** and add the following code, outside of the class:

```
func < (lhs: SearchResult, rhs: SearchResult) -> Bool {  
    return lhs.name.localizedStandardCompare(rhs.name) ==  
        .orderedAscending  
}
```

This should look familiar! You're creating a function named < that contains the same code as the closure from earlier. This time, the two **SearchResult** objects are called **lhs** and **rhs**, for left-hand side and right-hand side, respectively.

You have now overloaded the less-than operator so that it takes two **SearchResult** objects and returns **true** if the first one should come before the second, and **false** otherwise. Like so:

```
searchResultA.name = "Waltz for Debby"  
searchResultB.name = "Autumn Leaves"  
  
searchResultA < searchResultB // false  
searchResultB < searchResultA // true
```

- Back in **SearchViewController.swift**, change the sorting code to:

```
searchResults.sort { $0 < $1 }
```

That's pretty sweet. Using the < operator makes it very clear that you're sorting the items from the array in ascending order.

But wait, you can write it even shorter:

```
searchResults.sort(by: <)
```

Wow, it doesn't get much simpler than that! This line literally says, “Sort this array in ascending order.” Of course, this only works because you added your own `func <` to overload the less-than operator so it takes two **SearchResult** objects and compares them.

- Run the app again and make sure everything is still sorted.

Exercise: See if you can make the app sort by the artist name instead.

Exercise. Try to sort in descending order, from Z to A. Tip: use the > operator.

Excellent! You made the app talk to a web service and you were able to convert the data that was received into your own data model object.

The app may not support every product that's shown on the iTunes store, but hopefully it illustrates the principle of how you can take data that comes in slightly different forms and convert it to objects that are more convenient to use in your own apps.

Feel free to dig through the web service API documentation to add the remaining items that the iTunes store sells: <https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/>

► Commit your changes with a message such as "Add fetching data from web service using synchronous network request".

You can find the project files for this chapter under **39 – Networking** in the Source Code folder.



Chapter 40: Asynchronous Networking

Eli Ganim

You've got your app doing network searches and it's working well. The synchronous network calls aren't so bad, are they?

Yes they are, and I'll show you why! Did you notice that whenever you performed a search, the app became unresponsive? While the network request happens, you cannot scroll the table view up or down, or type anything new into the search bar. The app is completely frozen for a few seconds.

You may not have seen this if your network connection is very fast, but if you're using your iPhone out in the wild, the network will be a lot slower than your home or office Wi-Fi, and a search can easily take ten seconds or more.

To most users, an app that does not respond is an app that has crashed. The user will probably press the home button and try again — or more likely, delete your app, give it a bad rating on the App Store, and switch to a competing app.

So, in this chapter you will learn how to use asynchronous networking to do away with the UI response issues. You'll do the following:

- **Extreme synchronous networking:** Learn how synchronous networking can affect the performance of your app by dialing up the synchronous networking to the maximum.
- **The activity indicator:** Add an activity indicator to show when a search is going on so that the user knows something is happening.
- **Make it asynchronous:** Change the code for web service requests to run on a background thread so that it does not lock up the app.



Extreme synchronous networking

Still not convinced of the evils of synchronous networking? Let's slow down the network connection to pretend the app is running on an iPhone that someone may be using on a bus or in a train, not in the ideal conditions of a fast home or office network. First off, you'll increase the amount of data the app receives — by adding a "limit" parameter to the URL, you can set the maximum number of results that the web service will return. The default value is 50, the maximum is 200.

- Open **SearchViewController.swift** and in `iTunesURL(searchText:)`, change the line with the web service URL to the following:

```
let urlString = String(format:  
    "https://itunes.apple.com/search?term=%@&limit=200",  
    encodedText)
```

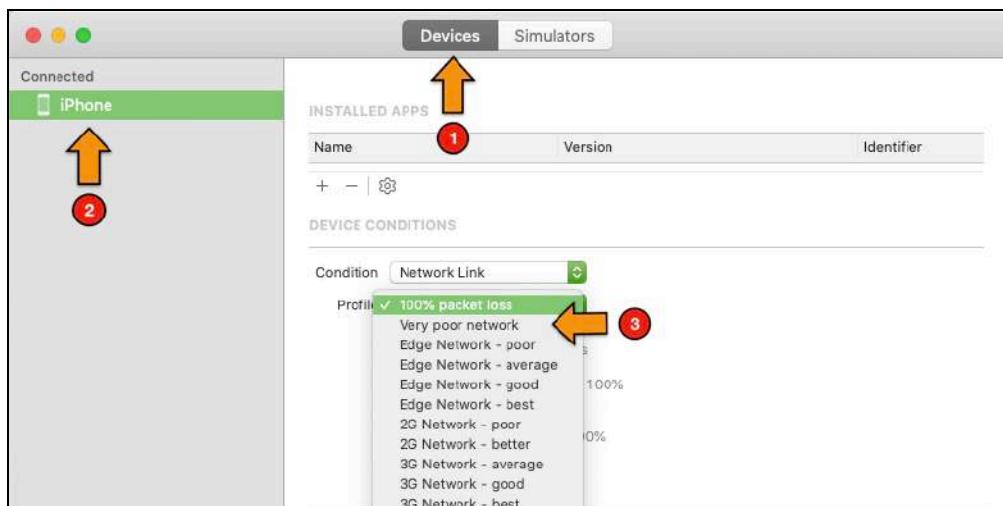
You added `&limit=200` to the URL. Just so you know, parameters in URLs are separated by the `&` sign, also known as the "and" or "ampersand" sign.

- If you run the app now, the search should be quite a bit slower.

Device Conditions

Still too fast for you to see any app response issues? Then use **Device Conditions**. This lets you simulate different network conditions such as bad cellphone network, in order to test your iOS apps. In order to activate it you need to connect a device running iOS 13 to your Mac, then

- Select **Devices and Simulators** from the **Window** menu.
- Choose the **Devices** tab.
- Choose your iPhone from the left pane and scroll down to **Device Conditions**
- Under **Condition** choose **Network Link** and under **Profile** choose **Very poor network**.
- Finally, click on **Start**.



Device Conditions dialog

- Now run the app and search for something. The Device Conditions tool will simulate a slow connection and download the data very slowly.

Tip: If the download still appears very fast, then try searching for some term you haven't used before; the system may be caching the results from a previous search.

Notice how the app totally doesn't respond during this time? It feels like something is wrong. Did the app crash or is it still doing something? It's impossible to tell and very confusing to your users when this happens.

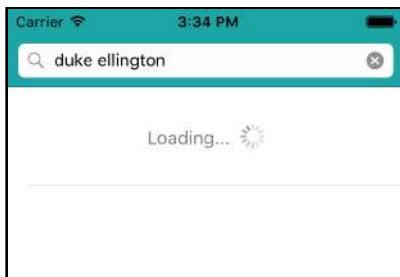
Even worse, if your program is unresponsive for too long, iOS may actually force kill it, in which case it really does crash. You don't want that to happen!

"Ah," you say, "let's show some type of animation to let the user know that the app is communicating with a server. Then at least they will know that the app is busy."

That sounds like a decent thing to do, so let's get to it.

The activity indicator

You've used a spinning activity indicator before in *MyLocations* to show the user that the app was busy. Let's create a new table view cell that you'll show while the app is querying the iTunes store. It will look like this:

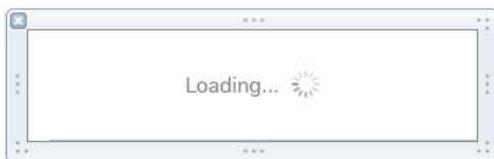


The app shows that it is busy

The activity indicator table view cell

- Create a new, empty nib file. Call it **LoadingCell.xib**. ➤ Drag a new **Table View Cell** on to the canvas. Set its width to **375** points and its height to **80** points.
- Set the reuse identifier of the cell to **LoadingCell** and set the **Selection** attribute to **None**.
- Drag a new **Label** into the cell. Set the title to **Loading...** and change the font to **System 15**. The label's text color should be 50% opaque black.
- Drag a new **Activity Indicator View** into the cell and put it next to the label. Set its **Style** to **Gray** and give it the **Tag** **100**.

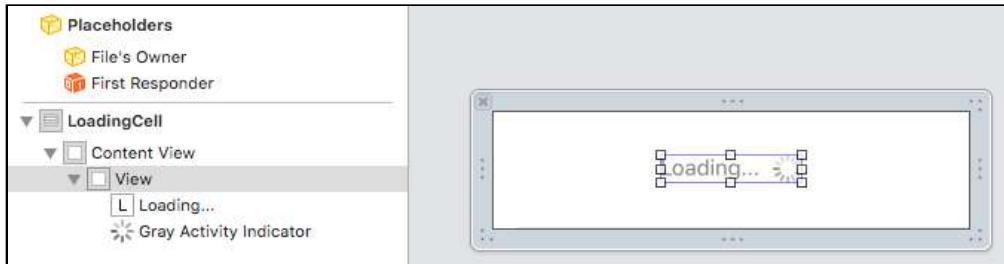
The design should look like this:



The design of the LoadingCell nib

To make this cell work properly on larger screens, you'll add constraints that keep the label and the activity spinner centered in the cell. The easiest way to do this is to place these two items inside a container view and center that.

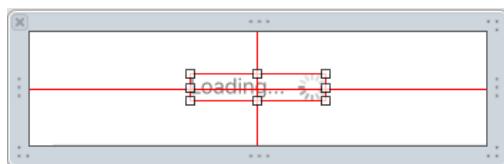
- Select both the Label and the Activity Indicator View – hold down **⌘** to select multiple items. From the Xcode menu bar, choose **Editor** ▶ **Embed In** ▶ **View Without Inset**. This puts a white view behind the selected views.



The label and the spinner now sit in a container view

Note: If you're wondering what the difference is between the **Embed In** ▶ **View** and **Embed In** ▶ **View Without Inset** options in the **Editor** menu is, try it and you should see what happens. The first option adds a view which is slightly larger than the items it encloses because it has *inset* the new view to add some padding around the enclosed items. The second option, the one you used, simply encloses all of the items without any padding.

- With this new container view selected, click the **Align** button and put checkmarks in front of **Horizontally in Container** and **Vertically in Container** to make new constraints.



The container view has red constraints

You end up with a number of red constraints. That's no good; we want to see blue ones. The reason your new constraints are red is that Auto Layout does not know how large this container view should be; you've only added constraints for the view's position, not its size.

To fix this, you're going to add constraints to the label and activity indicator as well, so that the width and height of the container view are determined by the size of the two things inside it.

This is especially important for later when you’re going to translate the app to another language. If the Loading... text becomes larger or smaller, then so should the container view, in order to stay centered inside the cell.

- Select the label and click the **Add New Constraints** button. Simply pin it to all four sides and press **Add 4 Constraints**.
- Repeat this for the Activity Indicator View. You don’t need to pin it to the left because that constraint already exists — pinning the label added it.

Now the constraints for the label and the activity indicator should be all blue. At this point, the container view may still have orange lines indicating that the constraints are fine but that the view’s frame is not in the proper position. If so, select it and choose **Editor** ▶ **Resolve Auto Layout Issues** ▶ **Update Frames** — under Selected Views. This will move the container view into the position dictated by its constraints.

Cool, you now have a cell that automatically adjusts itself to any size screen.

Using the activity indicator cell

To make this special table view cell appear, you’ll follow the same steps as for the “Nothing Found” cell.

- Add the following line to the `TableView.CellIdentifiers` structure in `SearchViewController.swift`:

```
static let loadingCell = "LoadingCell"
```

- And register the nib in `viewDidLoad()`:

```
cellNib = UINib(nibName: TableView.CellIdentifiers.loadingCell,  
                 bundle: nil)  
tableView.register(cellNib, forCellReuseIdentifier:  
                  TableView.CellIdentifiers.loadingCell)
```

You now have to come up with some way to let the table view’s data source know that the app is currently in a state of downloading data from the server. The simplest way to do that is to add another boolean flag. If this variable is `true`, then the app is downloading stuff and the new Loading... cell should be shown; if the variable is `false`, you show the regular contents of the table view.

- Add a new instance variable:

```
var isLoading = false
```

- Change `tableView(_:numberOfRowsInSection:)` to:

```
func tableView(_ tableView: UITableView,
              numberOfRowsInSection section: Int) -> Int {
    if isLoading {
        return 1
    } else if !hasSearched {
        .
    } else if .
    .
}
```

The `if isLoading` condition returns 1, because you need a row in order to show a cell.

- Update `tableView(_:cellForRowAt:)` as follows:

```
func tableView(_ tableView: UITableView,
              cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // New code
    if isLoading {
        let cell = tableView.dequeueReusableCell(withIdentifier:
            TableView.CellIdentifiers.loadingCell, for: indexPath)

        let spinner = cell.viewWithTag(100) as!
            UIActivityIndicatorView
        spinner.startAnimating()
        return cell
    } else
    // End of new code
    if searchResults.count == 0 {
        .
    }
}
```

You added an `if` condition to return an instance of the new Loading... cell. It also looks up the `UIActivityIndicatorView` by its tag and then tells the spinner to start animating. The rest of the method stays the same.

- Change `tableView(_:willSelectRowAt:)` to:

```
func tableView(_ tableView: UITableView,
              willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    if searchResults.count == 0 || isLoading { // Changed
        return nil
    } else {
        return indexPath
    }
}
```

You added `|| isLoading` to the `if` statement. Just like you don't want users to select the "Nothing Found" cell, you also don't want them to select the "Loading..." cell, so you return `nil` in both cases.



There's only one thing remaining: you should set `isLoading` to `true` before you make the HTTP request to the iTunes server, and also reload the table view to make the Loading... cell appear.

- Change `searchBarSearchButtonClicked(_:)` to:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
    if !searchBar.text!.isEmpty {  
        searchBar.resignFirstResponder()  
        // New code  
        isLoading = true  
        tableView.reloadData()  
        // End of new code  
        . . .  
        isLoading = false  
        tableView.reloadData()  
    }  
}
```

Before you do the networking request, you set `isLoading` to `true` and reload the table to show the activity indicator.

After the request completes and you have the search results, you set `isLoading` back to `false` and reload the table again to show the `SearchResult` objects.

Makes sense, right? Let's fire up the app and see this in action!

Testing the new loading cell

- Run the app and perform a search. While search is taking place the Loading... cell with the spinning activity indicator should appear...

...or should it?!

The sad truth is that there is no spinner to be seen. And in the unlikely event that it does show up for you, it won't be spinning — try it with Network Link Conditioner enabled.

- To show you why, first change `searchBarSearchButtonClicked(_:)` as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()
        isLoading = true
        tableView.reloadData()
        /*
            . . . the networking code (commented out) . .
        */
    }
}
```

Note that you don't have to remove anything from the code — simply comment out everything after the first call to `tableView.reloadData()`.

- Run the app and do a search. Now the activity spinner does show up!

So at least you know that part of the code is working fine. But with the networking code enabled, the app is not only totally unresponsive to any input from the user, it also doesn't want to redraw its screen. What's going on here?

The main thread

The CPU (Central Processing Unit) in older iPhone and iPad models has one core, which means it can only do one thing at a time. More recent models have a CPU with two cores, which allows for a whopping two computations to happen simultaneously. Your Mac may have 4 cores.

With so few cores available, how come modern computers can have many more applications and other processes running at the same time?

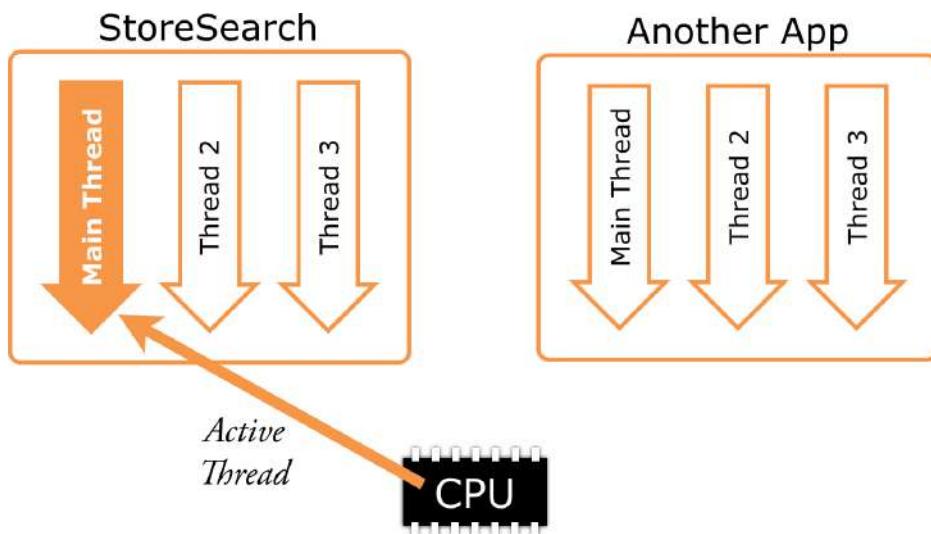
To get around the hardware limitation of having only one or two CPU cores, most computers, including the iPhone and iPad, use **preemptive multitasking** and **multithreading** to give the illusion that they can do many things at once.

Multitasking is something that happens between different apps. Each app is said to have its own **process** and each process is given a small portion of each second of CPU time to perform its jobs. Then it is temporarily halted, or *pre-empted*, and control is given to the next process.



Each process contains one or more **threads**. Each process is given a bit of CPU time to do its work. The process splits up that time among its threads. Each thread typically performs its own work and is as independent as possible from the other threads within that process.

An app can have multiple threads and the CPU switches between them:



If you go into the Xcode debugger and pause the app, the debugger will show you which threads are currently active and what they were doing before you stopped them.

For the StoreSearch app, there were apparently six threads at the time the following screenshot was taken:



Most of these threads are managed by iOS itself and you don't have to worry about them. Also, you may see less or more than six threads. However, there is one thread that requires special care: the **main thread**. In the image above, that is **Thread 1**.

The main thread is the app's initial thread and from there all the other threads are spawned. The main thread is responsible for handling user interface events and also for drawing the UI. Most of your app's activities take place on the main thread. Whenever the user taps a button in your app, it is the main thread that performs your action method.

Because it's so important, you should be careful not to hold up, or "block," the main thread. If your action method takes more than a fraction of a second to run, then doing all these computations on the main thread is not a good idea because that would lock up your main thread.

The app becomes unresponsive because the main thread cannot handle any UI events while you're keeping it busy doing something else — and if the operation takes too long, the app may even be killed by the system.

In *StoreSearch*, you're doing a lengthy network operation on the main thread. It could potentially take many seconds, maybe even minutes, to complete.

After you set the `isLoading` flag to `true`, you tell the `tableView` to reload its data so that the user can see the spinning animation. But that never comes to pass. Telling the table view to reload schedules a "redraw" event, but the main thread gets no chance to handle that event as you immediately start the networking operation, keeping the main thread busy for a long time.

This is why the current synchronous approach to doing networking is bad: **Never block the main thread**. It's one of the cardinal sins of iOS programming!

Making it asynchronous

To prevent blocking the main thread, any operation that might take a while to complete should be **asynchronous**. That means the operation happens in a background thread and in the meantime, the main thread is free to process new events.

That is not to say you should create your own thread. If you've programmed on other platforms before, you may not think twice about creating new threads, but on iOS that is often not the best solution.

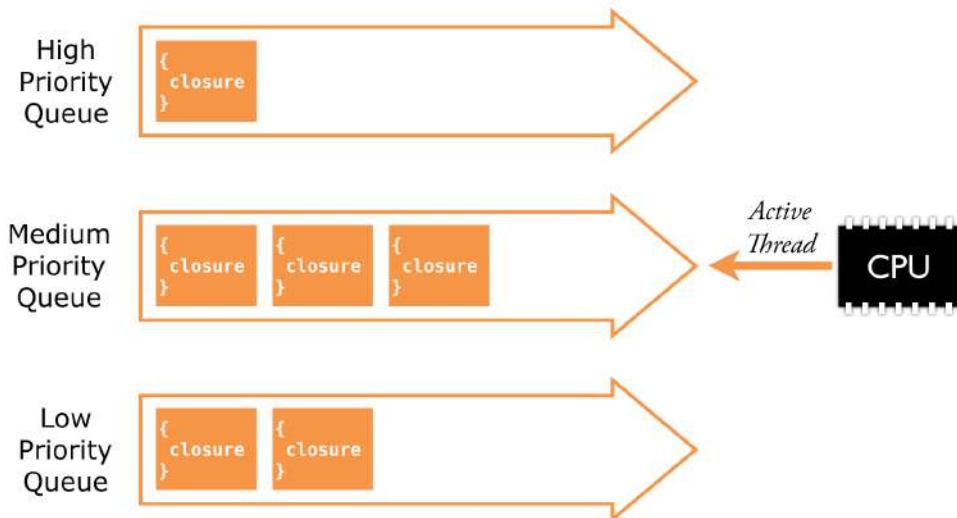
You see, threads are tricky. Not threads per se, but doing things in parallel. There's no need to go into too much detail here, but generally, you want to avoid the situation where two threads are modifying the same piece of data at the same time. That can lead to very surprising — but not very pleasant — results.



Rather than making your own threads, iOS has several more convenient ways to start background processes. For this app you'll be using **queues** and **Grand Central Dispatch**, or GCD. GCD greatly simplifies tasks that require parallel programming. You've already briefly played with GCD in *MyLocations*, but now you'll put it to even better use.

In short, GCD has a number of queues with different priorities. To perform a job in the background, you put the job in a closure and then pass that closure to a queue and forget about it. It's as simple as that.

GCD will get the closures — or “blocks” as it calls them — from the queues one-by-one and perform their code in the background. Exactly how it does that is not important, you're only guaranteed it happens on a background thread somewhere. Queues are not exactly the same as threads, but they use threads to do their job.



Queues have a list of closures to perform on a background thread

Putting the web request in a background thread

To make the web service requests asynchronous, you're going to put the networking part from `searchBarSearchButtonClicked(_:)` into a closure and then place that closure on a medium priority queue.

- Change searchBarSearchButtonClicked(_:) as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {

        searchResults = []
        // Replace all code after this with new code below
        // 1
        let queue = DispatchQueue.global()
        let url = self.iTunesURL(searchText: searchBar.text!)
        // 2
        queue.async {

            if let data = self.performStoreRequest(with: url) {
                self.searchResults = self.parse(data: data)
                self.searchResults.sort(by: <)
                // 3
                print("DONE!")
                return
            }
        }
    }
}
```

Here is the new stuff:

1. This gets a reference to the queue. You’re using a “global” queue, which is a queue provided by the system. You can also create your own queues, but using a standard queue is fine for this app. You also get the URL for your search here, outside the closure.
2. Once you have the queue, you can dispatch a closure on it — everything between `queue.async {` and the closing `}` is the closure. Whatever code in the closure will be put on the queue and be executed asynchronously in the background. After scheduling this closure, the main thread is immediately free to continue. It is no longer blocked.
3. Inside the closure, you remove the code that reloads the table view after the search is done, as well as the error handling code. For now, this has been replaced by `print()` statements. There is a good reason for this and we’ll get to it in a second. First let’s try the app again.

- Run the app and do a search. The “Loading...” cell should be visible, complete with animating spinner! After a short while you should see the “DONE!” message appear in the Console.

Of course, the Loading... cell sticks around forever because you still haven’t told it to go away.

Putting UI updates on the main thread

The reason you need to remove all the user interface code from the closure — and moved getting the search URL outside the closure — is that UIKit has a rule that UI code should *always* be performed on the main thread. This is important!

Accessing the same data from multiple threads can create all sorts of misery, so the designers of UIKit decided that changing the UI from other threads would not be allowed. That means you cannot reload the table view from within this closure, because it runs on a queue that is on a background thread, not the main thread.

As it happens, there is also a “main queue” that is associated with the main thread. If you need to do anything on the main thread from a background queue, you can simply create a new closure and schedule the main thread actions on the main queue.

- Replace the line in `searchBarSearchButtonClicked(_:)` that says `print("DONE!")` with:

```
DispatchQueue.main.async {
    self.isLoading = false
    self.tableView.reloadData()
}
```

With `DispatchQueue.main.async` you can schedule a new closure on the main queue. This new closure sets `isLoading` back to `false` and reloads the table view. Note that `self` is required because this code sits inside a closure.

- Try it out. With these changes in place, your networking code no longer occupies the main thread and the app suddenly feels a lot more responsive!

All kinds of queues

When working with GCD queues you will often see this pattern:

```
let queue = DispatchQueue.global()
queue.async {
    // code that needs to run in the background

    DispatchQueue.main.async {
        // update the user interface
    }
}
```

Basically, while you do your work in a background thread, you still have to switch over to the main thread to do any user interface updates. That's just the way it is.

There is also `queue.sync`, without the “a,” which takes the next closure from the queue and performs it in the background, but makes you wait until that closure is done. That can be useful in some cases but most of the time you’ll want to use `queue.async`. No one likes to wait!

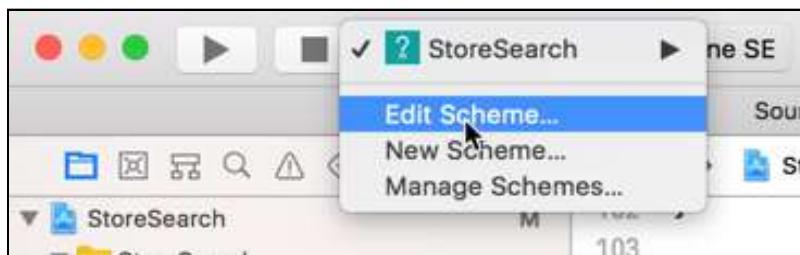
The main thread checker

You read previously that you should not run UI code on a background thread. However, till Xcode 9, there was no easy way to discover UI code running on background threads except by scouring the source code laboriously line-by-line trying to determine what code ran on the main thread and what ran on a background thread.

With Xcode 9, Apple introduced a new diagnostic setting called the Main Thread Checker which would warn you if you had any UI code running on a background thread.

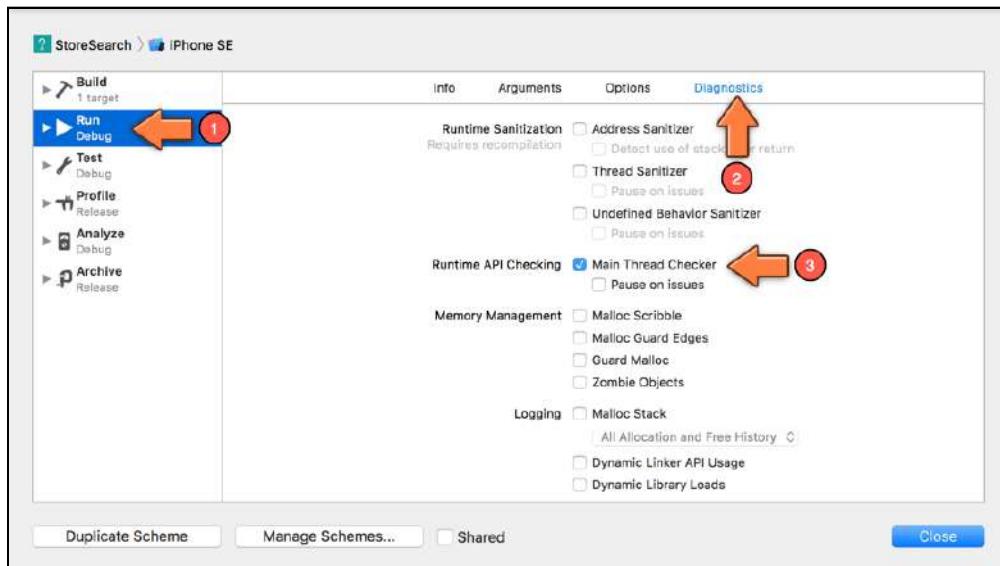
This setting is supposed to be enabled by default, but if it is not, you can enable it quite easily — It’s highly recommended that you have it enabled at all times if possible since it can be quite invaluable.

- Click on the scheme dropdown in the Xcode toolbar and select Edit Scheme...



Edit scheme

- Select Run in the left panel, switch to the **Diagnostics** tab, and make sure **Main Thread Checker** is checked under Runtime API Checking.



Main Thread Checker setting

- Now, move the following line from outside the closure:

```
let url = self.iTunesURL(searchText: searchBar.text!)
```

To be inside the closure like this:

```
queue.async {
    let url = self.iTunesURL(searchText: searchBar.text!)
    ...
}
```

- Run StoreSearch and do a search for an item, you should see something like the following in the Xcode Console:

```
Main Thread Checker: UI API called on a background thread: -
[UISearchBar text]
PID: 12986, TID: 11267540, Thread name: (none), Queue name:
com.apple.root.default-qos, QoS: 0
Backtrace:
4  StoreSearch                      0x000000010bccfa75
$S11StoreSearch@B14ViewController@09searchBarB13ButtonClickedyyS
o08UISearchF0CFyycfU_ + 469
5  StoreSearch                      0x000000010bcd0101
$S11StoreSearch@B14ViewController@09searchBarB13ButtonClickedyyS
```

```
008UISearchF0CFyycfU_TA + 17
6 StoreSearch 0x000000010bcd02bd
$SIeg_IeyB_TR + 45
7 libdispatch.dylib 0x000000010f3a1225
_dispatch_call_block_and_release + 12
8 libdispatch.dylib 0x000000010f3a22e0
_dispatch_client_callout + 8
9 libdispatch.dylib 0x000000010f3a4d8a
_dispatch_queue_override_invoke + 1028
10 libdispatch.dylib 0x000000010f3b2daa
_dispatch_root_queue_drain + 351
11 libdispatch.dylib 0x000000010f3b375b
_dispatch_worker_thread2 + 130
12 libsystem_pthread.dylib 0x000000010f791169
_pthread_wqthread + 1387
13 libsystem_pthread.dylib 0x000000010f790be9
start_wqthread + 13
2018-07-28 11:39:02.726132+0200 StoreSearch[12986:11267540]
[reports] Main Thread Checker: UI API called on a background
thread: -[UISearchBar text]
PID: 12986, TID: 11267540, Thread name: (none), Queue name:
com.apple.root.default-qos, QoS: 0
Backtrace:
4 StoreSearch 0x000000010bccfa75
$S11StoreSearch0B14ViewControllerC09searchBarB13ButtonClickedyyS
008UISearchF0CFyycfU_ + 469
5 StoreSearch 0x000000010bcd0101
$S11StoreSearch0B14ViewControllerC09searchBarB13ButtonClickedyyS
008UISearchF0CFyycfU_TA + 17
6 StoreSearch 0x000000010bcd02bd
$SIeg_IeyB_TR + 45
7 libdispatch.dylib 0x000000010f3a1225
_dispatch_call_block_and_release + 12
8 libdispatch.dylib 0x000000010f3a22e0
_dispatch_client_callout + 8
9 libdispatch.dylib 0x000000010f3a4d8a
_dispatch_queue_override_invoke + 1028
10 libdispatch.dylib 0x000000010f3b2daa
_dispatch_root_queue_drain + 351
11 libdispatch.dylib 0x000000010f3b375b
_dispatch_worker_thread2 + 130
12 libsystem_pthread.dylib 0x000000010f791169
_pthread_wqthread + 1387
13 libsystem_pthread.dylib 0x000000010f790be9
start_wqthread + 13
```

You might also notice that the Xcode toolbar's activity view now has a purple icon and that there's a purple icon on the right corner of the jump bar, where errors are normally displayed.



Purple icons indicating Main Thread Checker issues

If you click on the icon in the activity view, you will be taken to the **Runtime** tab of the **Issue navigator**, where you can click on a listed issue to be taken to the offending line in your source code:



Issue navigator

And you finally see what the issue is — you access the data from a UI control, the Search Bar, in a background thread. It might be better to do this in the main thread. Since we created this issue to illustrate the background thread checker, the fix is simple, just move the line of code back to where it was originally.

Committing your code

- With this important improvement, the app deserves a new version number. So commit the changes and create a tag for **v0.2**. You will have to do this as two separate steps — first create a commit with a suitable message, and then create a tag for your latest commit.

You can find the project files for this chapter under **40 – Asynchronous Networking** in the Source Code folder.

Chapter 41: URLSession

Eli Ganim

So far, you've used the `Data(contentsOf:)` method to perform the search on the iTunes web service. That is great for simple apps, but there's another way to do networking that is more powerful.

iOS itself comes with a number of different classes for doing networking, from low-level sockets stuff that is only interesting to really hardcore network programmers, to convenient classes such as `URLSession`.

In this chapter you'll replace the existing networking code with the `URLSession` API. That is the API the pros use for building real apps, but don't worry, it's not more difficult than what you've done before — just more powerful.

You'll cover the following items in this chapter:

- **Branch it:** Creating Git branches for major code changes.
- **Put URLSession into action:** Use the `URLSession` class for asynchronous networking instead of downloading the contents of a URL directly.
- **Cancel operations:** Canceling a running network request when a second network request is initiated.
- **Search different categories:** Allow the user to select a specific iTunes Store category to search in instead of returning items from all categories.
- **Download the artwork:** Download the images for search result items and display them as part of the search result listing.
- **Merge the branch:** Merge your changes from your working Git branch back to your master branch.



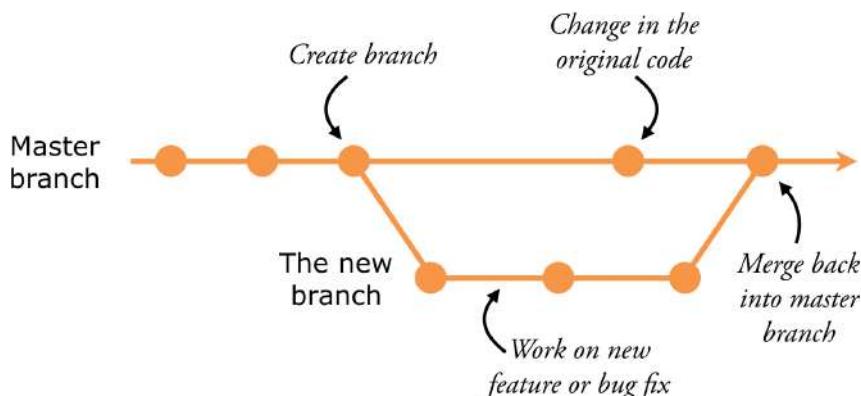
Branching it

Whenever you make a big change to the code — such as replacing all the networking stuff with URLSession — there is a possibility that you’ll mess things up. That’s why it’s smart to create a Git **branch** first.

The Git repository contains a history of all the app’s code, but it can also contain this history along different paths.

You just finished the first version of the networking code and it works pretty well. Now you’re going to completely replace that with a — hopefully — better solution. In doing so, you may want to commit your progress at several points along the way.

What if it turns out that switching to URLSession wasn’t such a good idea after all? Then you’d have to restore the source code to a previous commit from before you started making those changes. In order to avoid this potential mess, you can make a branch instead.

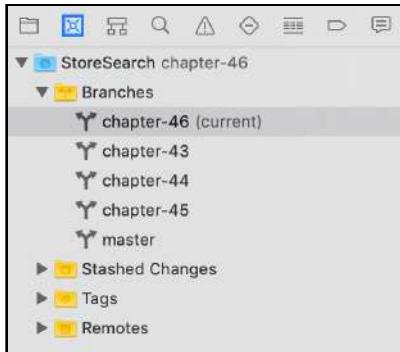


Branches in action

Every time you’re about to add a new feature to your code or have a bug to fix, it’s a good idea to make a new branch and work on that. When you’re done and are satisfied that everything works as it should, merge your changes back into the master branch. Different people use different branching strategies but this is the general principle.

So far you have been committing your changes to the “master” branch. Now you’re going to make a new branch, let’s call it “urlsession,” and commit your changes to that. When you’re done with this new feature you will merge everything back into the master branch.

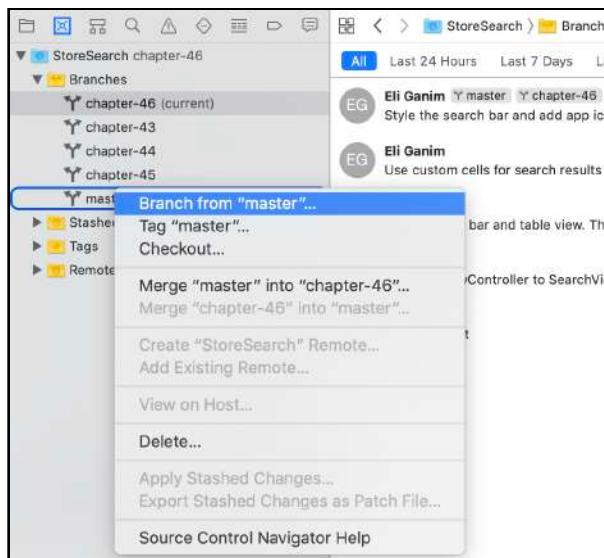
You can find the branches for your repository in the **Source Control navigator**:



The Source Control branch list

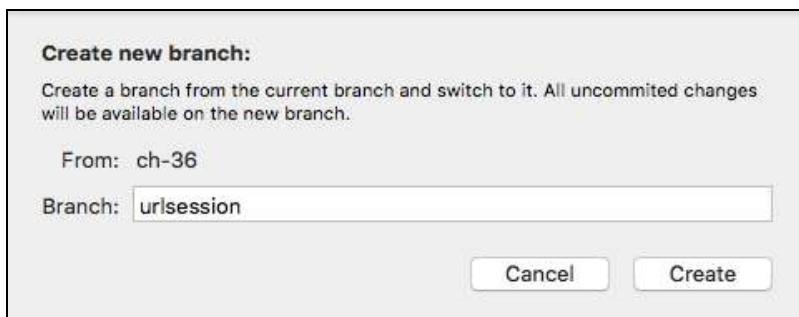
Note: In the above screenshot, there are multiple branches already — one branch for each chapter of the book. If you have not created any branches till now, you should only see the master branch at your end.

- Select **master** — or whatever is your current branch — from the branch list, and right-click on the branch name to get a context-menu with possible actions. Select **Branch from "master"...**:



The branch context-menu

- You will get a dialog asking for the new branch name. Enter **urlsession** as the new name and click **Create**.



Creating a new branch

When Xcode is done, you'll see that a new “urlsession” branch has been added and that it is now the current one.

This new branch contains the exact same source code and history as the master branch, or whichever branch you used as the parent for the new branch. But from here on out the two paths will diverge — any changes you make happen on the “urlsession” branch only.

Putting URLSession into action

Good, now that you're in a new branch, it's safe to experiment with these new APIs.

- First, remove `performStoreRequest(with:)` from **SearchViewController.swift**. Yup, that's right, you won't be needing that method anymore.

Don't be afraid to remove old code. Some developers only comment out the old code but leave it in the project, just in case they may need it again some day. You don't have to worry about that because you're using source control. Should you really need it, you can always find the old code in the Git history. Besides, if the experiment should fail, you can simply throw away this branch and switch back to the “original” one.

Anyway, on to URLSession. This is a closure-based API, meaning that instead of making a delegate, you pass it a closure containing the code that should be performed once the response from the server has been received. URLSession calls this closure the *completion handler*.

- Change searchBarSearchButtonClicked(_:) as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {

        .
        .
        searchResults = []
        // Replace all code after this with new code below
        // 1
        let url = iTunesURL(searchText: searchBar.text!)
        // 2
        let session = URLSession.shared
        // 3
        let dataTask = session.dataTask(with: url,
            completionHandler: { data, response, error in
                // 4
                if let error = error {
                    print("Failure! \(error.localizedDescription)")
                } else {
                    print("Success! \(response!)")
                }
            })
        // 5
        dataTask.resume()
    }
}
```

This is what the changes do:

1. Create the URL object using the search text, just like before.
2. Get a shared URLSession instance, which uses the default configuration with respect to caching, cookies, and other web stuff.

If you want to use a different configuration — for example, to restrict networking to when Wi-Fi is available but not when there is only cellular access — then you have to create your own URLSessionConfiguration and URLSession objects. But for this app, the default one will be fine.

3. Create a data task. Data tasks are for fetching the contents of a given URL. The code from the completion handler will be invoked when the data task has received a response from the server.
4. Inside the closure, you're given three parameters: data, response, and error. These are all optionals so they can be nil and have to be unwrapped before you can use them.

If there was a problem, error contains an Error object describing what went wrong. This happens when the server cannot be reached or the network is down or there is some other hardware failure.



If `error` is `nil`, the communication with the server succeeded; `response` holds the server's response code and headers, and `data` contains the actual data fetched from the server, in this case a blob of JSON.

For now, you simply use a `print()` to show success or failure.

- Finally, once you have created the data task, you need to call `resume()` to start it. This sends the request to the server on a background thread. So, the app is immediately free to continue — `URLSession` is as asynchronous as they come.

With these changes made, you can run the app and see what `URLSession` makes of it.

- Run the app and search for something. After a second or two you should see a Console message saying “Success!” followed by a dump of the HTTP response headers.

Excellent!

A brief review of closures

You've seen closures a few times now. They are a really powerful feature of Swift and you can expect to be using them all the time when you're working with Swift code. So, it's good to have at least a basic understanding of how they work.

A closure is simply a piece of source code that you can pass around just like any other type of object. The difference between a closure and regular source code is that the code from the closure does not get performed right away. Instead, it is stored in a “closure object” and can be performed at a later point, even more than once.

That's exactly what `URLSession` does: it holds on to the “completion handler” closure and only performs it when a response is received from the web server or when a network error occurs.

A closure typically looks like this:

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    data, response, error in  
    . . . source code . . .  
})
```

The thing behind `completionHandler` inside the `{ }` brackets is the closure. The form of a closure is always:

```
{ parameters in  
    your source code  
}
```

Or without parameters:

```
{  
    your source code  
}
```

Just like a method or function, a closure can accept parameters. They are separated from the source code by the “in” keyword. In URLSession’s completion handler the parameters are data, response, and error.

Thanks to Swift’s type inference, you don’t need to specify the data types of the parameters. However, you could write them out in full if you wanted to:

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    (data: Data?, response: URLResponse?, error: Error?) in  
    . . .  
})
```

Tip: For a parameter without the type annotation, you can Option-click in Xcode to find out what its type is. This trick works for any symbol in your code.

If you don’t care about a particular parameter you can substitute it with _, the *wildcard* symbol:

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    data, _, error in  
    . . .  
})
```

If a closure is really simple, you can leave out the parameter list altogether and use \$0, \$1, and so on as the parameter names.

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    print("My parameters are \$(\$0), \$(\$1), \$(\$2)")  
})
```

You wouldn’t do that with URLSession’s completion handler, though. It’s much easier if you know the parameters are called data, response, and error than remembering what \$0, \$1, and \$2 stand for.

If a closure is the last parameter of a method, you can use *trailing* syntax to simplify the code a little:

```
let dataTask = session.dataTask(with: url) {  
    data, response, error in  
    . . .  
}
```

Now the closure appears after the closing parenthesis, not inside. Many people, myself included, find this more natural to read.

Closures are useful for other things too, such as initializing objects and lazy loading:

```
lazy var dateFormatter: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateStyle = .medium
    formatter.timeStyle = .short
    return formatter
}()
```

The code to create and initialize the `NSDateFormatter` object sits inside a closure. The `()` at the end causes the closure to be *evaluated* and the returned object is put inside the `dateFormatter` variable. This is a common trick for placing complex initialization code right next to the variable declaration.

It's no coincidence that closures look a lot like functions. In Swift, closures, methods, and functions are really all the same thing. For example, you can supply the name of a method or function when a closure is expected, as long as the parameters match:

```
let dataTask = session.dataTask(with: url,
                                completionHandler: myHandler)
    .
    .
func myHandler(data: Data?, response: URLResponse?,
               error: Error?) {
    .
}
```

The above somewhat negates one of the prime benefits of closures — keeping all the code in the same place — but there are situations where this is quite useful when the method acts as a “mini” delegate.

One final thing to be aware of with closures is that they *capture* any variables used inside the closure, including `self`. This can create ownership cycles, often leading to memory leaks. To avoid this, you can supply a *capture list*:

```
let dataTask = session.dataTask(with: url) {
    [weak self] data, response, error in
    .
}
```

Whenever you access a property or call a method, you’re implicitly using `self`. Inside a closure, however, Swift requires that you always write `self.` in front of the method or property name. This makes it clear that `self` is being captured by the closure:

```
let dataTask = session.dataTask(with: url) {  
    data, response, error in  
    self.callSomeMethod() // self is required  
}
```

`SearchViewController` doesn’t have to worry about `URLSession` capturing `self` because the data task is only short-lived, while the view controller sticks around for as long as the app itself. This ownership cycle is quite harmless. As you add more functionality to `StoreSearch` you *will* have to use `[weak self]` with `URLSession` or the app might crash and burn!

Note: Swift also has the concept of “no escape” closures. We won’t go into that here, except to mention that no-escape closures don’t capture `self`, so you don’t have to write “`self.`” everywhere. Nice, but you can only use such closures under very specific circumstances!

Handling status codes

After a successful request, the app prints the HTTP response from the server. The response object might look something like this:

```
Success! <NSHTTPURLResponse: 0x7f8b19e38d10> { URL: https://  
itunes.apple.com/search?term=metallica&limit=200 } {  
status code: 200, headers {  
    "Cache-Control" = "no-transform, max-age=41";  
    Connection = "keep-alive";  
    "Content-Encoding" = gzip;  
    "Content-Length" = 34254;  
    "Content-Type" = "text/javascript; charset=utf-8";  
    Date = "Fri, 21 Aug 2015 09:53:20 GMT";  
    . . .  
} }
```

If you’ve done any web development before, this should look familiar. These “HTTP headers” are always the first part of the response from a web server that precedes the actual data you’re receiving. The headers give additional information about the communication that just happened.



What you’re especially interested in is the *status code*. The HTTP protocol has defined a number of status codes that tell clients whether the request was successful or not. No doubt you’re familiar with 404, web page not found.

The status code you want to see is 200 OK, which indicates success — Wikipedia has the complete list of codes, [wikipedia.org/wiki/List_of_HTTP_status_codes](https://en.wikipedia.org/w/index.php?title=List_of_HTTP_status_codes&oldid=90300000).

To make the error handling of the app a bit more robust, let’s check to make sure the HTTP response code really is 200. If not, something has gone wrong and we can’t assume that the received data contains the JSON we’re after.

► Change the contents of the `completionHandler` to:

```
if let error = error {
    print("Failure! \(error.localizedDescription)")
} else if let httpResponse = response as? HTTPURLResponse,
          httpResponse.statusCode == 200 {
    print("Success! \(data!)")
} else {
    print("Failure! \(response!)")
}
```

The `response` parameter has the data type `URLResponse`, but that doesn’t have a property for the status code. Because you’re using the HTTP protocol, what you’ve really received is an `HTTPURLResponse` object, a subclass of `URLResponse`. So, first you cast it to the proper type, and then look at its `statusCode` property — you’ll consider the job a success only if it is 200.

Notice the use of the comma inside the `if let` statement to combine these checks into a single line. You could also have written it with a second `if`, but it might be harder to read:

```
} else if let httpResponse = response as? HTTPURLResponse {
    if httpResponse.statusCode == 200 {
        print("Success! \(data!)")
    }
}
```

Whenever you need to unwrap an optional and also check the value of that optional, using `if let ...`, `...` is the nicest way to do that.

► Run the app and search for something. You should now see something like:

```
Success! 295831 bytes
```

Since your received data is in the form of a `Data` object, unlike text, its content can’t be printed out. So, you just get the length of the data instead.



It's always a good idea to actually test your error handling code. So, let's first fake an error and get that out of the way.

- In `iTunesURL(searchText:)`, change the URL string to:

```
"https://itunes.apple.com/searchLOL?term=%@&limit=200"
```

Here, I've changed the endpoint from `search` to `searchLOL`. It doesn't really matter what you type there, as long as it's something that cannot possibly exist on the iTunes server.

- Run the app again. Now a search should respond with something like this:

```
Failure! <NSHTTPURLResponse: 0x7ff76b42d4b0> { URL: https://itunes.apple.com/searchLOL?term=metallica&limit=200 } {
  status code: 404, headers {
    Connection = "keep-alive";
    "Content-Length" = 207;
    "Content-Type" = "text/html; charset=iso-8859-1";
  }
}
```

As you can see, the status code is now 404 — there is no `searchLOL` page — and the app correctly considers this a failure. That's a good thing too, because if you were to convert the value of `data` to text, `data` now contains the following:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /searchLOL was not found on this server.</p>
</body></html>
```

That is definitely not JSON but HTML. If you tried to convert that into JSON objects, you'd fail horribly.

Great, so the error handling works! Let's parse received JSON data.

Parsing the data

- First, put `iTunesURL(searchText:)` back to the way it was — use `⌘+Z` to undo.
- In the `completionHandler`, replace the `print("Success! \(data)")` line with:

```
if let data = data {
```

```
    self.searchResults = self.parse(data: data)
    self.searchResults.sort(by: <)
    DispatchQueue.main.async {
        self.isLoading = false
        self.tableView.reloadData()
    }
    return
}
```

This unwraps the optional object from the `data` parameter and then calls `parse(data:)` to turn the dictionary's contents into `SearchResult` objects, just like you did before. Finally, you sort the results and put everything into the table view. This should look very familiar.

It's important to realize that the completion handler closure won't be performed on the main thread. Because `URLSession` does all the networking asynchronously, it will also call the completion handler on a background thread.

Parsing the JSON and sorting the list of search results could potentially take a while — not seconds but possibly long enough to be noticeable. You don't want to block the main thread while that is happening, so it's preferable that this happens in the background too.

But when the time comes to update the UI, you need to switch back to the main thread — them's the rules. That's why you wrap the reloading of the table view in a `DispatchQueue.main.async` closure.

If you forget to do this, your app may still appear to work. That's the insidious thing about working with multiple threads. However, it may also crash in all kinds of mysterious ways. So remember, UI stuff should always happen on the main thread. Write it on a Post-It note and stick it to your screen!

► Run the app. The search should work again. You have successfully replaced the old networking code with `URLSession`!

Tip: If you want to determine via code whether a particular piece of code is being run on the main thread or not, add the following code snippet:

```
print("On main thread? " + (Thread.current.isMainThread ?
    "Yes" : "No"))
```

Go ahead, paste this at the top of the `completionHandler` closure and see what it says.

Of course, the official framework documentation should be your first stop. Usually when a method takes a closure, the docs mention whether it is



performed on the main thread or not. But if you're not sure, or just can't find it in the docs, add the above `print()` and be enlightened.

Handling errors

- At the very end of the completion handler closure, below the `if` statements, add the following:

```
DispatchQueue.main.async {
    self.hasSearched = false
    self.isLoading = false
    self.tableView.reloadData()
    self.showNetworkError()
}
```

The code execution reaches here only if something went wrong. You call `showNetworkError()` to let the user know about the problem.

Note that you do `tableView.reloadData()` here too, because the contents of the table view need to be refreshed to get rid of the Loading... indicator. And of course, all this happens on the main thread.

Exercise: Why doesn't the error alert show up on success? After all, the above piece of code sits at the bottom of the closure, so doesn't it always get executed?

Answer: Upon successfully loading the data, the `return` statement exits the closure after the search results get displayed in the table view. So in that case, execution never reaches the bottom of the closure.

- Fake an error situation to test that the error handling code really works.

Testing errors is not a luxury! The last thing you want is for your app to crash when a networking error occurs because of faulty error handling code. I've worked on codebases where it was obvious the previous developer never bothered to verify that the app was able to recover from errors — that's probably why they were the *previous* developer.

Things will go wrong in the wild and your app better be prepared to deal with it. As the MythBusters say, "failure is always an option."



Does the error handling code work? Great! Time to add some new networking features to the app.

- This is a good time to commit your changes. Remember, this commit only happens on the "urlsession" branch, not on the master branch.

Cancelling operations

What happens when a search takes a long time and the user starts a second search while the first one is still going? The app doesn't disable the search bar, so it's possible for the user to do this. When dealing with networking — or any asynchronous process, really — you have to think these kinds of situations through.

There is no way to predict what happens, but it will most likely be a strange experience for the user.

They might see the results from their first search, which they are no longer expecting, only for that to be replaced by the results of the second search a few seconds later. Confusing!

But there is no guarantee the first search completes before the second, so the results from search #2 may arrive first and then get overwritten by the results from search #1, which is definitely not what the user wanted to see either.

Because you're no longer blocking the main thread, the UI always accepts user input, and you cannot assume the user will sit still and wait until the request is done.

You can usually fix this in one of two ways:

1. Disable all controls. The user cannot tap anything while the operation is taking place. This does not mean you're blocking the main thread; you're just making sure the user cannot mess up the order of things.
2. Cancel the on-going request when the user initiates a new request.

For this app, you're going to pick the second solution because it makes for a nicer user experience. Every time the user performs a new search, you cancel the previous request. URLSession makes this easy: data tasks have a `cancel()` method.

When you created the data task, you were given a URLSessionDataTask object, and you placed this into a local constant named `dataTask`. Cancelling the task, however, needs to happen the *next* time `searchBarSearchButtonClicked(_ :)` is called.

Storing the URLSessionDataTask object into a local variable isn't good enough anymore; you need to keep that reference beyond the scope of the method. In other words, you have to store it in an instance variable.

- Add the following instance variable to **SearchViewController.swift**:

```
var dataTask: URLSessionDataTask?
```

This is an optional because you won't have a data task until the user performs a search.

- In `searchBarSearchButtonClicked(_:)`, remove `let` from the line that creates the new data task object:

```
dataTask = session.dataTask(with: url, completionHandler: {
```

You've removed the `let` keyword because `dataTask` should no longer be a local; it now refers to the instance variable.

- At the end of the method, add a question mark to the line that starts the task:

```
dataTask?.resume()
```

Because `dataTask` is an optional, you have to unwrap the optional somehow before you can use it. Here you're using optional chaining.

- Finally, near the top of the method before you set `isLoading` to `true`, add:

```
dataTask?.cancel()
```

If there is an active data task, this cancels it, making sure that no old searches can ever get in the way of the new search.

Thanks to the optional chaining, if no search has been done yet and `dataTask` is still `nil`, this simply ignores the call to `cancel()`. You could also unwrap the optional with `if let`, but using the question mark is shorter and just as safe.

Exercise: Why can't you write `dataTask!.cancel()` to unwrap the optional?

Answer: If an optional is `nil`, using `!` will crash the app. You're only supposed to use `!` to unwrap an optional when you're sure it won't be `nil`. But the very first time the user types something into the search bar, `dataTask` will still be `nil` and using `!` is not a good idea.

- Test the app with and without this call to `dataTask.cancel()` to experience the difference.

Use the Network Link Conditioner preferences pane to delay each query by a few seconds so it's easier to get two requests running at the same time.

Hmm... you may notice something odd. When the data task gets cancelled, you get the error pop-up and the Xcode Console says:

```
Failure! cancelled
2018-07-28 16:17:18.314825+0200 StoreSearch[21563:11484838] Task
<FD358C0B-EE6B-4B10-8336-970CF1C2192D>.<2> finished with error -
code: -999
```

As it turns out, when a data task gets cancelled, its completion handler is still invoked but with an `Error` object that has error code `-999`. That's what caused the error alert to pop up.

You'll have to make the error handler a little smarter to ignore code `-999`. After all, the user cancelling the previous search is no cause for panic.

- In the `completionHandler`, change the `if let error` section to:

```
if let error = error as NSError?, error.code == -999 {
    return // Search was cancelled
} else if let httpResponse = . . .
```

This simply ends the closure when there is an error with code `-999`. The rest of the closure gets skipped.

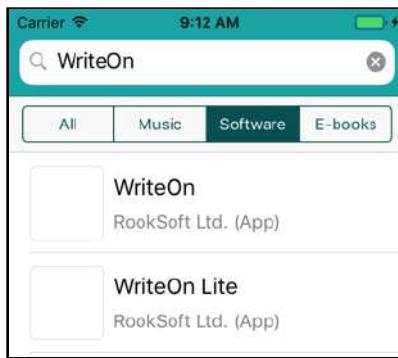
- If you're satisfied it works, commit the changes to the repository.

Note: Maybe you don't think it's worth making a commit when you've only changed a few lines, but many small commits are often better than a few big ones. Each time you fix a bug or add a new feature, it is a good time to commit.

Searching different categories

The iTunes store has a vast collection of products and each search returns at most 200 items. It can be hard to find what you're looking for by name alone. So, you'll add a control to the screen that lets users pick the category they want to search in.

It will look like this:



Searching in the Software category

This type of control is called a **segmented control** and is used to pick one option out of a set of choices.

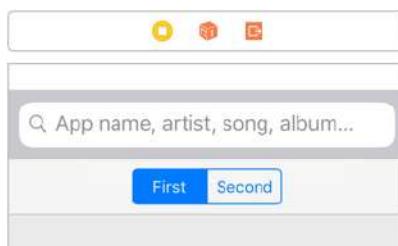
Adding the segmented control

► Open the storyboard. Drag a new **Navigation Bar** into the view and put it below the Search Bar. You're using the Navigation Bar purely for decorative purposes, as a container for the segmented control.

Make sure the Navigation Bar doesn't get added inside the Table View. It may be easiest to drag it from the Objects Library directly into the Document Outline and drop it below the Search Bar. Then change its Y-position to 76.

- With the Navigation Bar selected, open the **Add New Constraints menu** and pin its **top**, **left**, and **right** sides.
- Drag a new **Segmented Control** from the Objects Library on to the Navigation Bar's title to replace the title.

The design should now look like this:

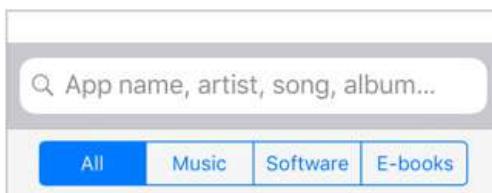


The Segmented Control sits in a Navigation Bar below the Search Bar

- Select the Segmented Control. Set its **Width** to 300 points (make sure you change the width of the entire control, not of the individual segments).
- In the **Attributes inspector**, set the number of segments to 4.
- Change the title of the first segment to **All**. Then select the second segment and set its title to **Music**. The title for the third segment should be **Software** and the fourth segment is **E-books**.

You can change the segment title by double-clicking inside the segment or by changing the **Segment** dropdown in the Attributes inspector to select the correct segment.

The scene should look like this now:



The finished Segmented Control

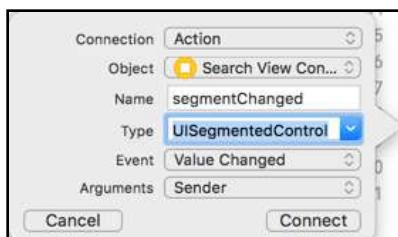
Next, you'll add a new outlet and action method for the Segmented Control. This is a good opportunity to practice using the Assistant editor.

Using the assistant editor

- Press **Option+⌘+Enter** to open the Assistant editor and then Control-drag from the Segmented Control into the view controller source code to add the new outlet:

```
@IBOutlet weak var segmentedControl: UISegmentedControl!
```

To add the action method you can also use the Assistant editor. Control-drag from the Segmented Control into the source code again, but this time choose:



Adding an action method for the segmented control

- Connection: **Action**
- Name: **segmentChanged**
- Type: **UISegmentedControl**
- Event: Value Changed
- Arguments: Sender

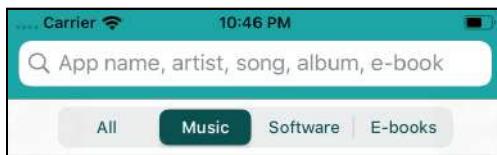
► Press **Connect** to add the action method. Then, add a `print()` statement to the new method:

```
@IBAction func segmentChanged(_ sender: UISegmentedControl) {  
    print("Segment changed: \(sender.selectedSegmentIndex)")  
}
```

Type **⌘+Enter** (without Option) to close the Assistant editor again. These are very handy keyboard shortcuts to remember.

► Run the app to make sure everything still works. Tapping a segment should log a number – the index of that segment – to the Console.

Your segmented control will look as follows:



The segmented control in action

Using the segmented control

Notice that the first row of the table view is partially obscured again. Because you placed a navigation bar below the search bar, you need to add another 44 points to the table view's content inset.

► Change that line in `viewDidLoad()` to:

```
tableView.contentInset = UIEdgeInsets(top: 108, left: 0, . . .)
```

and add this at the end of `viewDidLoad()`:

```
let segmentColor = UIColor(red: 10/255, green: 80/255, blue:  
80/255, alpha: 1)
```

```
let selectedTextAttributes =  
[NSAttributedString.Key.foregroundColor: UIColor.white]  
let normalTextAttributes =  
[NSAttributedString.Key.foregroundColor: segmentColor]  
segmentedControl.selectedSegmentTintColor = segmentColor  
  
segmentedControl.setTitleTextAttributes(normalTextAttributes,  
for: .normal)  
  
segmentedControl.setTitleTextAttributes(selectedTextAttributes,  
for: .selected)  
  
segmentedControl.setTitleTextAttributes(selectedTextAttributes,  
for: .highlighted)
```

Here you're setting the color of the segments in their normal, selected and highlighted state.

You will be using the segmented control in two ways. First of all, it determines what sort of products the app will search for. Second, if you have already performed a search and you tap on one of the other segment buttons, the app will search again for the new product category. That means a search can now be triggered by two different events: tapping the Search button on the keyboard and selecting an item in the Segmented Control.

► Rename the `searchBarSearchButtonClicked(_:)` method to `performSearch()` and remove the `searchBar` parameter.

You're doing this to put the search logic into a separate method that can be invoked from more than one place. Removing `searchBar` as the parameter of this method is no problem because there is also an `@IBOutlet` property with that name and any references to `searchBar` in `performSearch()` will simply use that property.

► Now add a new version of `searchBarSearchButtonClicked(_:)` to the source code:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
    performSearch()  
}
```

► Also replace the `segmentChanged(_:)` action method with:

```
@IBAction func segmentChanged(_ sender: UISegmentedControl) {  
    performSearch()  
}
```

- Run the app and verify that searching still works. When you tap on the different segments, the search should be performed again as well.

Note: The second time you search for the same thing, the app may return results very quickly. The networking layer is now returning a *cached* response so it doesn't have to download the whole thing again, which is usually a performance gain on mobile devices. However, there is an API to turn off this caching behavior if that makes sense for your app.

There is one thing left to be done — you have to tell the app to use the category based on the selected segment for the search. You've already seen that you can get the index of the selected segment with the `selectedSegmentIndex` property. This returns an `Int` value (0, 1, 2, or 3).

- Change the `iTunesURL(searchText:)` method so that it accepts this `Int` as a parameter and then builds up the request URL accordingly:

```
func iTunesURL(searchText: String, category: Int) -> URL {  
    let kind: String  
    switch category {  
        case 1: kind = "musicTrack"  
        case 2: kind = "software"  
        case 3: kind = "ebook"  
        default: kind = ""  
    }  
  
    let encodedText = searchText.addingPercentEncoding(  
        withAllowedCharacters: CharacterSet.urlQueryAllowed)!  
  
    let urlString = "https://itunes.apple.com/search?" +  
        "&term=\(encodedText)&limit=200&entity=\(kind)"  
  
    let url = URL(string: urlString)  
    return url!  
}
```

This first turns the category index from a number into a string, `kind`. Note that the category index is passed to the method as a new parameter.

Then it puts this string behind the `&entity=` parameter in the URL. For the “All” category, the entity value is empty, but for the other categories it is “`musicTrack`,” “`software`,” and “`ebook`,” respectively. Also note that instead of calling `String(format:)`, you now construct the URL string using string interpolation.

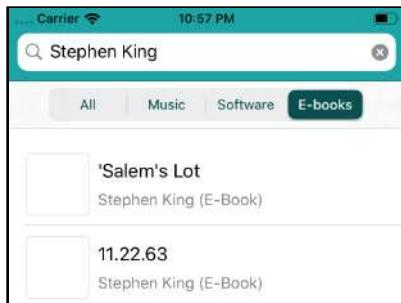
- In `performSearch()`, change the line that gets the URL to the following:

```
let url = iTunesURL(searchText: searchBar.text!,  
                     category: segmentedControl.selectedSegmentIndex)
```

And that should do it!

Note: You could have used `segmentedControl.selectedSegmentIndex` directly inside `iTunesURL` instead of passing the category index as a parameter. Using the parameter is the better design, though. It makes it possible to reuse the same method with a different type of control, should you decide that a Segmented Control isn't really the right component for this app. It is always a good idea to make methods as independent from each other as possible.

- Run the app and search for “stephen king.” In the All category that gives results for anything from songs to movies to podcasts to audio books. But if all you wanted were to get to his books, you can now use the E-Books category to finally find some of his novels.



You can now limit the search to just e-books

This finalizes the UI design of the main screen. This is as good a point as any to replace the empty white launch screen from the template.

Setting the launch screen

- Remove the `LaunchScreen.storyboard` file from the project.
- In the **Project Settings** screen, under **App Icons and Launch Images**, change **Launch Screen File** to `Main.storyboard`.

Now when the app starts, it uses the initial view controller from the storyboard as the launch image. Also verify that the app works properly on the iPad simulator and the larger iPhone models.

- Commit the changes and get ready for some more networking!

Downloading the artwork

The JSON search results contain a number of URLs to images and you put two of those — `imageSmall` and `imageLarge` — into the `SearchResult` object. Now you are going to download these images over the Internet and display them in the table view cells.

Downloading images, just like using a web service, is simply a matter of doing an HTTP request to a server that is connected to the Internet. An example of such a URL is:

<http://is4.mzstatic.com/image/thumb/Video1/v4/9c/d2/a5/9cd2a5e5-4710-abf4-925f-377e1666b0de/source/100x100bb.jpg>

Click that link and it will open the picture in a new web browser window. The server where this picture is stored is not `itunes.apple.com` but `is4.mzstatic.com`, but that doesn't matter at all to the app. As long as it has a valid URL, the app will just go fetch the file at that location, no matter where it is and what kind of file it is.

There are various ways that you can download files from the Internet. You're going to use `URLSession` and write a handy `UIImageView` extension to make this really convenient. Of course, you'll be downloading these images asynchronously!

SearchResultCell refactoring

First, you will move the logic for configuring the contents of the table view cells into the `SearchResultCell` class. That's a better place for it. Logic related to an object should live inside that object as much as possible, not somewhere else. Many developers have a tendency to stuff everything into their view controllers, but if you can move some of the logic into other objects, that makes for a much cleaner program.

- Add the following method to `SearchResultCell.swift`:

```
// MARK:- Public Methods
func configure(for result: SearchResult) {
```



```
nameLabel.text = result.name

if result.artist.isEmpty {
    artistNameLabel.text = "Unknown"
} else {
    artistNameLabel.text = String(format: "%@ (%@)",
                                    result.artist, result.type)
}
}
```

This is basically the same code as in `tableView(_:cellForRowAt:)`.

- Now, change `tableView(_:cellForRowAt:)` as follows:

```
func tableView(_ tableView: UITableView,
              cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if isLoading {
        .
        .
    } else if searchResults.count == 0 {
        .
        .
    } else {
        let searchResult = searchResults[indexPath.row]
        // Replace all code after this with new code below
        cell.configure(for: searchResult)
        return cell
    }
}
```

This small refactoring of moving some code from one class, `SearchViewController`, into another, `SearchResultCell`, was necessary to make the next bit work right.

In hindsight, it makes more sense to do this sort of thing in `SearchResultCell` anyway, but until now it did not really matter. Don't be afraid to refactor your code! Remember, if you screw up, you can always go back to your last Git commit.

- Run the app to make sure everything still works fine.

UIImageView extension for downloading images

OK, here comes the cool part. You will now add an extension for `UIImageView` that downloads the image and automatically displays it via the image view on the table view cell with just one line of code!



An extension can be used to extend the functionality of an existing class without having to subclass it. This works even for classes from system frameworks.

UIImageView doesn't have built-in support for downloading images, but this is a very common thing to do in apps. It's great that you can simply plug in your own extension and from then on every UIImageView in your app has this new ability.

► Add a new file to the project using the **Swift File** template, and name it **UIImageView+DownloadImage.swift**.

► Replace the contents of the new file with the following:

```
import UIKit

extension UIImageView {
    func loadImage(url: URL) -> URLSessionDownloadTask {
        let session = URLSession.shared
        // 1
        let downloadTask = session.downloadTask(with: url,
                                                completionHandler: { [weak self] url, response, error in
            // 2
            if error == nil, let url = url,
                let data = try? Data(contentsOf: url), // 3
                let image = UIImage(data: data) {
                    // 4
                    DispatchQueue.main.async {
                        if let weakSelf = self {
                            weakSelf.image = image
                        }
                    }
                }
            // 5
            downloadTask.resume()
        })
        return downloadTask
    }
}
```

This should look very similar to what you did before with URLSession, but there are some differences:

1. After obtaining a reference to the shared URLSession, you create a download task. This is similar to a data task, but it saves the downloaded file to a temporary location on disk instead of keeping it in memory.
2. Inside the completion handler for the download task, you're given a URL where you can find the downloaded file — this URL points to a local file rather than an internet address. Of course, you must also check that `error` is `nil` before you continue.

3. With this local URL you can load the file into a `Data` object and then create an image from that. It's possible that constructing the `UIImage` fails, for example, when what you downloaded was not a valid image but a 404 page or something else unexpected. As you can tell, when dealing with networking code, you need to check for errors every step of the way!
4. Once you have the image, you can put it into the `UIImageView`'s `image` property. Because this is UI code you need to do this on the main thread.

Here's the tricky thing: it is theoretically possible that the `UIImageView` no longer exists by the time the image arrives from the server. After all, it may take a few seconds and the user might have navigated away to a different part of the app by then.

That won't happen in this part of the app because the image view is part of a table view cell and they get recycled but not thrown away. But later on you'll use this same code to load an image on a screen that may be closed while the image file is still downloading. In that case, you don't want to set the image if the `UIImageView` is not visible anymore.

That's why the capture list for this closure includes `[weak self]`, where `self` now refers to the `UIImageView`. Inside the `DispatchQueue.main.async` you need to check whether "self" still exists; if not, then there is no more `UIImageView` to set the image on.

5. After creating the download task, you call `resume()` to start it, and then return the `URLSessionDownloadTask` object to the caller. Why return it? That gives the app the opportunity to call `cancel()` on the download task if necessary. You'll see how that works in a minute.

And that's all you need to do. From now on you can call `loadImage(url:)` on any `UIImageView` object in your project. Cool, huh?

Note: Swift lets you combine multiple `if let` statements into a single line, like you did above:

```
if error == nil, let url = ..., let data = ..., let image = ... {
```

There are three optionals being unwrapped here: 1) `url`, 2) the result from `Data(contentsOf:)`, and 3) the result from `UIImage(data:)`.

You can write this as three separate `if let` statements, and one for `if error == nil`, but having everything inside a single `if` statement is easier to read than many nested `if` statements spread over several lines.

Using the image downloader extension

- Switch to **SearchResultCell.swift** and add a new instance variable, `downloadTask`, to hold a reference to the image downloader:

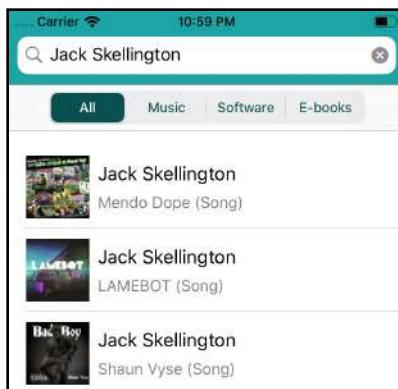
```
var downloadTask: URLSessionDownloadTask?
```

- Now, add the following lines to the end of `configure(for:)`:

```
artworkImageView.image = UIImage(named: "Placeholder")
if let smallURL = URL(string: result.imageSmall) {
    downloadTask = artworkImageView.loadImage(url: smallURL)
}
```

This tells the `UIImageView` to load the image from `imageSmall` and to place it in the cell's image view. While the real artwork is downloading, the image view displays a placeholder image — the same one from the nib for this cell.

- Run the app and enjoy your colourful images!



The app now downloads the album artwork

App transport security

While your image downloading experience worked brilliantly here, sometimes when dealing with image downloads, or accessing any web URL for that matter, you might see something like the following in the Xcode Console, alongwith a ton of error messages for failed download tasks:

```
App Transport Security has blocked a cleartext HTTP (http://)
resource load since it is insecure. Temporary exceptions can be
configured via your app's Info.plist file.
```

As of iOS 9, you can no longer download files over HTTP. Instead, you always need to use HTTPS.

As the error message indicates, you can add a key to the app's Info.plist to bypass this App Transport Security feature, allowing you to use plain `http://` URLs.

- Open `Info.plist` and add a new row. Give it the key `NSAppTransportSecurity`, or choose **App Transport Security Settings** from the list.
- Make sure the Type is a Dictionary.
- Add a new key inside that dictionary named `NSAllowsArbitraryLoads`, or choose **Allow Arbitrary Loads** from the list. Make this a Boolean and set it to YES.

▼ App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES

Overriding App Transport Security

That's all you need to do to access HTTP links. However, you're only supposed to bypass App Transport Security if there is absolutely no way you can make the app work over HTTPS. If you're making an app that talks to a server you control, then the best thing to do is to enable HTTPS on the server, not disable HTTPS in the app.

The `Info.plist` setting is only intended for when you need to communicate with other people's servers that do not support HTTPS. Obviously, in that case, the app should not send sensitive data to those servers! Unprotected HTTP should only be used for downloading publicly accessible data, such as images.

When you set the key `NSAllowsArbitraryLoads` to YES, the app can use *any* URL that starts with `http://`, regardless of the domain. To allow HTTP on specific domains only, set `NSAllowsArbitraryLoads` to NO and add a new dictionary named `NSEExceptionDomains`. Under that dictionary, you can add a new dictionary for each domain.

For example, the iTunes web service appears to host all its preview images on the website `mzstatic.com`. You could configure `Info.plist` as follows:

▼ App Transport Security Settings	Dictionary	(2 items)
Allow Arbitrary Loads	Boolean	NO
▼ Exception Domains	Dictionary	(1 item)
▼ mzstatic.com	Dictionary	(2 items)
NSEExceptionAllowsInsecureHTTPLoads	Boolean	YES
NSIncludesSubdomains	Boolean	YES

Now the app only allows `http://` requests from `mzstatic.com` and any of its subdomains, but requires `https://` URLs for any other domains.



Note that Apple has indicated that this ability to bypass App Transport Security (ATS) will be removed at some time in the future. So do not rely on the ATS-bypass being something which would always be available.

Cancelling previous image downloads

These images already look pretty sweet, but you're not quite done yet. Remember that table view cells can be reused, so it's theoretically possible that you're scrolling through the table and some cell is about to be reused while its previous image is still downloading.

You no longer need that image, so you should really cancel the pending download. Table view cells have a special method named `prepareForReuse()` that is ideal for this.

► Add the following method to `SearchResultCell.swift`:

```
override func prepareForReuse() {
    super.prepareForReuse()
    downloadTask?.cancel()
    downloadTask = nil
}
```

You simply cancel any image download that is still in progress.

Exercise: Put a `print()` in the `prepareForReuse()` method and see if you can trigger it.

On a decent Wi-Fi connection, loading the images is very fast. You almost cannot see it happen, even if you scroll quickly. It also helps that the image files are small — only 60 by 60 pixels — and that the iTunes servers are fast.

That is key to having a snappy app: don't download more data than you need.

Caching

Depending on what you searched for, you may have noticed that many of the images were the same. For example, you might get many identical album covers in the search results. `URLSession` is smart enough not to download identical images — or at least images with identical URLs — twice. That principle is called *caching* and it's very important on mobile devices.

Mobile developers are always trying to optimize their apps to do as little as possible.



If you can download something once and then use it over and over, that's a lot more efficient than re-downloading it all the time.

Images aren't the only things that you can cache. You can also cache the results of big computations, for example. Or views, as you have been doing in the previous apps, probably without even realizing it. When you use the principle of lazy loading, you delay the creation of an object until you need it and then you cache it for the next time.

Cached data does not stick around forever. When your app gets a memory warning, it's a good idea to remove any cached data that you don't need right away. That means you will have to reload that data when you need it again later, but that's the price you have to pay. For URLSession this is completely automatic, so that takes another burden off your shoulders.

Some caches are in-memory — the cached data only stays in the computer's working memory. But it is also possible to cache the data to the disk. Your app even has a special directory for it, Library/Caches.

The caching policy used by *StoreSearch* is very simple — it uses the default settings. But you can configure URLSession to be much more advanced. Look into the documentation for URLCache and URLSessionConfiguration to learn more.

Merge the branch

This concludes the section on talking to the web service and downloading images. Later on, you'll tweak the web service requests a bit more to include the user's language and country, but for now, you're done with this feature. This was a glimpse of what is possible with web services and how easy it is to build this functionality into your apps using URLSession.

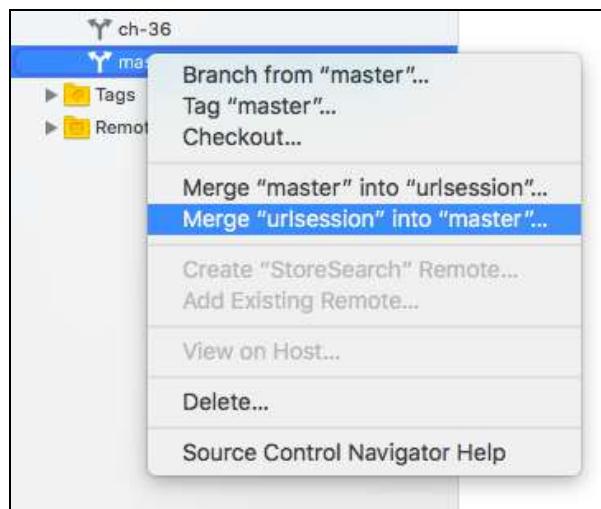
- Commit these latest changes to the repository.

Merge the branch using Xcode

Now that you've completed a feature, you can merge this temporary branch back into the master branch.

- Switch to the **Source Control navigator**, select the **master** branch — or whatever was your main branch previously — under branches, and right-click to get the context menu of actions. Select **Merge "urlsession" into "master"...**:





Merging your changes back to the master branch

- You'll get a confirmation dialog. Click **Merge** if you want to continue.



The confirmation dialog before merging changes

Now that the master branch is up-to-date with the networking changes, if you wanted to, you could remove the “urlsession” branch. Or, you could keep it and do more work on it later.

Merge the branch from the command line

The source control features in Xcode used to be a bit rough around the edges. So, it was possible that certain commands, especially merging changes, might not work correctly. If Xcode didn't want to cooperate when you tried to merge changes, here is how you'd do it from the command line.

- First close Xcode. You don't want to do any of this while Xcode still has the project open. That's just asking for trouble.

- Open a Terminal, cd to the *StoreSearch* folder, and type the following commands:

```
git stash
```

This moves any unsaved files out of the way — no, it doesn’t have anything to do with facial hair... This saves any uncommitted changes so you can later restore them, if need be.

```
git checkout master
```

This switches the current branch back to the master branch.

```
git merge urlsession
```

This merges the changes from the “urlsession” branch back into the master branch. If you get an error message at this point, then simply do `git stash` again and repeat the `git merge` command. By the way, you don’t really need to keep those stashed files around, so if you want to remove them from your repository, you can do `git stash drop`. If you stashed twice, you also need to drop twice.

► Open the project again in Xcode. Now you’re back at the master branch and it also has the latest networking changes.

► Build and run to see if everything still works.

Git is a pretty awesome tool, but it takes a while to get familiar with it. Xcode’s Git support has improved a lot since Xcode 9, but for more complex things you might still need to use the command line — it’s well worth learning!

Note: Even though URLSession is pretty easy to use and quite capable, many developers prefer to use third-party networking libraries that are often even more convenient and powerful. One of the most popular native Swift libraries at this point is Alamofire (github.com/Alamofire).

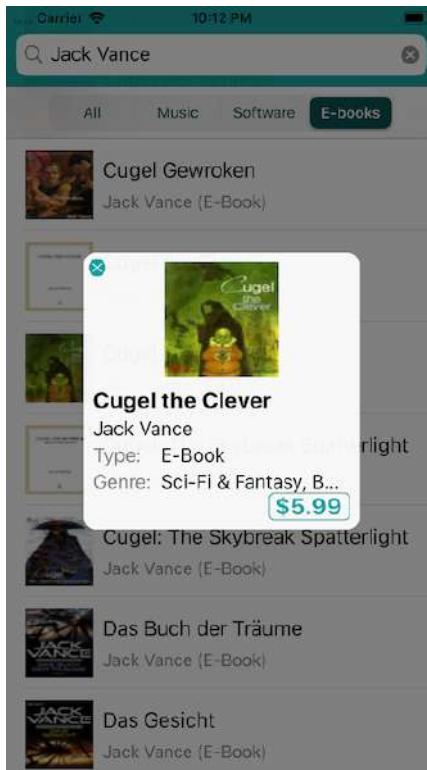
You should check out some of these libraries and see how you like them. Networking is such an important feature of mobile apps that it’s worth being familiar with the different possible approaches to send data up and down the ‘net.

You can find the project files for this chapter under **41 – URLSession** in the Source Code folder.

Chapter 42: The Detail Pop-Up

Eli Ganim

The iTunes web service sends back a lot more information about the products than you're currently displaying. Let's add a "details" screen to the app that pops up when the user taps a row in the table:



The app shows a pop-up when you tap a search result



raywenderlich.com

1168

The table and search bar are still visible in the background, but they have been darkened.

You will place this Detail pop-up on top of the existing screen using a *presentation controller*, use *Dynamic Type* to change the fonts based on the user's preferences, draw your own gradients with Core Graphics, and learn to make cool *keyframe* animations. Fun times ahead!

This chapter will cover the following:

- **The new view controller:** Create the bare minimum necessary for the new Detail pop-up and add the code to show/hide the pop-up.
- **Add the rest of the controls:** Complete the design for the Detail pop-up.
- **Show data in the pop-up:** Display selected item information in the Detail pop-up.

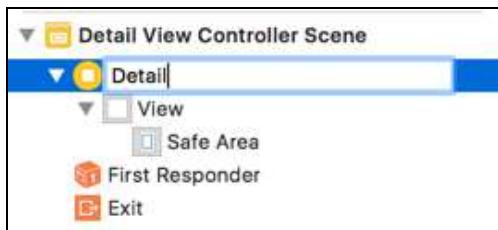
The new view controller

A new screen means a new view controller, so let's start with that.

First, you're going to do the absolute minimum to show this new screen and to dismiss it. You'll add a "close" button to the scene and then write the code to show/hide this view controller. Once that works, you will put in the rest of the controls.

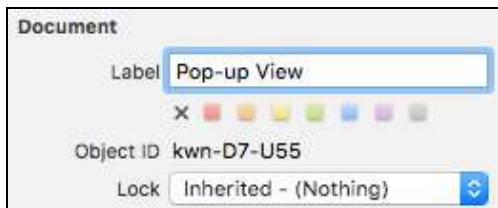
The basic view controller

- Add a new **Cocoa Touch Class** file to the project. Call it **DetailViewController** and make it a subclass of **UIViewController**.
- Open the storyboard and drag a new **View Controller** on to the canvas. Change its **Class** to **DetailViewController** — via the **Identify inspector** tab.
- For ease of reference, change the new scene's name from **Detail View Controller** to **Detail** by clicking on the yellow circle for the view controller on the Document Outline and clicking again to be able to edit the name.



Editing the scene name to give it a simpler name

- Similarly, rename the previously added **Search View Controller** scene to **Search**.
- Set the **Background** color of the Detail scene's view to **black, 50% opaque**. That makes it easier to see what is going on in the next steps.
- Drag a new **View** into the scene. Using the **Size inspector**, make it **240** points wide and **240** high. Add Auto Layout constraints for width and height to ensure that the view statys at this size.
- Center the view in the scene by setting up horizontal and vertical centering Auto Layout constraints.
- In the **Attributes inspector**, change the **Background** color of this new view to **Secondary System Background Color, 95% opaque**. This makes it appear slightly translucent, just like navigation bars.
- With this new view still selected, go to the **Identity inspector**. For **Document - Label** — the field with the hint text of “Xcode Specific Label” — type **Pop-up View**. You can use this field to give your views names, so they are easier to distinguish in the Document Outline in Interface Builder. Now, instead of having multiple items called "View", this particular view will display as "Pop-up View".

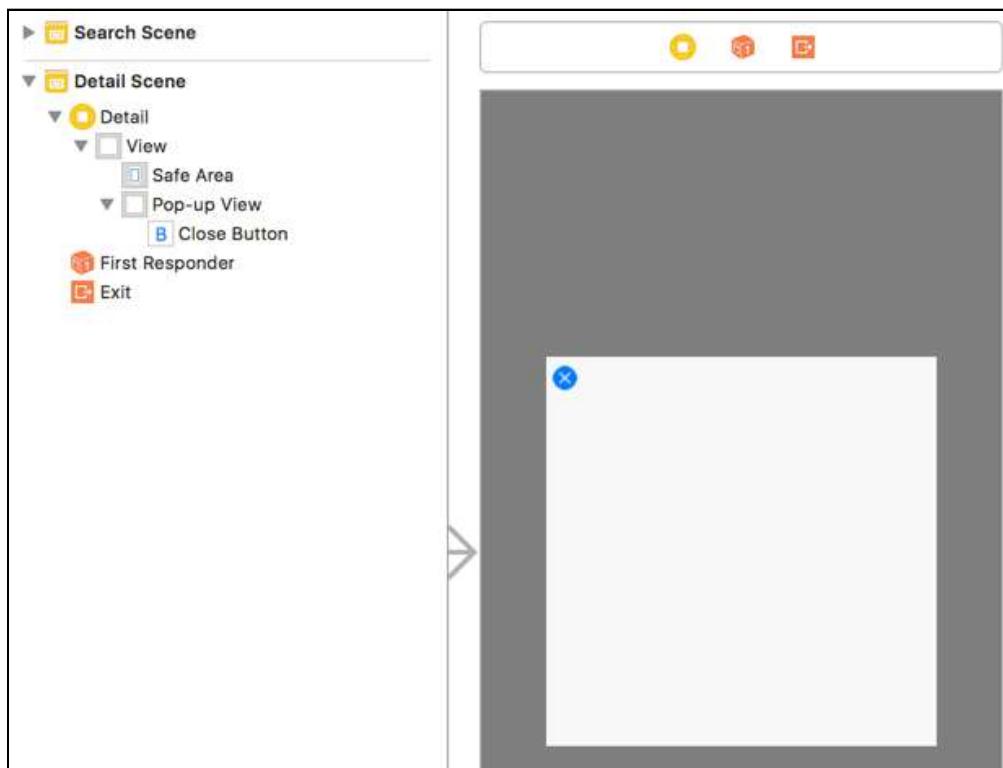


Giving the view a description for use in Xcode

- Drag a **Button** into the scene and place it somewhere on the Pop-up View. In the **Attributes inspector**, change **Image** to **CloseButton** — you already added this image to the asset catalog earlier.

- Remove the button's text. Choose **Editor** ▶ **Size to Fit Content** to resize the button and place it in the top-left corner of the Pop-up View, at X = 4 and Y = 2.
- If the button's **Type** now says **Custom**, change it back to **System**. That will make the image turn blue, because the default tint color is blue.
- Set the Xcode Specific Label for the Button to **Close Button**. Remember that this only changes the title displayed in the Interface Builder; the user will never see that text.

The design should look something like this:



The Detail scene has a white square and a close button on a dark background

Note: Xcode currently gives a warning that this new scene is unreachable. This warning will disappear after you make a segue to it, which you'll do in a second.

Showing and hiding the scene

Let's write the code to show and hide this new screen.

- In **DetailViewController.swift**, add the following action method:

```
// MARK:- Actions
@IBAction func close() {
    dismiss(animated: true, completion: nil)
}
```

- Connect this action method to the **X** button's Touch Up Inside event in Interface Builder — as before, Control-drag from the button to the view controller and pick from Sent Events.
- Control-drag from the yellow circle at the top of the Search scene to the Detail scene to make a **Present Modally** segue. Give it the identifier **ShowDetail**.

Because the table view doesn't use prototype cells, you have to put the segue on the view controller itself. That means you need to trigger the segue manually when the user taps a row.

- Open **SearchViewController.swift** and change `didSelectRowAt` to the following:

```
func tableView(_ tableView: UITableView,
              didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
    // Add the following line
    performSegue(withIdentifier: "ShowDetail", sender: indexPath)
}
```

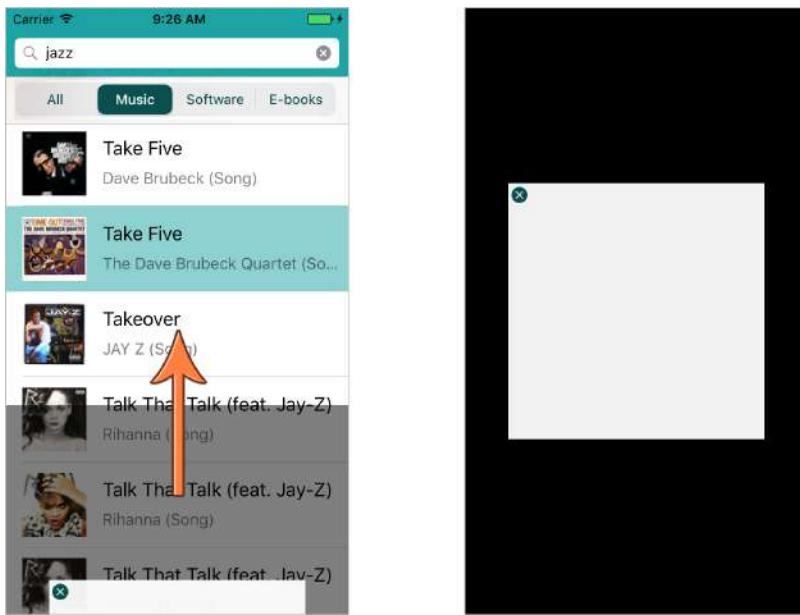
You're sending along the index-path of the selected row as the `sender` parameter. This will come in useful later when you're putting the `SearchResult` object into the Detail pop-up.

Let's see how well this works.

- Run the app, do a search, and tap on a search result. Hmm, that doesn't look too good.

Even though you set the main view to be half transparent, the Detail screen still has a solid black background. Only during the animation is it see-through.





What happens when you present the Detail screen modally

Hmm, presenting this new screen with a regular modal segue isn't going to achieve the effect we're after.

There are three possible solutions:

1. Don't have a `DetailViewController`. You can load the view for the detail pop-up from a nib and add it as a subview of `SearchViewController`, and put all the logic for this screen in `SearchViewController`. This is not a very good solution because it makes `SearchViewController` more complex — the logic for a new screen should really go into its own view controller.
2. Use the `view controller containment` APIs to embed the `DetailViewController` “inside” the `SearchViewController`. This is a better solution but it's still more work than necessary — you'll see an example of view controller containment in an upcoming chapter where you'll be adding a special landscape mode to the app.
3. Use a `presentation controller`. This lets you customize how modal segues present their view controllers on the screen. You can even have custom animations to show and hide the view controllers.

Let's go for #3. Transitioning from one screen to another in an iOS app involves a complex web of objects that take care of all the details concerning presentations, transitions, and animations. Normally, that all happens behind the scenes and you can safely ignore it.

But if you want to customize how some of this works, you'll have to dive into the excitingly strange world of presentation controllers and transitioning delegates.

Custom presentation controller

- Add a new Swift File to the project, named **DimmingPresentationController**.
- Replace the contents of this new file with the following:

```
import UIKit

class DimmingPresentationController: UIPresentationController {
    override var shouldRemovePresentersView: Bool {
        return false
    }
}
```

The standard `UIPresentationController` class contains all the logic for presenting new view controllers. You're providing your own version that overrides some of this behavior — in particular, telling `UIKit` to leave the `SearchViewController` visible. That's necessary to get the see-through effect.

Later you'll also add a light-to-dark gradient background view to this presentation controller; that's where the “dimming” in its name comes from.

Note: It's called a presentation controller, but it is not a *view* controller. The use of the word *controller* may be a bit confusing here but not all controllers are for managing screens in your app — generally, only those with “view” in their name do that.

A presentation controller is an object that “controls” the presentation of something, just like a view controller is an object that controls a view and everything in it. Soon you'll also see an animation controller, which controls — you guessed it — an animation.

There are quite a few different kinds of controller objects in the various iOS frameworks. Just remember that there's a difference between a view controller and other types of controllers.

Now you need to tell the app that you want to use your own presentation controller to show the Detail pop-up.

- In **DetailViewController.swift**, add the following extension to the end of the file:

```
extension DetailViewController: UIViewControllerTransitioningDelegate {  
    func presentationController(  
        forPresented presented: UIViewController,  
        presenting: UIViewController?, source: UIViewController) ->  
        UIPresentationController? {  
            return DimmingPresentationController(  
                presentedViewController: presented,  
                presenting: presenting)  
        }  
}
```

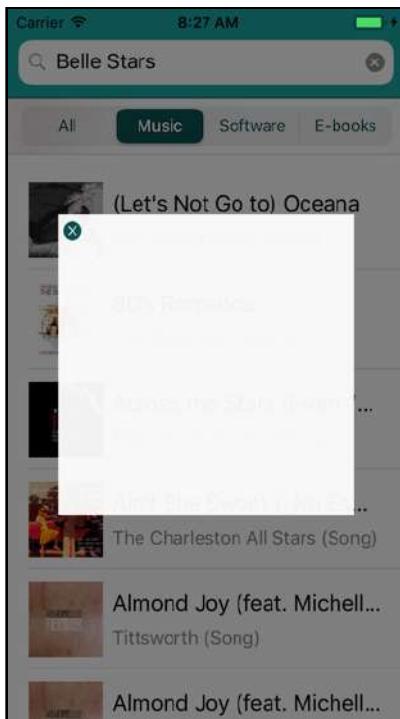
The methods from this delegate protocol tell UIKit what objects it should use to perform the transition to the Detail View Controller. It will now use your new **DimmingPresentationController** class instead of the standard presentation controller.

- Also add the following init method to **DetailViewController**:

```
required init?(coder aDecoder: NSCoder) {  
    super.init(coder: aDecoder)  
    modalPresentationStyle = .custom  
    transitioningDelegate = self  
}
```

Recall that **init?(coder)** is invoked to load the view controller from the storyboard. Here you tell UIKit that this view controller uses a custom presentation and you set the delegate that will call the method you just implemented.

- Run the app again and tap a row to bring up the detail pop-up. That looks much better! Now the list of search results remains visible.



The Detail pop-up background is now see-through

The standard presentation controller removed the underlying view from the screen, making it appear as if the Detail pop-up had a solid black background. Removing the view makes sense most of the time when you present a modal screen, as the user won't be able to see the previous screen anyway. Plus, not having to redraw this view saves battery life too.

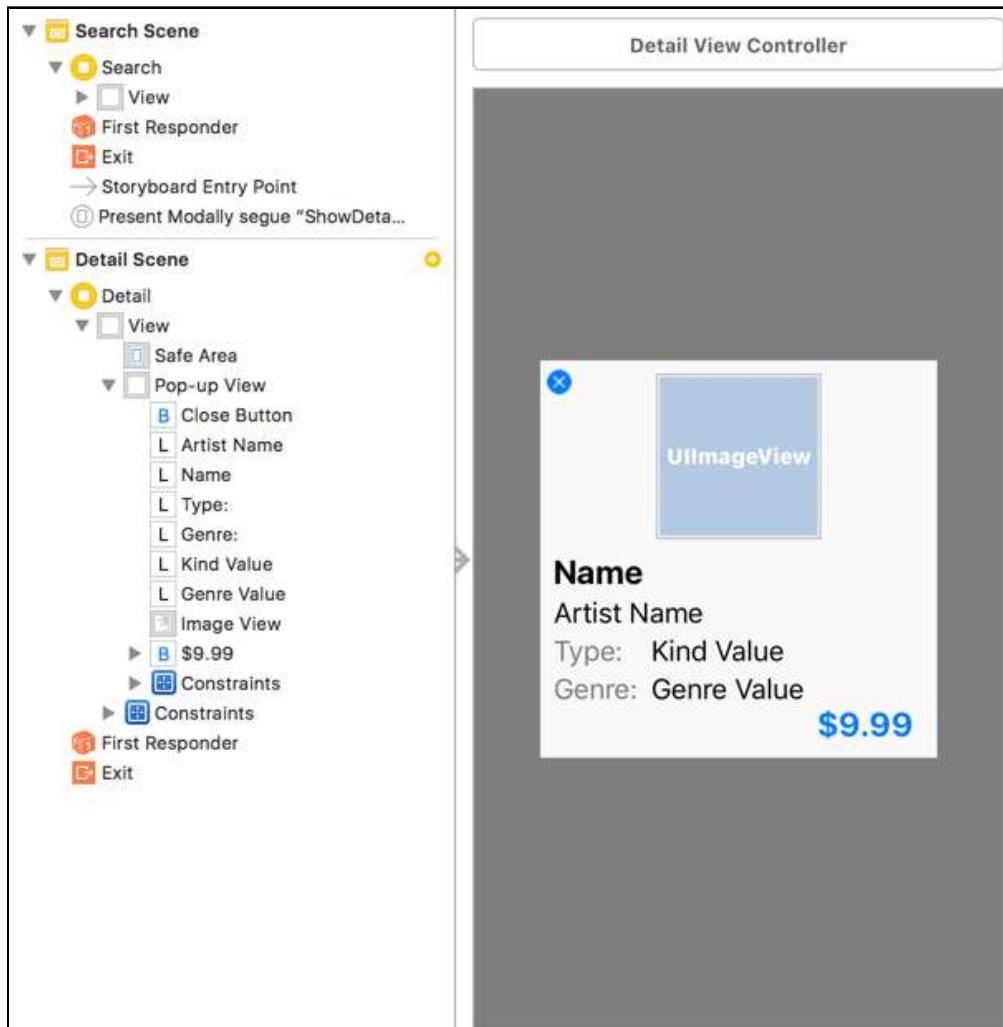
However, in your case, the modal segue leads to a view controller that only partially covers the previous screen. You want to keep the underlying view to get the see-through effect. That's why you needed to supply your own presentation controller object.

- Also verify that the close button works to dismiss the pop-up.

Adding the rest of the controls

Let's finish the design of the Detail screen. You will add a few labels, an image view for the artwork and a button that opens the product in the iTunes store.

The finished design will look like this:



The Detail screen with the rest of the controls

Adding the controls

- Drag a new **Image View**, six **Labels**, and a **Button** on to the pop-up view and build a layout like the one from the picture.

Some suggestions for the dimensions and positions:

Control	X	Y	Width	Height
Image View	70	8	100	100
Name label	8	116	220	24
Artist Name label	8	142	220	21
Type: label	8	165	43	21
Kind Value label	67	165	160	21
Genre: label	8	188	51	21
Genre Value label	67	188	160	21
\$9.99 button	164	208	68	24

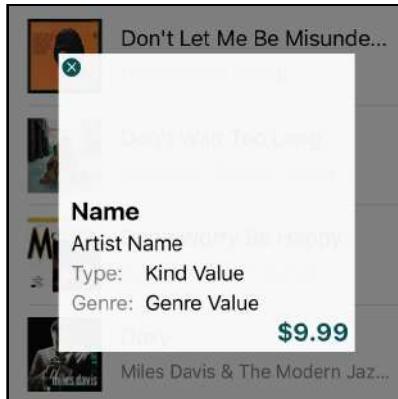
- The **Name** label's font is **System Bold 20**. Set **Autoshrink** to **Minimum Font Scale** so the font can become smaller if necessary to fit as much text as possible.
- The font for the **\$9.99** button is also **System Bold 20**. You will add a background image for this button in a bit.
- You shouldn't have to change the font for the other labels; they use the default value of System 17.
- Set the **Color** for the **Type:** and **Genre:** labels to 50% opaque black.

These new controls are pretty useless without outlet properties, so add the following lines to **DetailViewController.swift**:

```
@IBOutlet weak var popupView: UIView!
@IBOutlet weak var artworkImageView: UIImageView!
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var artistNameLabel: UILabel!
@IBOutlet weak var kindLabel: UILabel!
@IBOutlet weak var genreLabel: UILabel!
@IBOutlet weak var priceButton: UIButton!
```

- Connect the outlets to the views in the storyboard. Control-drag from Detail View Controller to each of the views and pick the corresponding outlet. The Type: and Genre: labels and the X button do not get an outlet.

- Run the app to see if everything still works.



The new controls in the Detail pop-up

Did you notice something interesting here? You didn't add any Auto Layout constraints to the contents of the Pop-up View and yet, the controls you added stay in place fine no matter which size screen you run the app on. Try it by running the app on several different simulators.

Can you guess why?

The reason is that the Pop-up View itself is constrained to 240 x 240 points in size. So the contents of the view do not shift around — or change size — when the screen size changes. Because of this, you can rely on your original positioning of your controls to serve you here without any issues.

However, if the parent view — Pop-up View, in this case — were to change size, then you would need to set up Auto Layout constraints for the child controls if you wanted them to size correctly as the parent's size changed.

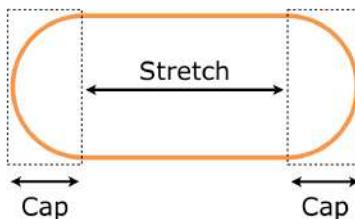
In the meantime, you will get Xcode warnings about the controls inside the pop-up view not having Auto Layout constraints. Generally, whether you add Auto Layout constraints in such situations, or disregard these warnings, is totally up to you.

Note: For this particular app, let the Xcode warnings stay around for the moment. We will be adding Auto Layout constraints in the next chapter — there's a reason for the delay.

Stretchable images

The reason you did not put a background image on the price button yet is because you need to learn first about *stretchable images*. When you put a background image on a button in Interface Builder, it always has to fit the button exactly. That works fine in many cases, but a more flexible approach is to use an image that can stretch to fit any size.

When an image view is wider than the image, it will automatically stretch the image to fit. In the case of a button, however, you don't want to stretch the ends (or "caps") of the button, only the middle part. That's what a stretchable image lets you do.



The caps are not stretched but the inner part of the image is

For *Bull's Eye* you used `resizableImage(withCapInsets:)` to cut the images for the slider track into stretchable parts. You can also do this in the asset catalog without having to write any code.

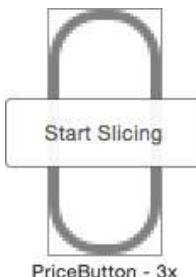
- Open **Assets.xcassets** and select the **PriceButton** image set.



The PriceButton image

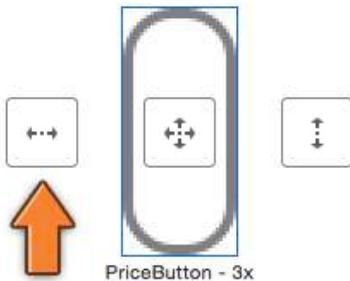
If you take a look at the image info, you will see that it is only 11 points wide. That means it has a 5-point cap on the left, a 5-point cap on the right, and a 1-point body that will be stretched out.

Click the **Show Slicing** button at the bottom of the central panel.



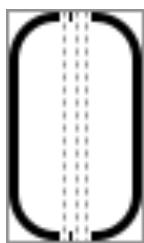
The Start Slicing button

Now all you have to do is click **Start Slicing** on each of the two images, followed by the **Slice Horizontally** button:



The Slice Horizontally button

You should end up with something like this for each of the button sizes:



After slicing

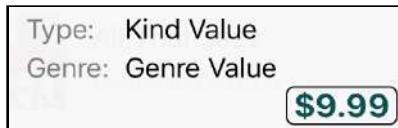
Each image is cut into three parts: the caps on the end and a one-pixel area in the middle that is the stretchable part. Now when you use this image with a button or a UIImageView, it will automatically stretch itself to whatever size it needs to be.

Important: Do the above for *both* the 2x image and the 3x image.

- Go back to the storyboard. For the \$9.99 button, change **Background** to **PriceButton**.

If you see the image repeating, make sure that the button is only 24 points high, the same as the image height.

- Run the app and check out that button. Here's a close-up of what it looks like:



The price button with the stretchable background image

The main reason you're using a stretchable image here is that the text on the button may vary in size depending on the price of the item. So, you don't know in advance how big the button needs to be. If your app has a lot of custom buttons, it's worth making their images stretchable. That way you won't have to re-do the images whenever you're tweaking the sizes of the buttons.

The button could still look a little better, though — a black frame around dark green text doesn't particularly please the eye. You could go into Photoshop and change the color of the image to match the text color, but there's an easier method.

The tint color

The color of the button text comes from the global tint color. `UIImage` makes it very easy to make images appear in the same tint color.

- In the asset catalog, select the **PriceButton** set again and go to the **Attribute inspector**. Change **Render As** to **Template Image**.

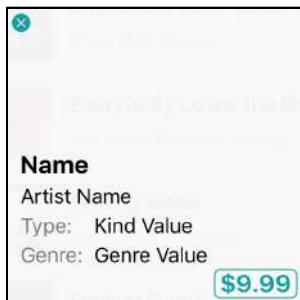
When you set the “template” rendering mode on an image, `UIKit` removes the original colors from the image and paints the whole thing in the tint color.

The dark green tint color looks nice in the rest of the app, but for this pop-up it's a bit too dark. You can change the tint color on a per-view basis; if that view has subviews the new tint color also applies to these subviews.

- In `DetailViewController.swift`, add the following line to `viewDidLoad()`:

```
view.tintColor = UIColor(red: 20/255, green: 160/255,  
blue: 160/255, alpha: 1)
```

Note that you're setting the new `tintColor` on `view`, not just on `priceButton`. That will apply the lighter tint color to the close button as well:



The buttons appear in the new tint color

Much better, but there is still more to tweak. In the screenshot at the start of this section the pop-up view had rounded corners. You could use an image to make it look like that, but instead you'll learn a neat little trick.

Rounded corner views

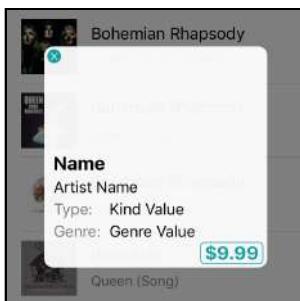
UIViews do their drawing using what's known as a `CALayer` object. The `CA` prefix stands for Core Animation, which is the awesome framework that makes animations so easy on the iPhone. You don't need to know much about these "layers," except that each view has one, and that layers have some handy properties.

- Add the following line to `viewDidLoad()`:

```
popupView.layer.cornerRadius = 10
```

You ask the Pop-up View for its layer and then set the corner radius of that layer to 10 points. And that's all you need to do!

- Run the app. You have your rounded corners:



The pop-up now has rounded corners

Tap gesture recognizer

The close button is pretty small, about 15 by 15 points. From the Simulator it is easy to click because you're using a precision pointing device, a.k.a. the mouse. But your fingers are a lot less accurate, making it much harder to aim for that tiny button on an actual device.

That's one reason why you should always test your apps on real devices and not just on the Simulator. Apple recommends that buttons always have a tap area of at least 44×44 points.

To make the app more user-friendly, you'll also allow users to dismiss the pop-up by tapping anywhere outside it. The ideal tool for this job is a **gesture recognizer**.

► Add a new extension to **DetailViewController.swift**:

```
extension DetailViewController: UIGestureRecognizerDelegate {  
    func gestureRecognizer(  
        _ gestureRecognizer: UIGestureRecognizer,  
        shouldReceive touch: UITouch) -> Bool {  
        return (touch.view === self.view)  
    }  
}
```

You only want to close the Detail screen when the user taps outside the pop-up, i.e. on the background. Any other taps should be ignored. That's what this delegate method is for. It only returns `true` when the touch was on the background view — it will return `false` if the touch was inside the Pop-up View.

Note that you're using the identity operator `==` to compare `touch.view` with `self.view`. You want to know whether both variables refer to the same object. This is different from using the `==` equality operator. That would check whether both variables refer to objects that are considered equal, even if they aren't the same object.

Using `==` here would have worked too, but only because `UIView` treats `==` and `==` the same. But not all objects do, so be careful!

► Add the following lines to `viewDidLoad()`:

```
let gestureRecognizer = UITapGestureRecognizer(target: self,  
                                            action: #selector(close))  
gestureRecognizer.cancelsTouchesInView = false  
gestureRecognizer.delegate = self  
view.addGestureRecognizer(gestureRecognizer)
```



This creates a new gesture recognizer that listens to taps anywhere in this view controller and calls the `close()` method in response.

- Try it out. You can now dismiss the pop-up by tapping anywhere outside the white pop-up area. That's a common thing that users expect to be able to do, and it was easy enough to add to the app!

Showing data in the pop-up

Now that the app can show this pop-up after a tap on a search result, you should put the name, genre and price from the selected product in the pop-up.

Exercise: Try to do this by yourself. It's not very different from what you've done in the previous apps!

There is more than one way to pull this off. One common way is to pass the `SearchResult` object to the `DetailViewController`.

Displaying selected item information in pop-up

- Add a property to `DetailViewController.swift` to store the passed in object reference:

```
var searchResult: SearchResult!
```

As usual, this is an implicitly-unwrapped optional because you won't know what its value will be until the segue is performed. It is `nil` in the mean time.

- Also add a new method, `updateUI()`:

```
// MARK:- Helper Methods
func updateUI() {
    nameLabel.text = searchResult.name

    if searchResult.artist.isEmpty {
        artistNameLabel.text = "Unknown"
    } else {
        artistNameLabel.text = searchResult.artist
    }
    kindLabel.text = searchResult.type
    genreLabel.text = searchResult.genre
}
```

That looks very similar to what you did in `SearchResultCell`. The logic for setting the text on the labels has its own method, `updateUI()`, because that is cleaner than stuffing everything into `viewDidLoad()`.

- Add a call to the new method to the end of `viewDidLoad()`:

```
override func viewDidLoad() {  
    . . .  
    if searchResult != nil {  
        updateUI()  
    }  
}
```

The `if != nil` check is a defensive measure, just in case the developer forgets to fill in `searchResult` on the segue.

Note: You can also write the above check as `if let _ = searchResult` to unwrap the optional. Because you're not actually using the unwrapped value for anything, you use the `_` wildcard symbol.

The Detail pop-up is launched with a segue triggered from `SearchViewController`'s `tableView(_:didSelectRowAt:)`. You'll have to add a `prepare(for:sender:)` method to configure the `DetailViewController` when the segue happens.

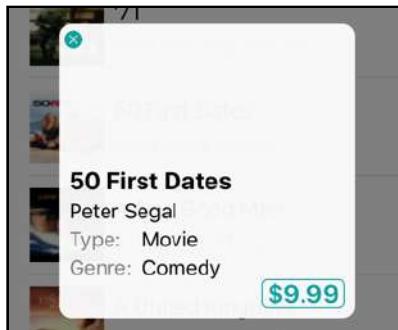
- Add this method to `SearchViewController.swift`:

```
// MARK:- Navigation  
override func prepare(for segue: UIStoryboardSegue,  
                      sender: Any?) {  
    if segue.identifier == "ShowDetail" {  
        let detailViewController = segue.destination  
                               as! DetailViewController  
        let indexPath = sender as! IndexPath  
        let searchResult = searchResults[indexPath.row]  
        detailViewController.searchResult = searchResult  
    }  
}
```

This should hold no big surprises for you. When `didSelectRowAt` starts the segue, it sends along the index-path of the selected row.

That lets you find the `SearchResult` object and pass it on to `DetailViewController`.

- Try it out. All right, now you're getting somewhere!



The pop-up with filled-in data

Showing the price

You still need to show the price for the item and the correct currency.

- Add the following code to the end of `updateUI()` in `DetailViewController.swift`:

```
// Show price
let formatter = NumberFormatter()
formatter.numberStyle = .currency
formatter.currencyCode = searchResult.currency

let priceText: String
if searchResult.price == 0 {
    priceText = "Free"
} else if let text = formatter.string(
    from: searchResult.price as NSNumber) {
    priceText = text
} else {
    priceText = ""
}

priceButton.setTitle(priceText, for: .normal)
```

You've used `NSDateFormatter` previously to turn a `Date` object into human-readable text. Here you use `NumberFormatter` to do the same thing for numbers.

Previously, you've turned numbers into text using string interpolation `\(...)` and `String(format:)` with the `%f` or `%d` format specifier. However, in this case you're not dealing with regular numbers but with money in a certain currency.

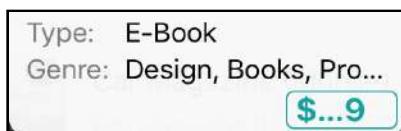
There are different rules for displaying various currencies, especially if you take the user's language and country settings into consideration. You could program all of these rules yourself — which is a lot of effort — or, choose to ignore them. Fortunately, you don't have to make that tradeoff because you have `NumberFormatter` to do all the heavy lifting for you.

You simply tell the `NumberFormatter` that you want to display a currency value and what the currency code is. That currency code comes from the web service and is something like "USD" or "EUR." `NumberFormatter` will insert the proper symbol, such as \$ or € or ¥, and format the monetary amount according to the user's regional settings.

There's one caveat: if you're not feeding `NumberFormatter` an actual number, it cannot do the conversion. That's why `string(from:)` returns an optional that you need to unwrap.

- Run the app and see if you can find some good deals.

Sometimes, you might see this, or something similar:



The price doesn't fit into the button

When you designed the storyboard, you made this button 68 points wide. You didn't put any constraints on it, so Xcode gave it an automatic constraint that always forces the button to be 68 points wide, no more, no less.

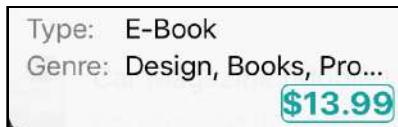
But buttons, like labels, are perfectly able to determine what their ideal size is based on the amount of text they contain. That's called their *intrinsic content size*.

- Open the storyboard and with the price button selected, click the **Add New Constraints** button. Add spacing constraints for the **right** and the **bottom**, both 8 points in size. Also add a **24** point **Height** constraint.

To recap, you have set the following constraints on the button:

- Fixed height of 24 points. That is necessary because the background image is 24 points tall.
- Pinned to the right edge of the pop-up with a distance of 8 points. When the button needs to grow to accommodate larger prices, it will extend towards the left.
- Pinned to the bottom of the pop-up, also with a distance of 8 points.

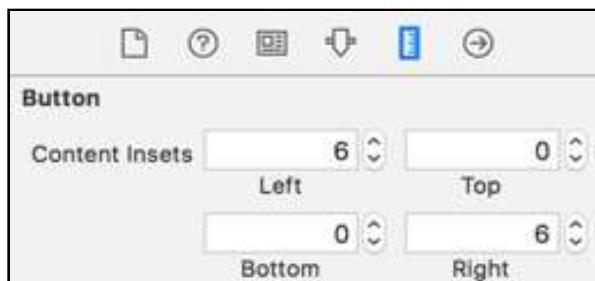
- There is no constraint for the width. That means the button will use its intrinsic width — the larger the text, the wider the button. And that's exactly what you want to happen here.► Run the app again and pick an expensive product — something with a price over \$9.99; e-books are a good category for this.



The button is a little cramped

That's better, but the text is now right up against the button border. You can fix this by setting "content edge insets" for the button.

- Go to the **Size inspector** and find where it says **Content Insets**. Change **Left** and **Right** to 6.



Changing the content edge insets of the button

This adds 6 points of padding on the left and right sides of the button.

- Run the app; now the price button should finally look good:



That price button looks so good you almost want to tap it!

Note: After you added spacing constraints for the price button, you might have noticed that you started getting an additional Xcode warning saying "Leading constraint is missing, which may cause overlapping with other views."

If you think about it, this makes sense since there is no leading constraint for the price button and if you were to add a new button to the left of the price button, you do run the risk of the price button accidentally expanding enough to overlap that hypothetical button. In this particular instance, it is not strictly necessary to do anything since there won't be any other buttons for the price button to overlap. But if you wanted to remove the compiler warning, all you need to do is to add a leading constraint for the price button.

Navigating to the product page on iTunes

Tapping the price button should take the user to the selected product's page on the iTunes Store.

- Add the following method to **DetailViewController.swift**:

```
@IBAction func openInStore() {
    if let url = URL(string: searchResult.storeURL) {
        UIApplication.shared.open(url, options: [:],
                                   completionHandler: nil)
    }
}
```

- Connect the `openInStore` action to the button's Touch Up Inside event in the storyboard.

That's all you have to do. The web service returns a URL for the product page. You simply tell the `UIApplication` object to open this URL. iOS will now figure out what sort of URL it is and launch the proper app in response — iTunes Store, App Store, or Mobile Safari. On the Simulator you'll probably receive an error message that the URL could not be opened — try it on a device instead.

Note: You haven't used `UIApplication` before, but every app has a `UIApplication` object and it handles application-wide functionality. You won't directly use `UIApplication` a lot, except for special features such as opening URLs. Instead, most of the time you deal with `UIApplication` through your `AppDelegate` class, which — as you can guess from its name — is the delegate for `UIApplication`.

Loading artwork

For the Detail pop-up, you need to display a slightly larger, more detailed image than the one from the table view cell. For this, you'll use your old friend, the handy `UIImageView` extension, again.

- First add a new instance variable to `DetailViewController.swift`. This is necessary to cancel the download task:

```
var downloadTask: URLSessionDownloadTask?
```

- Then add the following line to `updateUI()`:

```
// Get image
if let largeURL = URL(string: searchResult.imageLarge) {
    downloadTask = artworkImageView.loadImage(url: largeURL)
}
```

This is the same thing you did in `SearchResultCell`, except that you use the other artwork URL — 100×100 pixels — and no placeholder image.

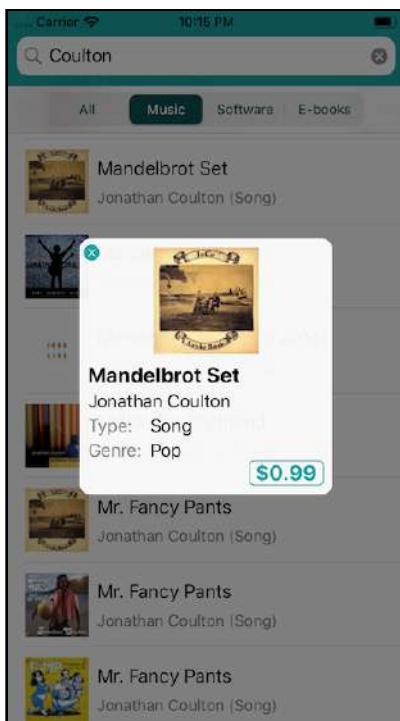
It's a good idea to cancel the image download if the user closes the pop-up before the image has been downloaded completely.

- To do that, add a `deinit` method:

```
deinit {
    print("deinit \(self)")
    downloadTask?.cancel()
}
```

Remember that `deinit` is called whenever the object instance is deallocated and its memory is reclaimed. That happens after the user closes the `DetailViewController` and the animation to remove it from the screen has completed. If the download task is not done by then, you cancel it.

- Try it out!



The pop-up now shows the artwork image

Did you see the `print()` from `deinit` after closing the pop-up? It's always a good idea to log a message when you're first trying out a new `deinit` method, to see if it really works. If you don't see that `print()`, it means `deinit` is never called, and you may have an ownership cycle somewhere keeping your object alive longer than intended. This is why you used `[weak self]` in the closure from the `UIImageView` extension, to break any such ownership cycles.

► This is a good time to commit the changes.

You can find the project files for this chapter under **42 – The Detail Pop-up** in the Source Code folder.

Chapter 43: Polish the Pop-up

By Eli Ganim

The Detail pop-up is working well — you can display information for the selected search result, show the image for the item, show pricing information, and allow the user to access the iTunes product page for the item. You are done with the Detail pop-up and can move on to the next item, right?

Well, not quite... There are still a few things you can do to make the Detail pop-up more polished and user friendly.

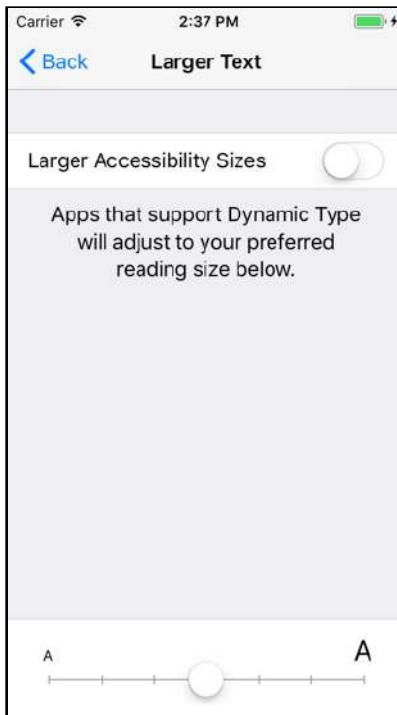
This chapter will cover the following:

- **Dynamic type:** Add support for dynamic type so that your text can display at a size specified by the user.
- **Gradients in the background:** Add a gradient background to make the Detail pop-up background look more polished.
- **Animation!:** Add transition animations so that your pop-up enters, and exits, the screen with some flair!



The iOS Settings app has an accessibility option — under **General ▶ Accessibility ▶ Larger Text** — that allows users to choose larger or smaller text. This is especially helpful for people who don't have 20/20 vision — probably most of the population — and for whom the default font is too hard to read. Nobody likes squinting at their device!

You can find this setting both in your device and in the Simulator:



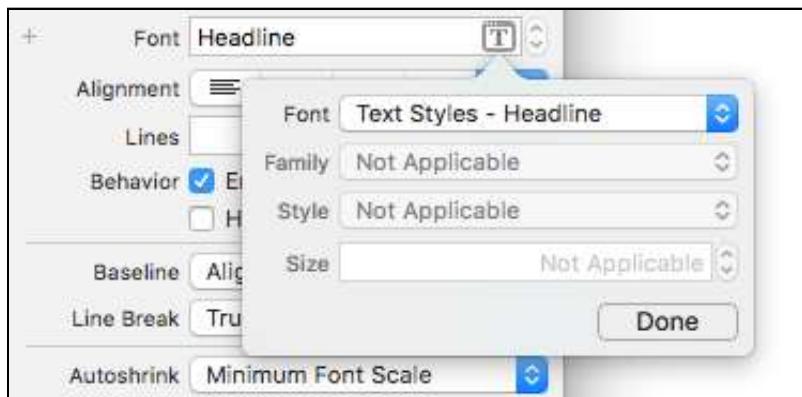
The Larger Text accessibility settings

Apps have to opt-in to use this **Dynamic Type** feature. Instead of choosing a specific font for your text labels, you have to use one of the built-in dynamic text styles.

Configuring for Dynamic Type

To provide a better user experience for all users, whether their eyesight is good or bad, you'll change the Detail pop-up to use Dynamic Type for its labels.

- Open the storyboard and go to the **Detail** scene. Change the **Font** setting for the **Name** label to the **Headline** text style:



Changing the font to the dynamic Headline style

You can't pick a font size when selecting text styles — the font size depends on the user and the Larger Text setting they use on their device.

- Set the **Lines** attribute to 0. This allows the Name label to fit more than one line of text.

Auto Layout for Dynamic Type

Of course, if you don't know beforehand how large the label's font will be, you also won't know how large the label itself will end up being, especially if it sometimes may have more than one line of text. You won't be surprised to hear that Auto Layout and Dynamic Type go hand-in-hand.

You want to make the name label resizable so that it can hold any amount of text at any possible font size, but it cannot go outside the bounds of the pop-up, nor overlap the labels below.

The trick is to capture these requirements in Auto Layout constraints.

Previously you've used the Add New Constraints button to make constraints, but that may not always give you the constraints you want. With this menu, pins are expressed as the amount of "spacing to nearest neighbor." But what exactly is the nearest neighbor?

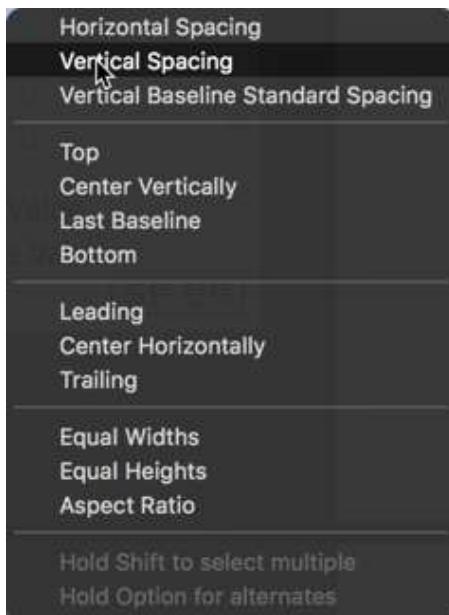
If you use the Add New Constraints button on the Name label, Interface Builder may decide to pin it to the bottom of the close button, which is weird. It makes more sense to pin the Name label to the image view instead. That's why you're going to use a different way to make constraints.

- Select the **Name** label. Now **Control-drag** to the **Image View** and let go of the mouse button.



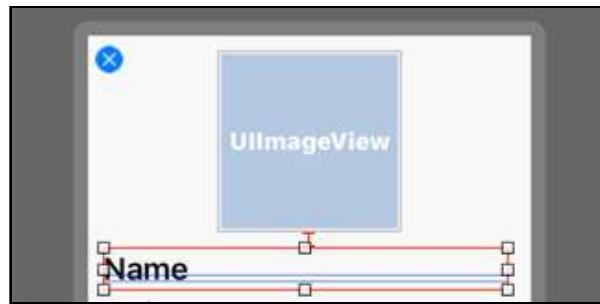
Control-drag to make a new constraint between two views

From the pop-up that appears, choose **Vertical Spacing**:



The possible constraint types

This puts a vertical spacing constraint between the label and the image view:

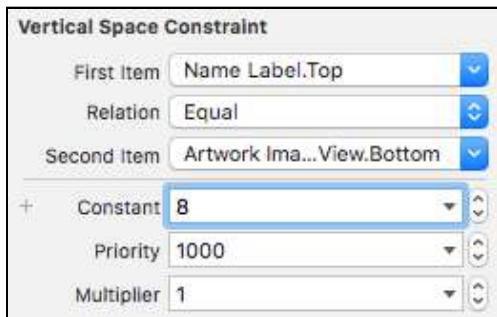


The new vertical space constraint

Of course, you'll also get some red lines because the label still needs additional constraints.

The vertical space you just added needs to be 8 points.

- Select the constraint — by carefully clicking it with the mouse or by selecting it from the Document Outline — then go to the **Size inspector**; or the Attributes inspector, they both show the same settings for layout constraints, and make sure that **Constant** is set to **8**.



Attributes for the vertical space constraint

Note that the inspector clearly describes what sort of constraint this is: Name Label.Top is connected to Artwork Image View.Bottom with a distance (Constant) of 8 points.

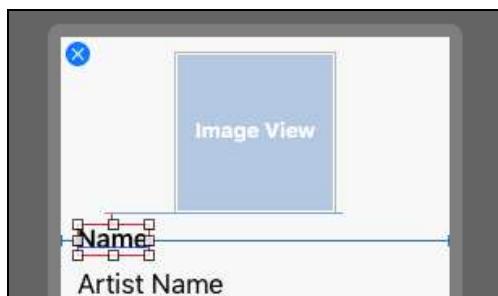
- Select the **Name** label again and **Control-drag** to the left and connect it to **Pop-up View**. Select **Leading Space to Container**:



The pop-up shows different constraint types

This adds a blue bar on the left. Notice how the pop-up offered different options this time? The constraints that you can set depend on the direction that you're dragging in.

- Repeat the step but this time Control-drag to the right. Now choose **Trailing Space to Container**.



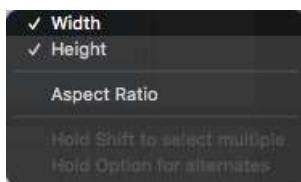
The constraints for the Name label

The Name label is now connected to the left edge of the Pop-up View and to its right edge — enough to determine its X-position and width — and to the bottom of the image view, for its Y-position. There is no constraint for the label's height, allowing it to grow as tall as it needs to using its intrinsic content size.

Shouldn't these constraints be enough to uniquely determine the label's position and size? If so, why is there still a red box?

Simple: the image view now has a constraint attached to it, and therefore no longer gets automatic constraints. You also have to add constraints that give the image view its position and size.

- Select the **Image View**, Control-drag up to the Pop-up View, and choose **Top Space to Container**. That takes care of the Y-position.
- Repeat but now Control-drag to the left (or right) and choose **Center Horizontally in Container**. That center-aligns the image view to take care of the X-position. If you don't see this option, then make sure you're not dragging outside the Pop-up View.
- Control-drag diagonally this time, but let go of the mouse button while you're *still inside the image view*. Hold down **Shift** and put checkmarks in front of both **Width** and **Height**, then press **return**. If you don't see both options, make sure you Control-drag diagonally instead of straight up or sideways.

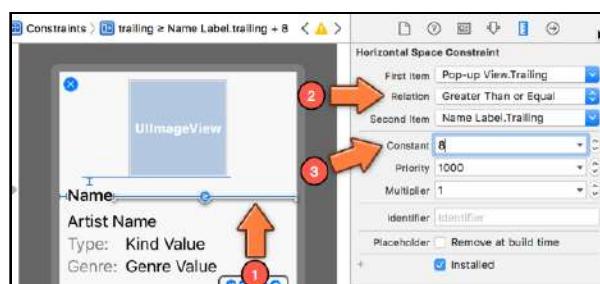


Adding multiple constraints at once

Now the image view and the Name label will have all blue bars.

There's one more thing you need to fix. Look again at that blue bar to the right of the Name label. This forces the label to be always about 45 points wide. That's not what you want; instead, the label should be able to grow until it reaches the edge of the Pop-up View.

- Click that blue bar to select it and go to the **Size inspector**. Change **Relation** to **Greater Than or Equal**, and **Constant** to **8**.



Converting the constraint to Greater Than or Equal

Now this constraint can resize to allow the label to grow, but it can never become smaller than 8 points. This ensures there is at least an 8 point margin between the label and the edge of the Detail pop-up.

By the way, notice how this constraint is between Pop-up View.Trailing and Name Label.Trailing? In Auto Layout terminology, trailing means “on the right,” while leading means “on the left.”

Why didn’t they just call this left and right? Well, not everyone writes in the same direction. With right-to-left languages such as Hebrew or Arabic, the meaning of trailing and leading is reversed. This allows your layouts to work without changes for those languages too.

► Run the app and try it out:

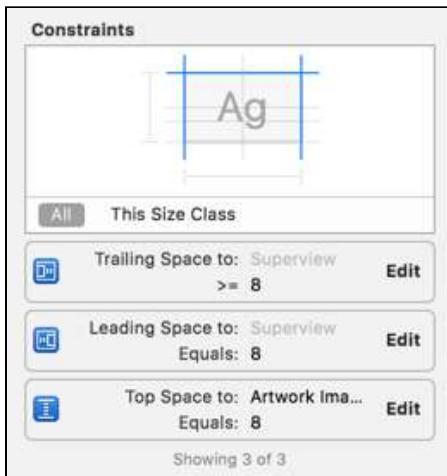


The text overlaps the other labels

Well, the word-wrapping seems to work, but the text overlaps the labels below it. Let’s add some more constraints so that the other labels get pushed down instead.

Tip: In the next steps you’ll change the properties of the constraints using the Attributes inspector, but it can be quite tricky to select those constraints. The blue bars are often tiny, making them difficult to click. It’s often easier to find the constraint in the Document Outline, but it’s not always immediately obvious which one you need.

A smarter way to find a constraint is to first select the view it belongs to, then go to the Size inspector and look in the Constraints section. Here is what it looks like for the Name label:



The Name label's constraints in the Size inspector

To edit the constraint, double-click it or use the **Edit** button to the right of each constraint.

OK, let's make those changes...

- Select the **Artist Name** label and set its **Font** to the **Subhead** text style.
- Set the **Font** of the other four labels to the **Caption 1** text style. You can do this in a single go if you multiple-select these labels by holding down the **⌘** key.

Auto Layout for Artist Name

Let's pin the **Artist Name** label. Again you do this by Control-dragging.

- Pin it to the left with a **Leading Space to Container**.
- Pin it to the right with a **Trailing Space to Container**. Just like before, change this constraint's **Relation** to **Greater Than or Equal** and **Constant** to **8**.
- Pin it to the Name label with a **Vertical Spacing**. Change this to size **4**.

Auto Layout for Type

For the **Type:** label:

- Pin it to the left with a **Leading Space to Container**.
- Pin it to the **Artist Name** label with a **Vertical Spacing**, size **8**.

The **Kind Value** label is slightly different:

- Pin it to the right with a **Trailing Space to Container**. Change this constraint's **Relation to Greater Than or Equal and Constant** to **8**.
- Control-drag from **Kind Value** to **Type** and choose **First Baseline**. This aligns the bottom of the text of both labels. This alignment constraint determines the **Kind Value**'s Y-position so you don't have to make a separate constraint for that.

Auto Layout for Genre

Two more labels to go. For the **Genre:** label:

- Pin it to the left with a **Leading Space to Container**.
- Pin it to the **Type:** label with a **Vertical Spacing**, size **4**.
- On the right, pin it to the **Genre Value** label with a **Horizontal Spacing**. This should be a **8** point distance.

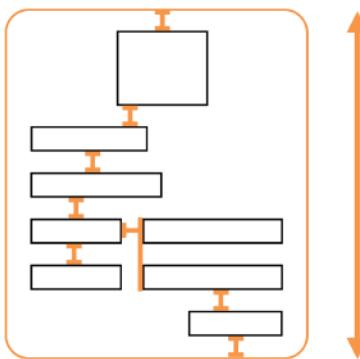
And finally, the **Genre Value** label:

- Pin it to the right with a **Trailing Space to Container, Greater Than or Equal 8**.
- Make a **First Baseline** alignment between **Genre Value** and **Genre:**.
- Make a **Leading** alignment between **Genre Value** and **Kind Value**. This makes these two labels neatly align on the left.
- Resolve any Auto Layout issues by selecting **Editor ▶ Resolve Auto Layout Issues ▶ Update Frames** from the Xcode menu. You may need to set the Constant of the alignment constraints to 0 if things don't line up properly.

That's quite a few constraints, but using Control-drag to make them is quite fast. With some experience you'll be able to whip together complex Auto Layout constraints in no time.

Auto Layout for Price button

There is one more thing to do. The last row of labels needs to be pinned to the price button. That way there are constraints going all the way from the top of the Pop-up View to the bottom. The heights of the labels plus the sizes of the Vertical Spacing constraints between them will now determine the height of the Detail pop-up.



The height of the pop-up view is determined by the constraints

- Control-drag from the **\$9.99** button up to **Genre Value**. Choose **Vertical Spacing**. In the Size inspector, set **Constant** to **10**.

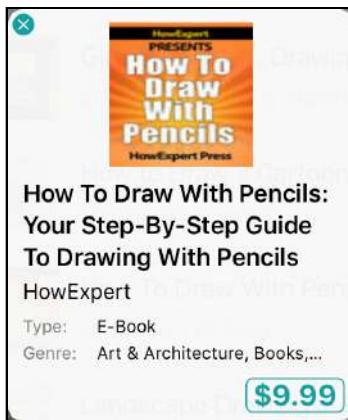
While you might not notice this immediately, this introduces some Auto Layout constraint issues at this point — try clicking on the **Genre:** or **Name** labels and you'll see some constraints turn red.

This is because the Pop-up View still has a Height constraint that forces it to be 240 points high. But the labels, image, and the vertical space constraints on these views don't add up to 240.

- You no longer need this Height constraint, so select it — the one called **height = 240** in the Document Outline — and press **delete** to get rid of it.
- If necessary — if you have any views with orange rectangles around them — from the **Editor ▶ Resolve Auto Layout Issues** menu, choose **Update Frames** from the “All Views” section.

Now all your constraints turn blue and everything fits snugly together.

- Run the app to try it out.



The text properly wraps without overlapping

You now have an automatically resizing Detail pop-up that uses Dynamic Type for its labels!

Testing Dynamic Type

- Close the app and open the Settings app. Go to **General** ▶ **Accessibility** ▶ **Larger Text**. Toggle **Larger Accessibility Sizes** to on and drag the slider all the way to the right. That gives you the maximum font size — it's huge!

Now go back to StoreSearch and open a new pop-up. The text is a lot bigger:



Changing the text size results in a bigger font

For fun, change the font of the Name label to Body. Bazinga, that's some big text!

When you're done playing, put the Name label font back to Headline, and turn off the Larger Text setting — the slider goes in the middle.

Dynamic Type is an important feature to add to your apps. This was only a short introduction, but hopefully the principle is clear: instead of a font with a fixed size, you use one of the available Text Styles: Body, Headline, Caption, and so on.

Then you set up Auto Layout constraints to make your views resizable and looking good no matter how large or small the font.

- This is a good time to commit the changes.

Exercise: Set up the cells from the table view for Dynamic Type. There's a catch: when the user returns from changing the text size settings, the app should refresh the screen without needing an app restart. You can do this by reloading the table view when the app receives a `UIContentSizeCategoryDidChange` notification — see the previous app for a refresher on how to handle notifications.

Also check out the property `adjustsFontSizeForContentSizeCategory` on `UILabel`. If you set this to `true`, then the app will automatically update the label whenever the font size changes. Good luck! Check the forums at forums.raywenderlich.com for solutions from other readers.

Stack Views

Setting up all those constraints was quite a bit of work, but it was good Auto Layout practice! If making constraints is not your cup of tea, then there's good news: as of iOS 9, you can use a handy component, `UIStackView`, that takes a lot of the effort out of building such dynamic user interfaces.

Using stack views is fairly straightforward: you drop a **Horizontal** or **Vertical Stack View** in your scene, and then you put your labels, image views, and buttons inside that stack view. Of course, a stack view can contain other stack views as well, allowing you to create very complex layouts quite easily.

Give it a try! See if you can build the Detail pop-up with stack views. If you get stuck, we have a video tutorial series on the website that goes into great detail on `UIStackView`: raywenderlich.com/tag/stack-view

Gradients in the background

As you can see in the previous screenshots, the table view in the background is dimmed by the view of the `DetailViewController`, which is 50% transparent black. That allows the pop-up to stand out more.

It works well, but a plain black overlay is a bit dull. Let's turn it into a circular gradient instead.

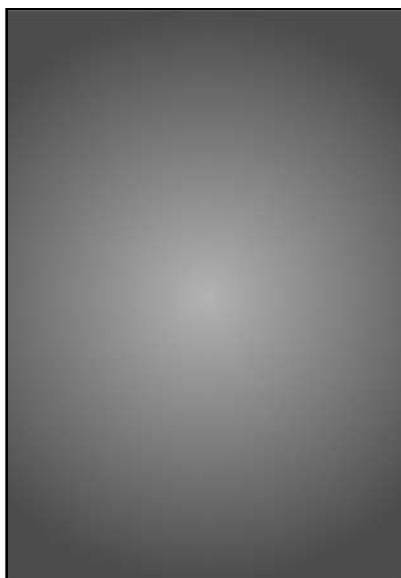
You could use Photoshop to draw such a gradient and place an image view behind the pop-up, but why use an image when you can also draw using Core Graphics? Additionally, an image would increase the size of your app and might also create some issues when you need to support larger screen sizes.

To pull this off, you will create your own `UIView` subclass.

The GradientView class

- Add a new **Swift File** to the project. Name it **GradientView**.

This will be a very simple view. It simply draws a black circular gradient that goes from mostly opaque in the corners to mostly transparent in the center. Placed on a white background, it looks something like this:



What the GradientView looks like by itself

- Replace the contents of **GradientView.swift** with:

```
import UIKit

class GradientView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = UIColor.clear
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        backgroundColor = UIColor.clear
    }

    override func draw(_ rect: CGRect) {
        // 1
        let components: [CGFloat] = [ 0, 0, 0, 0.3, 0, 0, 0, 0, 0.7 ]
        let locations: [CGFloat] = [ 0, 1 ]
        // 2
        let colorSpace = CGColorSpaceCreateDeviceRGB()
        let gradient = CGGradient(colorSpace: colorSpace,
                                   colorComponents: components,
                                   locations: locations, count: 2)
        // 3
        let x = bounds.midX
        let y = bounds.midY
        let centerPoint = CGPoint(x: x, y : y)
        let radius = max(x, y)
        // 4
        let context = UIGraphicsGetCurrentContext()
        context?.drawRadialGradient(gradient!,
                                    startCenter: centerPoint, startRadius: 0,
                                    endCenter: centerPoint, endRadius: radius,
                                    options: .drawsAfterEndLocation)
    }
}
```

In the `init(frame:)` and `init?(coder:)` methods you simply set the background color to fully transparent — the “clear” color. Then in `draw()` you draw the gradient on top of that transparent background, so that it blends with whatever is below.

The drawing code uses the Core Graphics framework. It may look a little scary but this is what it does:

1. First, you create two arrays that contain the “color stops” for the gradient. The first color (0, 0, 0, 0.3) is a black color that is mostly transparent. It sits at location 0 in the gradient, which represents the center of the screen because you’ll be drawing a circular gradient.

The second color (0, 0, 0, 0.7) is also black but much less transparent and sits at location 1, which represents the circumference of the gradient's circle. Remember that in UIKit, and also in Core Graphics, colors and opacity values don't go from 0 to 255 but are fractional values between 0.0 and 1.0.

The 0 and 1 from the `locations` array represent percentages: 0% and 100%, respectively. If you have more than two colors, you can specify the percentages of where in the gradient you want to place these colors.

2. With those color stops you can create the gradient. This gives you a new `CGGradient` object.
3. Now that you have the gradient object, you have to figure out how big you need to draw it. The `midX` and `midY` properties return the center point of a rectangle. That rectangle is given by `bounds`, a `CGRect` object that describes the dimensions of the view. If possible, it's better to not hard-code any dimensions such as "375 by 667 points." By using `bounds`, you can use this view anywhere you want to, no matter how big a space it should fill. You can use it without problems on any screen size from the smallest iPhone to the biggest iPad. The `centerPoint` constant contains the coordinates for the center point of the view and `radius` contains the larger of the `x` and `y` values; `max()` is a handy function that you can use to determine which of two values is the biggest.
4. With all those preliminaries done, you can finally draw the thing. Core Graphics drawing always takes places in what's known as a *graphics context*. We're not going to worry about exactly what that is, just know that you need to obtain a reference to the current context and then you can do your drawing.

And finally, the `drawRadialGradient()` function draws the gradient according to your specifications.

Generally speaking, it isn't optimal to create new objects inside your `draw()` method, such as gradients, especially if `draw()` is called often. In such cases it is better to create the objects the first time you need them and to reuse the same instance over and over — lazy loading for the win!

However, you don't really have to do that here because this `draw()` method will be called just once — when the `DetailViewController` gets loaded — so you can get away with being less than optimal.

Note: By the way, you'll only be using `init(frame:)` to create the `GradientView` instance. The other init method, `init?(coder:)`, is never used



in this app. However, `UIView` demands that all subclasses implement `init?(coder:)` — that is why it is marked as required — and if you remove this method, Xcode will complain with an error.

Using GradientView

Putting this new `GradientView` class to work is pretty easy. You'll add it to your own presentation controller object. That way, the `DetailViewController` doesn't need to know anything about it. Dimming the background is really a side effect of doing a presentation, so it belongs in the presentation controller.

- Open `DimmingPresentationController.swift` and add the following code to the class:

```
lazy var dimmingView = GradientView(frame: CGRect.zero)

override func presentationTransitionWillBegin() {
    dimmingView.frame = containerView!.bounds
    containerView!.insertSubview(dimmingView, at: 0)
}
```

The `presentationTransitionWillBegin()` method is invoked when the new view controller is about to be shown on the screen. Here you create the `GradientView` object, make it as big as the `containerView`, and insert it behind everything else in this “container view.”

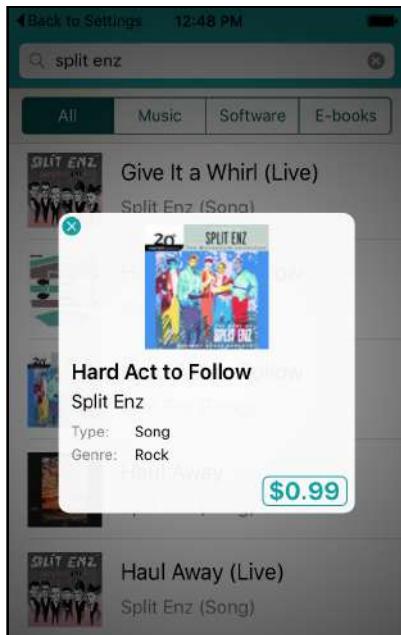
The container view is a new view that is placed on top of the `SearchViewController`, and it contains the views from the `DetailViewController`. So this piece of logic places the `GradientView` in between those two screens.

There's one more thing to do: because the `DetailViewController`'s background color is still 50% black, this color gets multiplied with the colors inside the gradient view, making the gradient look extra dark. It's better to set the background color to 100% transparent, but if we do that in the storyboard, it makes it harder to see and edit the pop-up view. So let's do this in code instead.

- Add the following line to `viewDidLoad()` in `DetailViewController.swift`:

```
view.backgroundColor = UIColor.clear
```

- Run the app and see what happens.



The background behind the pop-up now has a gradient

Nice! That looks a lot smarter.

Animation!

The pop-up itself looks good already, but the way it enters the screen — Poof! It's suddenly there — is a bit unsettling. iOS is supposed to be the king of animation, so let's make good on that.

You've used Core Animation and UIView animations before. This time you'll use a **keyframe animation** to make the pop-up bounce into view.

To animate the transition between two screens, you use an animation controller object. The purpose of this object is to animate a screen while it's being presented or dismissed, nothing more.

Now let's add some liveliness to this pop-up!

The animation controller class

- Add a new **Swift File** to the project, named **BounceAnimationController**.
- Replace the contents of the new file with:

```
import UIKit

class BounceAnimationController: NSObject,
    UIViewControllerAnimatedTransitioning {

    func transitionDuration(using transitionContext:
        UIViewControllerContextTransitioning?) -> TimeInterval {
        return 0.4
    }

    func animateTransition(using transitionContext:
        UIViewControllerContextTransitioning) {

        if let toViewController = transitionContext.viewController(
            forKey: UITransitionContextViewControllerKey.to),
            let toView = transitionContext.view(
                forKey: UITransitionContextViewKey.to) {

            let containerView = transitionContext.containerView
            toView.frame = transitionContext.finalFrame(for:
                toViewController)
            containerView.addSubview(toView)
            toView.transform = CGAffineTransform(scaleX: 0.7, y: 0.7)

            UIView.animateKeyframes(withDuration: transitionDuration(
                using: transitionContext), delay: 0, options:
                .calculationModeCubic, animations: {
                    UIView.addKeyframe(withRelativeStartTime: 0.0,
                        relativeDuration: 0.334, animations: {
                            toView.transform = CGAffineTransform(scaleX: 1.2,
                                y: 1.2)
                    })
                    UIView.addKeyframe(withRelativeStartTime: 0.334,
                        relativeDuration: 0.333, animations: {
                            toView.transform = CGAffineTransform(scaleX: 0.9,
                                y: 0.9)
                    })
                    UIView.addKeyframe(withRelativeStartTime: 0.666,
                        relativeDuration: 0.333, animations: {
                            toView.transform = CGAffineTransform(scaleX: 1.0,
                                y: 1.0)
                    })
                }, completion: { finished in
```



```
        transitionContext.completeTransition(finished)
    }
}
```

To become an animation controller, the object needs to extend `NSObject` and also implement the `UIViewControllerAnimatedTransitioning` protocol — quite a mouthful! The important methods from this protocol are:

- `transitionDuration(using:)` – This determines how long the animation is. You’re making the pop-in animation last for only 0.4 seconds, but that’s long enough. Animations are fun, but they shouldn’t keep the user waiting.
- `animateTransition(using:)` – This performs the actual animation.

To find out what to animate, you look at the `transitionContext` parameter. This gives you a reference to a new view controller and lets you know how big it should be.

The actual animation starts at the line `UIView.animateKeyframes(...)`. This works like all `UIView`-based animations: you set the initial state before the animation block, and `UIKit` will automatically animate any properties that get changed inside the closure. The difference from before is that a keyframe animation lets you animate the view in several distinct stages.

The property you’re animating is the `transform`. If you’ve ever taken any matrix math you’ll be pleased — or terrified! — to hear that this is an affine transformation matrix. It allows you to do all sorts of funky stuff with the view, such as rotating it or shearing it. But the most common use of the transform is for scaling.

The animation consists of several **keyframes**. It will smoothly proceed from one keyframe to the next over a certain amount of time. Because you’re animating the view’s scale, the different `toView.transform` values represent how much bigger or smaller the view will be over time.

The animation starts with the view scaled down to 70% (scale 0.7). The next keyframe inflates it to 120% of its normal size. After that, it will scale the view down a bit again but not as much as before — only 90% of its original size. The final keyframe ends up with a scale of 1.0, which restores the view to an undistorted shape.

By quickly changing the view size from small to big to small to normal, you create a bounce effect.

You also specify the duration between the successive keyframes. In this case, each transition from one keyframe to the next takes 1/3rd of the total animation time. These times are not in seconds but in fractions of the animation's total duration, which is 0.4 seconds.

Feel free to mess around with the animation code. No doubt you can make it much more spectacular!

Using the new animation controller

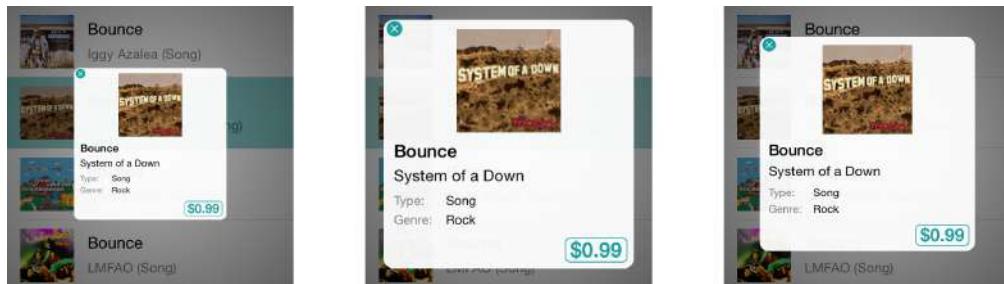
To use this animation in your app, you have to tell the app to use the new animation controller when presenting the Detail pop-up. That happens in the transitioning delegate inside **DetailViewController.swift**.

- Add the following method to the `UIViewControllerAnimatedTransitioning` extension:

```
func animationController(forPresented presented:  
    UIViewController, presenting: UIViewController,  
    source: UIViewController) ->  
    UIViewControllerAnimatedTransitioning? {  
    return BounceAnimationController()  
}
```

And that's all you need to do.

- Run the app and get ready for some bouncing action!



The pop-up animates

The pop-up looks a lot spiffier with the bounce animation, but there are two things that could be better: the `GradientView` still appears abruptly in the background, and the animation upon dismissal of the pop-up is very plain.

Animating the background

There's no reason why you cannot have two things animating at the same time. So, let's make the GradientView fade in while the pop-up bounces into view. That is a job for the presentation controller, because that's what provides the gradient view.

- Go to **DimmingPresentationController.swift** and add the following to the end of `presentationTransitionWillBegin()`:

```
// Animate background gradient view
dimmingView.alpha = 0
if let coordinator =
    presentedViewController.transitionCoordinator {
    coordinator.animate(alongsideTransition: { _ in
        self.dimmingView.alpha = 1
    }, completion: nil)
}
```

You set the alpha value of the gradient view to 0 to make it completely transparent, and then animate it back to 1 — or 100% — and fully visible, resulting in a simple fade-in. That's a bit more subtle than making the gradient appear so abruptly. The special thing here is the `transitionCoordinator` stuff. This is the UIKit traffic cop in charge of coordinating the presentation controller and animation controllers and everything else that happens when a new view controller is presented.

The important thing to know about the `transitionCoordinator` is that all of your animations should be done in a closure passed to `animateAlongsideTransition` to keep the transition smooth. If your users wanted choppy animations, they wouldn't be using iPhones, would they?

- Also add the method `dismissalTransitionWillBegin()`, which is used to animate the gradient view out of sight when the Detail pop-up is dismissed:

```
override func dismissalTransitionWillBegin() {
    if let coordinator =
        presentedViewController.transitionCoordinator {
        coordinator.animate(alongsideTransition: { _ in
            self.dimmingView.alpha = 0
        }, completion: nil)
    }
}
```

This does the reverse: it animates the alpha value back to 0% to make the gradient view fade out.

- Run the app. The dimming gradient now appears almost without you even noticing it. Slick!

Animating the pop-up exit

After tapping the Close button, the pop-up slides off the screen, like modal screens always do. Let's make this a bit more exciting and make it slide up instead of down. For that you need another animation controller.

- Add a new **Swift File** to the project, named **SlideOutAnimationController**.
- Replace the new file's contents with:

```
import UIKit

class SlideOutAnimationController: NSObject,
    UIViewControllerAnimatedTransitioning {
    func transitionDuration(using transitionContext:
        UIViewControllerContextTransitioning?) -> TimeInterval {
        return 0.3
    }

    func animateTransition(using transitionContext:
        UIViewControllerContextTransitioning) {
        if let fromView = transitionContext.view(forKey:
            UITransitionContextViewKey.from) {
            let containerView = transitionContext.containerView
            let time = transitionDuration(using: transitionContext)
            UIView.animate(withDuration: time, animations: {
                fromView.center.y -= containerView.bounds.size.height
                fromView.transform = CGAffineTransform(scaleX: 0.5,
                                                    y: 0.5)
            }, completion: { finished in
                transitionContext.completeTransition(finished)
            })
        }
    }
}
```

This is pretty much the same as the other animation controller, except that the animation itself is different. Inside the animation block you subtract the height of the screen from the view's center position while simultaneously zooming it out to 50% of its original size, making the Detail screen fly up-up-and-away.

- In **DetailViewController.swift**, add the following method to the **UIViewControllerTransitioningDelegate** extension:

```
func animationController(forDismissed dismissed:  
    UIViewController) -> UIViewControllerAnimatedTransitioning? {  
    return SlideOutAnimationController()  
}
```

This simply overrides the animation controller to be used when a view controller is dismissed.

- Run the app and try it out. That looks pretty sweet if you ask me!
- If you're happy with the way the animations look, then commit your changes.

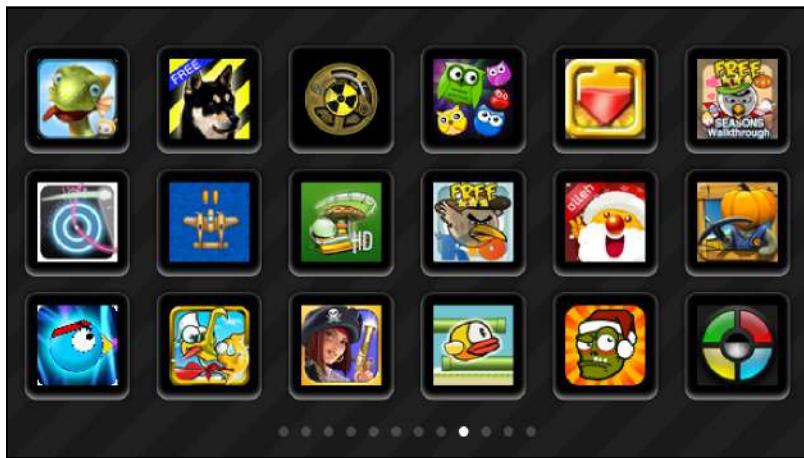
Exercise: Create some exciting new animations. You can definitely improve on the existing ones. Hint: use the `transform` matrix to add some rotation to the mix.

You can find the project files for this chapter under **43 – Polish the Pop-up** in the Source Code folder.

Chapter 44: Landscape

By Eli Ganim

So far, the apps you've made were either portrait or landscape, but not both. Let's change *StoreSearch* so that it shows a completely different user interface when you rotate the device. When you're done, the app will look like this:



The app will look completely different in landscape orientation

The landscape screen shows just the artwork for the search results. Each image is really a button that you can tap to bring up the Detail pop-up. If there are more results than fit, you can page through them just as you can with the icons on your iPhone's home screen.

You'll cover the following in this chapter:

- **The landscape view controller:** Create a basic landscape view controller to make sure that the functionality works.
- **Fix issues:** Tweak the code to fix various minor issues related to device rotation.
- **Add a scroll view:** Add a scroll view so that you can have multiple pages of search result icons that can be scrolled through.
- **Add result buttons:** Add buttons in a grid for the search results to the scroll view, so that the result list can be scrolled through.
- **Paging:** Configure scrolling through results page-by-page rather than as a single scrolling list.
- **Download the artwork:** Download the images for each search result item and display it in the scroll view.

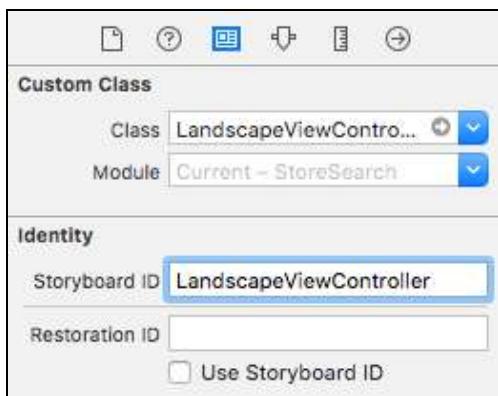
The landscape view controller

Let's begin by creating a very simple view controller that shows just a text label.

The storyboard

- Add a new file to the project using the **Cocoa Touch Class** template. Name it **LandscapeViewController** and make it a subclass of **UIViewController**.
- In Interface Builder, with **Main** storyboard open, drag a new **View Controller** on to the canvas.
- In the Document Outline, click on the yellow circle for the view controller and change its name to **Landscape**.
- In the Identity inspector, change the **Class** to **LandscapeViewController**. Also type this into the **Storyboard ID** field.





Giving the view controller an ID

There will be no segue to this view controller. Instead, you'll instantiate this view controller programmatically when you detect a device rotation. For that, it needs to have an ID so you can uniquely identify this particular view controller in the storyboard.

- Use the **View as:** panel to change the orientation to landscape.



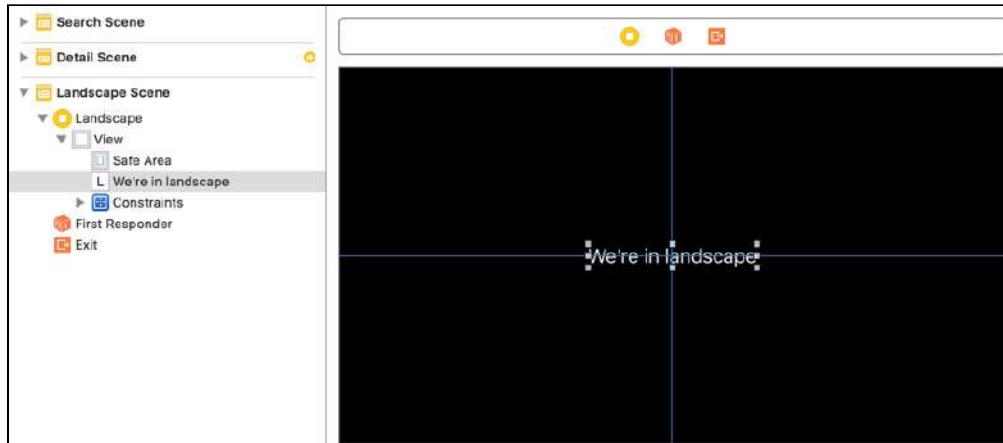
Changing Interface Builder to landscape

This flips *all* the scenes in the storyboard to landscape, but that is OK — it doesn't change what happens when you run the app. Putting Interface Builder in landscape mode is just a design aid that makes it easier to lay out your UI. What actually happens when you run the app depends on the orientation the user holds the device in. The trick is to use Auto Layout constraints to make sure that the view controllers properly resize to landscape or portrait at runtime.

- Change **View - Background** to **Black** color for the Landscape scene.
- Drag a new **Label** into the scene and give it some text. You're just using this label to verify that the new view controller shows up in the correct orientation.
- Change the label's **Label - Color** to **White**, and if not all the text is showing use the **Editor ▶ Size to Fit Content** menu option (or the ⌘= shortcut) to resize the label to fit its content.

- Use the **Align** Auto Layout menu to center the label horizontally and vertically.

Your design should look something like this:



Initial design for the Landscape scene

Show the landscape view on device rotation

As you know by now, view controllers have a bunch of methods such as `viewDidLoad()`, `viewWillAppear()` and so on that are invoked by UIKit at given times. There is also a method that is invoked when the device is rotated. You can override this method to show (and hide) the new `LandscapeViewController`.

- Add the following method to `SearchViewController.swift`:

```
override func willTransition(
    to newCollection: UITraitCollection,
    with coordinator: UIViewControllerTransitionCoordinator) {
    super.willTransition(to: newCollection, with: coordinator)

    switch newCollection.verticalSizeClass {
    case .compact:
        showLandscape(with: coordinator)
    case .regular, .unspecified:
        hideLandscape(with: coordinator)
    @unknown default:
        fatalError()
    }
}
```

This method isn't just invoked on device rotations, but any time the *trait collection* for the view controller changes. So what is a trait collection? It is, um, a collection of

traits, where a trait can be:

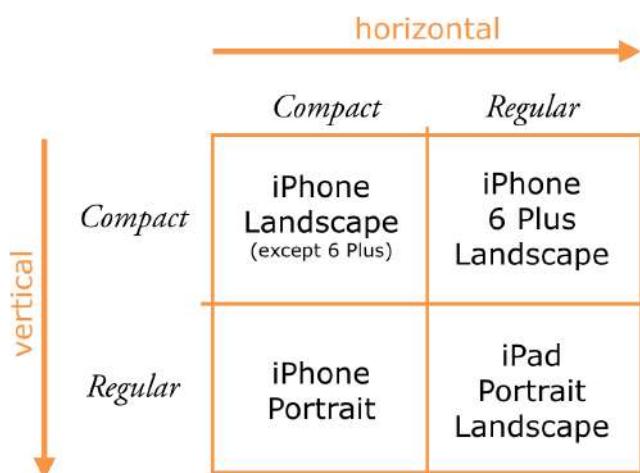
- The horizontal size class
- The vertical size class
- The display scale — is this a Retina screen or not?
- The user interface idiom — is this an iPhone or iPad?
- The preferred Dynamic Type font size
- And a few other things

Whenever one or more of these traits change, for whatever reason, UIKit calls `willTransition(to:with:)` to give the view controller a chance to adapt to the new traits.

What we are interested in here are the *size classes*. This feature allows you to design a user interface that is independent of the device's actual dimensions or orientation. With size classes, you can create a single storyboard that works across all devices, from iPhone to iPad — a “universal storyboard.”

So how exactly do these size classes work? Well, there's two of them, a horizontal one and a vertical one, and each can have two values: *compact* or *regular*.

The combination of these four things creates the following possibilities:



Horizontal and vertical size classes

When an iPhone app is in portrait orientation, the horizontal size class is *compact* and the vertical size class is *regular*.

Upon a rotation to landscape, the vertical size class changes to *compact*.

What you may not have expected is that the horizontal size class doesn't change and stays *compact* in both portrait and landscape orientations — except on the iPhone Plus models, that is.

In landscape, the horizontal size class on the Plus is *regular*. That's because the larger dimensions of the iPhone Plus devices can fit a split screen in landscape mode, like the iPad — something you'll see later on.

What this boils down to is, to detect an iPhone rotation you just have to look at how the vertical size class changed. That's exactly what the `switch` statement does:

```
switch newCollection.verticalSizeClass {  
    case .compact:  
        showLandscape(with: coordinator)  
    case .regular, .unspecified:  
        hideLandscape(with: coordinator)  
}
```

If the new vertical size class is `.compact` the device got flipped to landscape and you show the `LandscapeViewController`. But if the new size class is `.regular`, the app is back in portrait and you hide the landscape view again.

The reason the second `case` statement also checks `.unspecified` is because `switch` statements must always be exhaustive and have cases for all possible values. `.unspecified` shouldn't happen, but just in case it does, you also hide the landscape view. This is another example of defensive programming.

Just to keep things readable, the actual showing and hiding happens in methods of their own. You will add these next.

In the early years of iOS, it was tricky to put more than one view controller on the same screen. The motto used to be: one screen, one view controller. However, when devices with larger screens became available, that became inconvenient — you often want one area of the screen to be controlled by one view controller and a second area by a separate view controller. So now, view controllers are allowed to be part of other view controllers if you follow a few rules.

This is called *view controller containment*. These APIs are not limited to just the iPad; you can take advantage of them on the iPhone as well. These days a view controller is no longer expected to manage a screenful of content, but manages a “self-contained presentation unit,” whatever that may be for your app.



You're going to use view controller containment for the `LandscapeViewController`.

It would be eminently possible to make a modal segue to this scene and use your own presentation and animation controllers for the transition. But you've already done that and it's more fun to play with something new. Besides, it's useful to learn about containment and child view controllers.

► Add an instance variable to `SearchViewController.swift`:

```
var landscapeVC: LandscapeViewController?
```

This is an optional because there will only be an active `LandscapeViewController` instance if the phone is in landscape orientation. In portrait orientation this will be `nil`.

► Add the following helper method:

```
func showLandscape(with coordinator:  
                    UIViewControllerTransitionCoordinator) {  
    // 1  
    guard landscapeVC == nil else { return }  
    // 2  
    landscapeVC = storyboard!.instantiateViewController(  
        withIdentifier: "LandscapeViewController")  
        as? LandscapeViewController  
    if let controller = landscapeVC {  
        // 3  
        controller.view.frame = view.bounds  
        // 4  
        view.addSubview(controller.view)  
        addChild(controller)  
        controller.didMove(toParent: self)  
    }  
}
```

In previous apps you called `present(animated:completion:)` or made a segue to show a new modal screen. Here, however, you add the new `LandscapeViewController` as a *child* view controller of `SearchViewController`.

Here's how it works, step-by-step:

1. It should never happen that the app instantiates a second landscape view when you're already looking at one. The `guard` statement codifies this requirement. If it should happen that `landscapeVC` is not `nil`, then you're already showing the landscape view and you simply `return` right away.
2. Find the scene with the ID “`LandscapeViewController`” in the storyboard and instantiate it. Because you don't have a segue, you need to instantiate the view

controller manually. This is why you filled in that Storyboard ID field in the Identity inspector.

The `landscapeVC` instance variable is an optional, so you need to unwrap it before you can continue.

3. Set the size and position of the new view controller. This makes the landscape view just as big as the `SearchViewController`, covering the entire screen.

The `frame` is the rectangle that describes the view's position and size in terms of its superview. To move a view to its final position and size you usually set its `frame`. The `bounds` is also a rectangle but seen from inside the view.

Because `SearchViewController`'s view is the superview here, the `frame` of the landscape view must be made equal to the `SearchViewController`'s `bounds`.

4. These are the minimum required steps to add the contents of one view controller to another, in this order:
 - a. Add the landscape controller's view as a subview. This places it on top of the table view, search bar and segmented control.
 - b. Tell the `SearchViewController` that the `LandscapeViewController` is now managing that part of the screen, using `addChild()`. If you forget this step, then the new view controller may not always work correctly.
 - c. Tell the new view controller that it now has a parent view controller with `didMove(toParent:)`.

In this new arrangement, `SearchViewController` is the “parent” view controller, and `LandscapeViewController` is the “child.” In other words, the Landscape screen is embedded inside the `SearchViewController`.

Note: Even though it will appear on top of everything else, the Landscape screen is not presented modally. It is “contained” in its parent view controller, and therefore owned and managed by the parent — it isn't independent like a modal screen. This is an important distinction.

View controller containment is also used for navigation and tab bar controllers where the `UINavigationController` and `UITabBarController` “wrap around” their child view controllers.

Usually, when you want to show a view controller that takes over the whole screen, you'd use a modal segue. But when you want just a portion of the screen to be managed by its own view controller, you'd make it a child view



controller.

One of the reasons you’re not using a modal segue for the Landscape screen in this app, even though it is a full-screen view controller, is that the Detail pop-up already is modally presented and this could potentially cause conflicts. Besides, it’s a fun alternative to modal segues.

- To get the app to compile, add an empty implementation of the “hide” method:

```
func hideLandscape(with coordinator:  
    UIViewControllerTransitionCoordinator) {  
}
```

By the way, the transition coordinator parameter is needed for doing animations, which you’ll add soon.

- Try it out! Run the app, do a search and rotate your iPhone or the Simulator to landscape.



The Simulator after flipping to landscape

Remember: to rotate the Simulator, press ⌘ and the left (or right) arrow keys. It’s possible that the Simulator won’t flip over right away — it can be buggy like that. If that happens, press ⌘+arrow key a few more times.

This is not doing any animation just yet. As always, first get it to work right, and *then* make it look pretty.

If you don’t do a search first before rotating to landscape, the keyboard may remain visible. You’ll fix that shortly. In the mean time you can press ⌘+K (on the Simulator only) to hide the keyboard manually.

Switching back to the portrait view

Switching back to portrait doesn't work yet, but that's easily fixed.

- Replace the *method stub*, which is basically a method name with no implementation code, that you added earlier with the following implementation to hide the landscape view controller:

```
func hideLandscape(with coordinator:  
    UIViewControllerTransitionCoordinator) {  
    if let controller = landscapeVC {  
        controller.willMove(toParent: nil)  
        controller.view.removeFromSuperview()  
        controller.removeFromParent()  
        landscapeVC = nil  
    }  
}
```

This is essentially the inverse of what you did to embed the landscape view controller.

First, you call `willMove(toParent:)` to tell the view controller that it is leaving the view controller hierarchy and it no longer has a parent. Then, you remove its view from the screen, and finally, `removeFromParent()` truly disposes of the view controller.

You also set the instance variable to `nil` in order to remove the last strong reference to the `LandscapeViewController` object now that you're done with it.

- Run the app. Switching back to portrait should remove the black landscape view.

Note: If you press **⌘-right** (or **⌘-left**) twice, the Simulator first rotates to landscape and then to portrait, but the `LandscapeViewController` does not disappear. Why is that?

It's might not be immediately evident, but what you're looking at now is *not* portrait but portrait upside down. This orientation is not recognized by the app — see the Device Orientation setting under Deployment Info in the project settings — and therefore the app keeps thinking it's in landscape.

Press **⌘-right** (or **⌘-left**) twice again and you're back in regular portrait.

Animating the transition to landscape

The transition to the landscape view is a bit abrupt. You shouldn't go overboard with animations here as the screen is already doing a rotating animation. A simple crossfade will be sufficient.

- Change the `showLandscape(with:)` method in `SearchViewController.swift` as follows:

```
func showLandscape(with coordinator:  
    UIViewControllerTransitionCoordinator) {  
  
    if let controller = landscapeVC {  
        controller.view.frame = view.bounds  
        controller.view.alpha = 0 // New line  
  
        view.addSubview(controller.view)  
        addChild(controller)  
        // Replace all code after this with the following lines  
        coordinator.animate(alongsideTransition: { _ in  
            controller.view.alpha = 1  
        }, completion: { _ in  
            controller.didMove(toParent: self)  
        })  
    }  
}
```

You're still doing the same things as before, except now, the landscape view starts out completely transparent — `alpha = 0` — and slowly fades in while the rotation takes place until it's fully visible — `alpha = 1`.

Now you see why the `UIViewControllerTransitionCoordinator` object is needed — so your animation can be performed alongside the rest of the transition from the old traits to the new. This ensures the animations run as smoothly as possible.

The call to `animate(alongsideTransition:completion:)` takes two closures: the first is for the animation itself, the second is a “completion handler” that gets called after the animation finishes. The completion handler gives you a chance to delay the call to `didMove(toParent:)` until the animation is over.

Both closures are given a “transition coordinator context” parameter (the same context that animation controllers get) but you don't use it here and so, you use the `_` wildcard to ignore it.

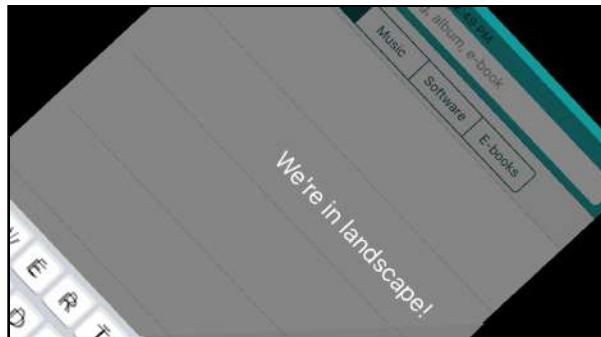
Animating the transition from landscape

- Make similar changes to `hideLandscape(with:)`:

```
func hideLandscape(with coordinator:  
    UIVViewControllerTransitionCoordinator) {  
    if let controller = landscapeVC {  
        controller.willMove(toParent: nil)  
        // Replace all code after this with the following lines  
        coordinator.animate(alongsideTransition: { _ in  
            controller.view.alpha = 0  
        }, completion: { _ in  
            controller.view.removeFromSuperview()  
            controller.removeFromParent()  
            self.landscapeVC = nil  
        })  
    }  
}
```

This time you fade out the view. You don't remove the view and the controller until the animation is completely done.

- Try it out. The transition between the portrait and landscape views should be a lot smoother now.



The transition from portrait to landscape

Tip: To see the transition animation in slow motion, select **Debug ▶ Slow Animations** from the Simulator menu.

Note: The order of operations for removing a child view controller is exactly the reverse of adding a child view controller, except for the calls to `willMove` and `didMove(toParent:)`.

The rules for view controller containment say that when adding a child view

controller, the last step is to call `didMove(toParent:)`. UIKit does not know when to call this method, as that needs to happen after any of your animations. You are responsible for sending the “did move to parent” message to the child view controller once the animation completes.

There is also a `willMove(toParent:)` but that gets called on your behalf by `addChild()` already, so you’re not supposed to do that yourself.

The rules are opposite when removing the child controller. First you should call `willMove(toParent: nil)` to let the child view controller know that it’s about to be removed from its parent. The child view controller shouldn’t actually be removed until the animation completes, at which point you call `removeFromParent()`. That method will then take care of sending the “did move to parent” message.

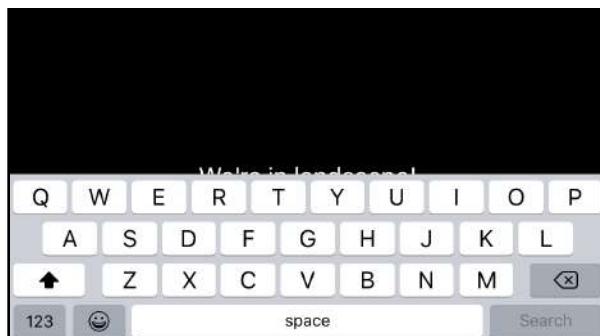
You can find these rules in the API documentation for `UIViewController`.

Fixing issues

There are two more small tweaks that you need to make.

Hiding the keyboard

Maybe you already noticed that when rotating the app while the keyboard is showing, the keyboard doesn’t go away.



The keyboard is still showing in landscape mode

Exercise: See if you can fix this yourself.

Answer: You've done something similar already after the user taps the Search button. The code is exactly the same here.

► Add the following line to `showLandscape(with:)`:

```
func showLandscape(with coordinator:  
    UIViewControllerTransitionCoordinator) {  
  
    coordinator.animate(alongsideTransition: { _ in  
        controller.view.alpha = 1  
        self.searchBar.resignFirstResponder() // Add this line  
    }, completion: { _ in  
        . . .  
    })  
}
```

Now the keyboard disappears as soon as you rotate the device. It looks best if you call `resignFirstResponder()` inside the `animate-alongside-transition` closure. After all, hiding the keyboard also happens with an animation.

Hiding the Detail pop-up

Speaking of things that stay visible, what happens when you tap a row in the table view and then rotate to landscape? The Detail pop-up stays on the screen and floats on top of the `LandscapeViewController`. That's a little strange. It would be better if the app dismissed the pop-up before rotating.

Exercise: See if you can fix that one.

The Detail pop-up is presented modally via a segue, so you can call `dismiss(animated:completion:)` to dismiss it, just like you do in the `close()` action method.

There's a complication though: you should only dismiss the Detail screen when it is actually visible. For that, you can look at the `presentedViewController` property. This returns a reference to the current modal view controller, if any. If `presentedViewController` is `nil` there isn't anything to dismiss.



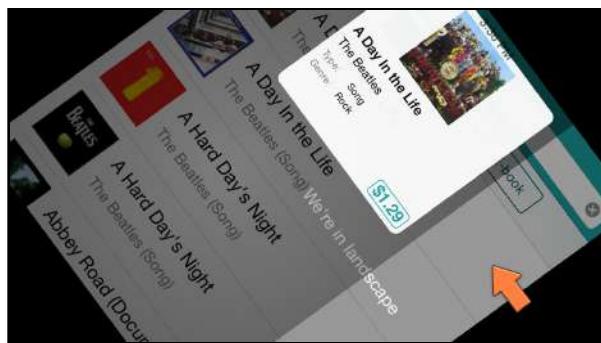
- Add the following code to the end of the `animate(alongsideTransition:)` closure in `showLandscape(with:)`:

```
if self.presentedViewController != nil {  
    self.dismiss(animated: true, completion: nil)  
}
```

- Run the app and tap on a search result, then rotate to landscape. The pop-up should now fly off the screen. When you return to portrait, the pop-up is nowhere to be seen.

Fixing the gradient view

If you look really carefully while the screen rotates, you can see a glitch at the right side of the screen. The gradient view doesn't appear to stretch to fill up the extra space:



There is a gap next to the gradient view

Press `⌘+T` to turn on slow animations in the Simulator so you can clearly see this happening. But do this only after the Detail pop-up has appeared. Some (all?) of the animation completions don't appear to fire with slow animations on and so you might not get the expected behavior otherwise.

It's only a small detail, but we can't have such imperfections in our app!

The solution is to pin the `GradientView` to the edges of the window so that it will always stretch along with the window. But you didn't create `GradientView` in Interface Builder... how do you give it constraints?

It is possible to create constraints in code, using the `NSLayoutConstraint` class, but there is an easier solution: you can simply change the `GradientView`'s autoresizing behavior.

Autoresizing is what iOS developers used before Auto Layout existed. It's simpler to use but also less powerful. It's very easy to add autoresizing via code.

Using the `autoresizingMask` property, you can tell a view what it should do when its superview changes size. You have a variety of options, such as: do nothing, stick to a certain edge of the superview, or change in size proportionally.

The possibilities are much more limited than what you can do with Auto Layout, but for many scenarios, autoresizing is good enough.

The easiest place to set this autoresizing mask is in `GradientView`'s `init` methods.

- Add the following line to `init(frame:)` and `init?(coder:)` in `GradientView.swift`:

```
autoresizingMask = [.flexibleWidth, .flexibleHeight]
```

This tells the view that it should change both its width and its height proportionally when the superview it belongs to resizes due to being rotated, or for some other reason.

In practice, this means the `GradientView` will always cover the same area that its superview covers and there should be no more gaps, even if the device is rotated.

- Try it out! The gradient now always covers the whole screen.

Tweak the animation

The Detail pop-up flying up and out the screen looks a little weird in combination with the rotation animation. There's too much happening on the screen at once for my taste. Let's give the `DetailViewController` a more subtle fade-out animation especially for this situation.

When you tap the X button to dismiss the pop-up, you'll still make it fly out of the screen. But when it is automatically dismissed upon rotation, the pop-up will fade out with the rest of the table view instead.

You'll give `DetailViewController` a property that specifies how it will animate the pop-up's dismissal. You can use an enum for this.



- Add the following to **DetailViewController.swift**, *inside* the class:

```
enum AnimationStyle {
    case slide
    case fade
}

var dismissStyle = AnimationStyle.fade
```

This defines a new enum named `AnimationStyle`. An enum, or enumeration, is simply a list of possible values. The `AnimationStyle` enum has two values, `slide` and `fade`. Those are the animations the Detail pop-up can perform when dismissed.

The `dismissStyle` variable determines which animation is chosen. This variable is of type `AnimationStyle`, so it can only contain one of the values from that enum. By default it is `.fade`, the animation that will be used when rotating to landscape.

Note: The full name of the enum is `DetailViewController.AnimationStyle` because it sits inside the `DetailViewController` class.

It's a good idea to keep the things that are closely related to a particular class, such as this enum, inside the definition for that class. That puts them inside the class's *namespace*.

Doing this allows you to also add a completely different `AnimationStyle` enum to one of the other view controllers, without running into naming conflicts.

- In the `close()` method, set the animation style to `.slide`, so that it keeps using the animation you're already familiar with:

```
@IBAction func close() {
    dismissStyle = .slide // Add this line
    dismiss(animated: true, completion: nil)
}
```

- Add a new **Swift File** to the project, named **FadeOutAnimationController**. This will handle the animation for the `.fade` style.

- Replace the source code of this new file with:

```
import UIKit

class FadeOutAnimationController: NSObject, UIVViewControllerAnimatedTransitioning {
    func transitionDuration(using transitionContext: UIVViewControllerContextTransitioning?) -> TimeInterval {
        return 0.4
    }

    func animateTransition(using transitionContext: UIVViewControllerContextTransitioning) {
        if let fromView = transitionContext.view(forKey: UITransitionContextViewKey.from) {
            let time = transitionDuration(using: transitionContext)
            UIView.animate(withDuration: time, animations: {
                fromView.alpha = 0
            }, completion: { finished in
                transitionContext.completeTransition(finished)
            })
        }
    }
}
```

This is mostly the same as the other animation controllers. The actual animation simply sets the view's alpha value to 0 in order to fade it out.

- Switch to **DetailViewController.swift** and in the extension for the transitioning delegate, change the method that returns the animation controller for dismissing the pop-up to the following:

```
func animationController(forDismissed dismissed: UIVViewController) -> UIVViewControllerAnimatedTransitioning? {
    switch dismissStyle {
    case .slide:
        return SlideOutAnimationController()
    case .fade:
        return FadeOutAnimationController()
    }
}
```

Instead of always returning a new `SlideOutAnimationController` instance, it now looks at the value from `dismissStyle`. If it is `.fade`, then it returns an instance of the new `FadeOutAnimationController` object.

- Run the app, bring up the Detail pop-up and rotate to landscape. The pop-up should now fade out while the landscape view fades in — enable slow animations to clearly see what is going on.





The pop-up fades out instead of flying away

And that does it. If you want to create more animations that can be used on dismissal, you only have to add a new value to the `AnimationStyle` enum and check for it in the `animationController(forDismissed:)` method. And build a new animation controller, of course.

That concludes the first version of the landscape screen. It doesn't do much yet, but it's already well integrated with the rest of the app. That's worthy of a commit, methinks.

Adding a scroll view

If an app has more content to show than can fit on the screen, you can use a *scroll view*, which allows the user to, as the name implies, scroll through the content horizontally and/or vertically.

You've already been working with scroll views all this time without knowing it: the `UITableView` object extends from `UIScrollView`.

In this section, you'll use a scroll view of your own, in combination with a *paging control*, to show the artwork for all the search results, even if there are more images than can fit on the screen at once.

Adding the scrollview to the storyboard

- Open the storyboard and delete the label from the Landscape scene.
- Now, drag a **Scroll View** into the scene and set it to completely cover the screen — 667 x 375 if you're using the iPhone 8 layout.

- Drag a new **Page Control** object into the scene — make sure you pick Page Control and *not* Page View Controller.

This gives you a small view with three white dots. Place it bottom center. The exact location doesn't matter because you'll move it to the right position later.

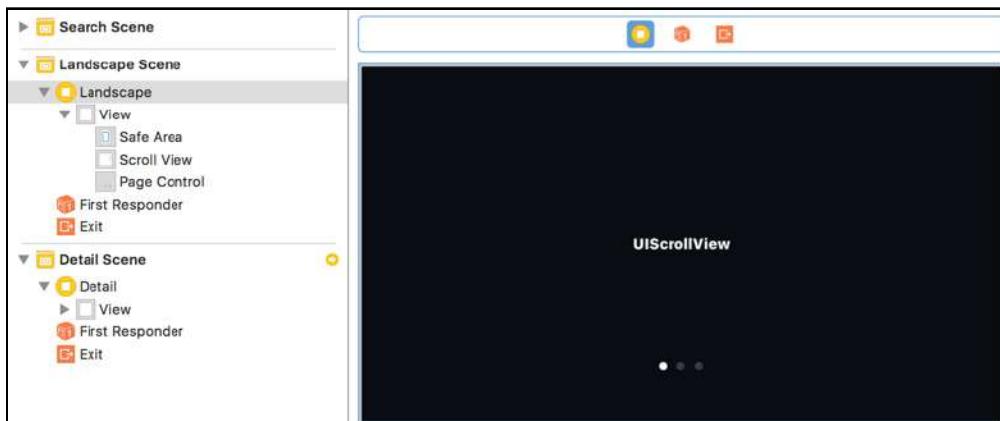
Important: Do not place the Page Control *inside* the Scroll View. They should be at the same level in the view hierarchy:



The Page Control should be a “sibling” of the ScrollView, not a child

If you did drop your Page Control inside the ScrollView instead of on top, then you can rearrange it in the Document Outline.

That concludes the design of the Landscape scene. The rest you will do in code, not in Interface Builder.



The final design of the Landscape scene

Disabling Auto Layout for a view controller

The other view controllers you've created all employ Auto Layout to resize them to the dimensions of the user's screen, but here, you're going to take a different approach. Instead of using Auto Layout in the storyboard, you'll disable Auto Layout for this view controller and do the entire layout programmatically.

You do need to hook up the controls to outlets, of course.

- Add these outlets to **LandscapeViewController.swift**, and connect them in Interface Builder:

```
@IBOutlet weak var scrollView: UIScrollView!
@IBOutlet weak var pageControl: UIPageControl!
```

Next up you'll disable Auto Layout for this view controller. The storyboard has a "Use Auto Layout" checkbox but you cannot use that. It would turn off Auto Layout for all the view controllers, not just this one.

- Replace **LandscapeViewController.swift**'s `viewDidLoad()` with:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Remove constraints from main view
    view.removeConstraints(view.constraints)
    view.translatesAutoresizingMaskIntoConstraints = true
    // Remove constraints for page control
    pageControl.removeConstraints(pageControl.constraints)
    pageControl.translatesAutoresizingMaskIntoConstraints = true
    // Remove constraints for scroll view
    scrollView.removeConstraints(scrollView.constraints)
    scrollView.translatesAutoresizingMaskIntoConstraints = true
}
```

Remember how, if you don't add constraints of your own, Interface Builder will give the views automatic constraints? Well, those automatic constraints get in the way if you're going to do your own layout. That's why you need to remove these unwanted constraints from all the visible views in the view controller first.

You also do `translatesAutoresizingMaskIntoConstraints = true`. This allows you to position and size your views manually by changing their `frame` property.

When Auto Layout is enabled, you’re not really supposed to change the `frame` yourself — you can only indirectly move views into position by creating constraints. Modifying the `frame` by hand can cause conflicts with the existing constraints and bring all sorts of trouble — you don’t want to make Auto Layout angry, you wouldn’t like it when it’s angry.

For this view controller, it’s much more convenient to manipulate the `frame` property directly than it is making constraints — especially when you’re placing the buttons for the search results — which is why you’re disabling Auto Layout.

Note: Auto Layout doesn’t really get disabled, but with the “translates autoresizing mask” option set to true, UIKit will convert your manual layout code into the proper constraints behind the scenes. That’s also why you removed the automatic constraints because they will conflict with the new ones, possibly causing your app to crash.

Custom scroll view layout

Now that Auto Layout is out of the way, you can do your own layout. That happens in the `viewWillLayoutSubviews()` method.

► Add this new method:

```
override func viewWillLayoutSubviews() {
    super.viewWillLayoutSubviews()
    let safeFrame = view.safeAreaLayoutGuide.layoutFrame
    scrollView.frame = safeFrame
    pageControl.frame = CGRect(x: safeFrame.origin.x,
        y: safeFrame.size.height - pageControl.frame.size.height,
        width: safeFrame.size.width,
        height: pageControl.frame.size.height)
}
```

The `viewWillLayoutSubviews()` method is called by UIKit as part of the layout phase of your view controller when it first appears on screen. It’s the ideal place for changing the frames of your views by hand.

The scroll view should always be as large as the entire screen, so you would think that you should make its `frame` equal to the main view’s bounds. This used to be the case till Apple introduced the iPhone X. But things change ...

With the iPhone X, you had to make sure that your content did not appear where the iPhone X’s notch was, or where the scroll bar appeared at the bottom of the screen.



So, Apple introduced the *safe area* concept — iOS would tell you what parts of a view were safe to have content on and each view would have several properties which defined the safe area for that view.

We make use of the `safeAreaLayoutGuide` property of the main view to get its `layoutFrame` — the safe area for the view in its own coordinate system — and then use that to set up the scroll view and the page control.

The page control is located at the bottom of the screen, and spans the entire width of the safe area. If this calculation doesn't make any sense to you, then try to sketch what happens on a piece of paper.

Note: If you're confused about how the layout looks/works, one easy way to get a better understanding is to set the background color of the scroll view and the page control to two distinctive colors like yellow and red and then run the app.

You will now see each control's actual content area in different colors against the black background and show you how each view is laid out.

You will find that this is a good technique to use in debugging any view positioning/sizing related issue.

- Run the app and flip to landscape. Nothing much happens yet: the screen has the page control at the bottom (the dots) but it still mostly black.

Add a background to the view

Let's make the view a little less plain by adding a background to it.

- Add the following line to `viewDidLoad()`:

```
view.backgroundColor = UIColor(patternImage:  
    UIImage(named: "LandscapeBackground")!)
```

This puts an image as the main view's background. An image? But you're setting the `backgroundColor` property, which is a `UIColor`, not a `UIImage`! Yup, that's true, but `UIColor` has a cool trick that lets you use a tileable image as a color.

If you take a peek at the **LandscapeBackground** image in the asset catalog, you'll see that it is a small square. When you set this image as a pattern image for the background, the image repeats to cover the entire area. Tileable images can be used anywhere where you can use a `UIColor`.



You might be tempted to set the background for the scroll view instead of the main view and for most iOS devices, that would work just as well. In fact, it would work better in the case of the scroll view because when you scroll the view, the background would animate.

However, on an iPhone X, if you set the image as the background for the scroll view, you'll notice that it doesn't cover the whole screen. This is again due to that pesky safe area.

Try it for yourself and see the difference.

And if you have the bright idea of setting the background of the main view *and* the background of the scroll view to the same image in the hopes of having a seamless background that scrolls - try that too and see what happens.

Set the Scroll View content size

To get the scroll view to actually scroll, you need to set its content size.

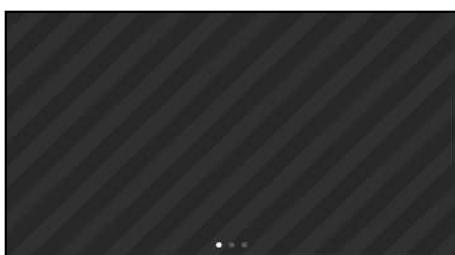
► Add the following line to `viewDidLoad()`:

```
scrollView.contentSize = CGSize(width: 1000, height: 1000)
```

It is very important to set the `contentSize` property when dealing with scroll views. This tells the scroll view how big the content area for the scroll view is — a scroll view's inside (the content area), can be bigger than its actual bounds. If the content area is bigger than the scroll view's bounds, that's when the scroll view allows you to scroll.

People often forget this step and then they wonder why their scroll view doesn't scroll. Unfortunately, you cannot set `contentSize` from Interface Builder, so it must be done from code.

► Run the app and try some scrolling:



The scroll view now has a background image and it can scroll

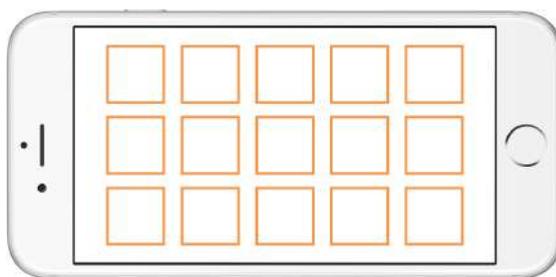
You might not notice too much of a difference since the background is static, but if you pay close attention, you'll notice that the horizontal and vertical scroll bars do move as you scroll around.

If the dots at the bottom also move while scrolling, then you've placed the page control inside the scroll view. Open the storyboard and in the Document Outline drag the Page Control below the Scroll View.

The page control itself doesn't do anything yet. Before you can make that work, you first have to add some content to the scroll view.

Adding result buttons

The idea is to show the search results in a grid:



Each of these results is really a button. Before you can place these buttons on the screen, you need to calculate how many will fit on the screen at once. Easier said than done, because different iPhone models have different screen sizes.

Time for some math! Let's assume the app runs on a 4-inch device. In that case, the scroll view is 667 points wide by 375 points tall. It can fit 3 rows of 6 columns if you put each search result in a rectangle of 94 by 88 points. That comes to $3 \times 6 = 18$ search results on the screen at once. A search may return up to 200 results.

Obviously, there is not enough room for everything and you will have to spread out the results over several pages.

One page contains 18 buttons. For the maximum number of results you will need $200 / 18 = 11.1111$ pages, which rounds up to 12 pages. That last page will only be filled partially.

The 4.7-inch iPhone models have room for 7 columns plus some leftover vertical space, and the 5.5-inch iPhone Plus models can fit yet another column plus an extra row.

That's a lot of different possibilities!

You need to add the logic to `LandscapeViewController` so it can calculate how big the scroll view's `contentSize` has to be. It will also need to add a `UIButton` object for each search result.

Once you have that working, you can display the artwork via that `UIButton`.

Of course, this means the app first needs to pass the array of search results to `LandscapeViewController` so it can use them for its calculations.

Passing the search results to the landscape view

► Let's add a property for this to `LandscapeViewController.swift`:

```
var searchResults = [SearchResult]()
```

Initially, this will be an empty array. `SearchViewController` replaces it with the real array upon rotation to landscape.

► Assign the array to the new property in `SearchViewController.swift`:

```
func showLandscape(with coordinator:  
    UIViewControllerTransitionCoordinator) {  
  
    if let controller = landscapeVC {  
        controller.searchResults = searchResults // add this line  
        . . .
```

You have to be sure to set `searchResults` before you access the `view` property from the `LandscapeViewController`, because that will trigger the view to be loaded and call `viewDidLoad()`.

The view controller will read from the `searchResults` array in `viewDidLoad()` to build up the contents of its scroll view. But if you access `controller.view` before setting `searchResults`, this property will still be `nil` and no buttons will be created. The order in which you do things matters here!

► Switch back to `LandscapeViewController.swift`. Remove the line that sets `scrollView.contentSize` from `viewDidLoad()`. That was just for testing.



Now let's go make those buttons.

Initial configuration

- Add a new instance variable:

```
private var firstTime = true
```

The purpose for this variable will become clear in a moment.

Private parts

You declared the `firstTime` instance variable as `private`. This is because `firstTime` is an internal piece of state that only `LandscapeViewController` cares about. It should not be visible to other objects.

You don't want the other objects in your app to know about the existence of `firstTime`, or worse, actually try to use this variable. Strange things are bound to happen if some other view controller changes the value of `firstTime` when `LandscapeViewController` isn't expecting the change.

We haven't talked much about the distinction between *interface* and *implementation* yet, but what an object shows to the outside is different from what it has on the inside. That's done on purpose because its internals — the implementation details — should not be of interest to anyone else, and are often even dangerous to expose since messing around with internal settings can crash the app.

It is considered good programming practice to hide as much as possible inside the object and only show a few things on the outside. To make certain variables and methods invisible from outside of your own class, you declare them to be `private`. That removes them from the object's public interface.

Exercise: Find other variables and methods in the app that can be made `private`.

- Add the following lines to the end of `viewWillLayoutSubviews()`:

```
if firstTime {  
    firstTime = false  
    tileButtons(searchResults)  
}
```

This calls a new method, `tileButtons(_:)`, that performs the necessary math and places the buttons on the screen in neat rows and columns. This needs to happen just once, when the `LandscapeViewController` is added to the screen.

You may think that `viewDidLoad()` would be a good place for this, but at the point in the view controller's lifecycle when `viewDidLoad()` is called, the view is not on the screen yet and has not been added into the view hierarchy. At this time, it doesn't know how large the view should be. Only after `viewDidLoad()` is done does the view get resized to fit the actual screen.

So you can't use `viewDidLoad()` for this. The only safe place to perform calculations based on the final size of the view — that is, any calculations that use the view's `frame` or `bounds` — is in `viewWillLayoutSubviews()`.

A warning: `viewWillLayoutSubviews()` may be invoked more than once! For example, it's also called when the landscape view gets removed from the screen. You use the `firstTime` variable to make sure you only place the buttons once.

Calculating the tile grid

- Add the new `tileButtons(_:)` method. It's a big 'un, so we'll take it piece-by-piece.

```
// MARK:- Private Methods
private func tileButtons(_ searchResults: [SearchResult]) {
    var columnsPerPage = 6
    var rowsPerPage = 3
    var itemWidth: CGFloat = 94
    var itemHeight: CGFloat = 88
    var marginX: CGFloat = 2
    var marginY: CGFloat = 20

    let viewWidth = scrollView.bounds.size.width

    switch viewWidth {
    case 568:
        // 4-inch device
        break

    case 667:
        // 4.7-inch device
        columnsPerPage = 7
        itemWidth = 95
        itemHeight = 98
        marginX = 1
        marginY = 29

    case 736:
```

```
// 5.5-inch device
columnsPerPage = 8
rowsPerPage = 4
itemWidth = 92
marginX = 0

case 724:
    // iPhone X
    columnsPerPage = 8
    rowsPerPage = 3
    itemWidth = 90
    itemHeight = 98
    marginX = 2
    marginY = 29

default:
    break
}

// TODO: more to come here
}
```

First, the method must decide on how big the grid squares should be and how many squares you need to fill up each page. There are four cases to consider, based on the width of the screen:

- **568 points**, 4-inch device. A single page fits 3 rows (`rowsPerPage`) of 6 columns (`columnsPerPage`). Each grid square is 94 by 88 points (`itemWidth` and `itemHeight`). The first row starts at Y = 20 (`marginY`). Because 568 doesn't evenly divide by 6, the `marginX` variable is used to adjust for the 4 points that are left over — 2 on each side of the page. All these are set as defaults at the top and so we don't need to do any changes for this case and we simply `break` out.
- **667 points**, 4.7-inch device. This still has 3 rows but 7 columns. Because there's some extra vertical space, the rows are higher (98 points) and there is a larger margin at the top.
- **736 points**, 5.5-inch device. This device is huge and can house 4 rows of 8 columns.
- **724 points**, iPhone X. This device is thinner than the 5.5-inch device, but is wider. However, the safe area takes out some of the available space and you end up with less space than the 5.5-inch device. So, it can only hold 3 rows by 8 columns and you still have to drop the cell width.

The variables at the top of the method keep track of all these measurements.



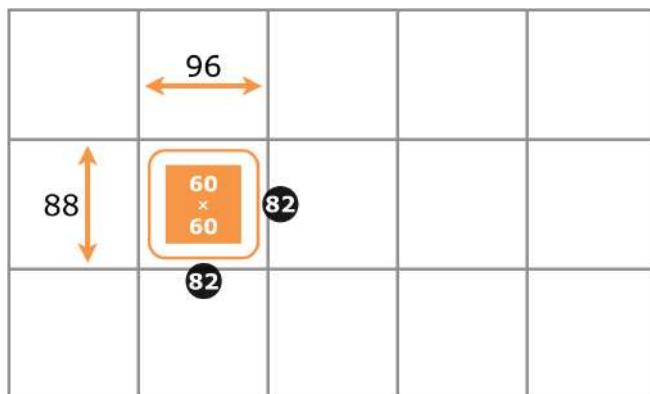
Note: Shouldn't it be possible to come up with a nice formula that calculates all this stuff for you, rather than *hard-coding* these sizes and margin values? Probably, but it won't be easy. There are two things you want to optimize for: getting the maximum number of rows and columns on the screen, but at the same time, not making the grid squares too small. Give it a shot if you think you can solve this puzzle!

From now on, you'll keep adding more code to the end of `tileButtons()` (where the TODO comment is) till the method is complete.

- Add the following lines to `tileButtons()`:

```
// Button size
let buttonWidth: CGFloat = 82
let buttonHeight: CGFloat = 82
let paddingHorz = (itemWidth - buttonWidth)/2
let paddingVert = (itemHeight - buttonHeight)/2
```

You've already determined that each search result gets a grid square of give-or-take 90 by 88 points (depending on the device), but that doesn't mean you need to make the buttons that big as well. The image you'll put on the buttons is 60×60 pixels, so that leaves quite a gap around the image. After playing with the design a bit, 82×82 points (`buttonWidth` and `buttonHeight`) seems to fit, leaving a small amount of padding between each button and its neighbors (`paddingHorz` and `paddingVert`).



The dimensions of the buttons in the 5x3 grid

Adding buttons

Now you can loop through the array of search results and make a new button for each `SearchResult` object.

- Add the following lines to `tileButtons()`:

```
// Add the buttons
var row = 0
var column = 0
var x = marginX
for (index, result) in searchResults.enumerated() {
    // 1
    let button = UIButton(type: .system)
    button.backgroundColor = UIColor.white
    button.setTitle("\(index)", for: .normal)
    // 2
    button.frame = CGRect(x: x + paddingHorz,
                          y: marginY + CGFloat(row)*itemHeight + paddingVert,
                          width: buttonWidth, height: buttonHeight)
    // 3
    scrollView.addSubview(button)
    // 4
    row += 1
    if row == rowsPerPage {
        row = 0; x += itemWidth; column += 1
        if column == columnsPerPage {
            column = 0; x += marginX * 2
        }
    }
}
```

Here is how this works:

1. Create the `UIButton` object. For debugging purposes, you give each button a title with the array index. If there are 200 results in the search, you also should end up with 200 buttons. Setting the index on the button will help to verify this.
2. When you make a button by hand, you always have to set its `frame`. Using the measurements you figured out earlier, you determine the position and size of the button. Notice that `CGRect`'s properties are all `CGFloat` but `row` is an `Int`. You need to convert `row` to a `CGFloat` before you can use it in the calculation.
3. You add the new button object to the `UIScrollView` as a subview. After the first 18 or so buttons (depending on the screen size), this places any subsequent buttons out of the visible range of the scroll view, but that's the whole point.

As long as you set the scroll view's `contentSize` accordingly, the user can scroll to view these other buttons.

4. You use the `x` and `row` variables to position the buttons, going from top to bottom (by increasing `row`). When you've reached the bottom (`row` equals `rowsPerPage`), you go up again to `row 0` and skip to the next column (by increasing the `column` variable).

When the `column` reaches the end of the screen (equals `columnsPerPage`), you reset it to 0 and add any leftover space to `x` (twice the X-margin).

Note that in Swift you can put multiple statements on a single line by separating them with a semicolon. It saves some space, but you can have those statements on separate lines, if you so prefer.

If this sounds like hocus pocus to you, you should play around a bit with these calculations to gain insight into how they work. It's not rocket science, but it does require some mental gymnastics. Tip: Sketching the process on paper can help!

Note: By the way, did you notice what happened in the `for` `in` loop?

```
for (index, result) in searchResults.enumerated() {
```

This `for...in` loop steps through the `SearchResult` objects from the array, but with a twist. By calling the array's `enumerated()` method, you get a *tuple* containing not only the next `SearchResult` object but also its index in the array.

A tuple is nothing more than a temporary list with two or more items in it. Here, the tuple is `(index, result)`. This is a neat trick to loop through an array and get both the objects and their indices.

- Finally, add the last part of this very long method:

```
// Set scroll view content size
let buttonsPerPage = columnsPerPage * rowsPerPage
let numPages = 1 + (searchResults.count - 1) / buttonsPerPage
scrollView.contentSize = CGSize(
    width: CGFloat(numPages) * viewWidth,
    height: scrollView.bounds.size.height)

print("Number of pages: \(numPages)")
```

At the end of the method you calculate the `contentSize` for the scroll view based on how many buttons fit on a page and the number of `SearchResult` objects.

You want the user to be able to “page” through these results — you’ll enable this feature shortly — rather than simply scroll. So, you should always make the content width a multiple of the scroll width (568, 667, 736, or 812 points). You can then determine how many pages you need with a simple formula.

Note: Dividing an integer value by an integer always results in an integer. If `buttonsPerPage` is 18 (3 rows × 6 columns) and there are fewer than 18 search results, `searchResults.count / buttonsPerPage` is 0.

It’s important to realize that `numPages` will never have a fractional value because all the variables involved in the calculation are `Ints`, which makes `numPages` an `Int` too.

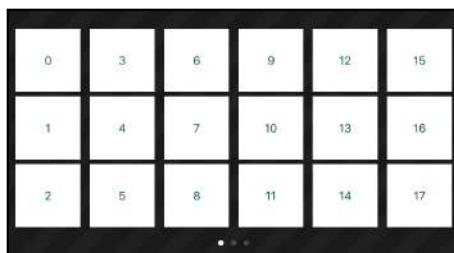
That’s why the formula is `1 + (searchResults.count - 1) / buttonsPerPage`.

If there are 18 results, exactly enough to fill a single page, `numPages = 1 + 17/18 = 1 + 0 = 1`. But if there are 19 results, the 19th result needs to go on the second page, and `numPages = 1 + 18/18 = 1 + 1 = 2`. Plug in some other values to verify this formula is correct.

There’s also a `print()` statement for good measure, so you can verify that you really end up with the right number of pages.

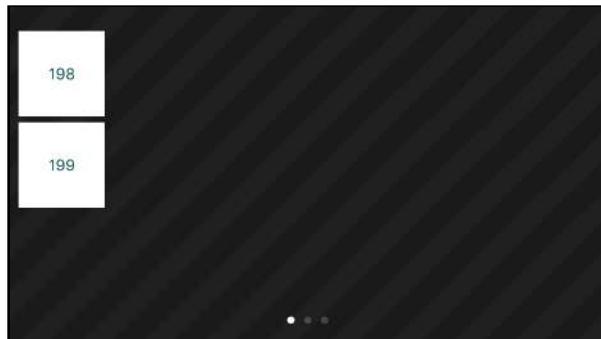
Note: Xcode currently gives a warning “Immutable value ‘result’ was never used; consider replacing with ‘’ or removing it.” *That warning will go away once you use the result variable in the next section.*

- ▶ Run the app, do a search, and rotate to landscape. You should now see a whole bunch of buttons:



The landscape view has buttons

Scroll all the way to the right and it looks like this on the iPhone 8:



The last page of the search results

That is 200 buttons indeed – you started counting at 0, remember? Just to make sure that this logic works properly, you should test a few different scenarios. What happens when there are fewer results than 18 – the amount that fit on a single page on 4-inch device? What happens when there are exactly 18 search results? How about 19, one more than can go on a single page? The easiest way to test these situations is to change the `&limit` parameter in the search URL.

Exercise: Try these situations for yourself and see what happens.

- Also test when there are no search results. The landscape view should now be empty. You'll add a “Nothing Found” label to this screen too, in a bit.

Paging

So far, the Page Control at the bottom of the screen has always shown three dots. And there wasn't much paging to be done on the scroll view either.

In case you're wondering what *paging* means: if the user has moved the scroll view a certain amount, it should snap to a new page.

With paging enabled, you can quickly flick through the contents of a scroll view, without having to drag it all the way. You're no doubt familiar with this effect because it is what the iPhone uses in its springboard. Many other apps use the effect too, for example, the Weather app uses paging to flip between the cards for different cities.

Enabling scroll view paging

- Go to **Landscape** scene in the storyboard and check the **Scrolling - Paging Enabled** option for the scroll view in the Attributes inspector.

There, that was easy! Now run the app and the scroll view will let you page rather than scroll. That's cool, but you also need to do something with the page control at the bottom of the screen.

Configuring the page control

- Switch to **LandscapeViewController.swift** and add this line to `viewDidLoad()`:

```
pageControl.numberOfPages = 0
```

This effectively hides the page control, which is what you want to do when there are no search results.

- Add the following lines to the end of `tileButtons()`:

```
pageControl.numberOfPages = numPages  
pageControl.currentPage = 0
```

This sets the number of dots that the page control displays to the number of pages that you calculated.

The active dot — the white one — needs to be synchronized with the active page in the scroll view. Currently, it never changes unless you tap in the page control and even then it has no effect on the scroll view.

To get this to work, you'll have to make the page control talk to the scroll view, and vice versa. The view controller must become the delegate of the scroll view so it will be notified when the user is flicking through the pages.

Connect the scroll view and page control

- Add this new extension to the end of **LandscapeViewController.swift**:

```
extension LandscapeViewController: UIScrollViewDelegate {  
    func scrollViewDidScroll(_ scrollView: UIScrollView) {  
        let width = scrollView.bounds.size.width  
        let page = Int((scrollView.contentOffset.x + width / 2)  
                      / width)  
        pageControl.currentPage = page
```



```
    }
```

This is a `UIScrollViewDelegate` method. You figure out what the index of the current page is by looking at the `contentOffset` property of the scroll view. This property determines how far the scroll view has been scrolled and is updated while you're dragging the scroll view.

Unfortunately, the scroll view doesn't simply tell us, "The user has flipped to page X." So, you have to calculate this yourself. If the content offset gets beyond halfway on the page (`width/2`), the scroll view will move to the next page. In that case, you update the `pageControl's` active page number.

You also need to know when the user taps on the Page Control so you can update the scroll view. There is no delegate for this, but you can use a regular `@IBAction` method for it.

► Add the action method:

```
// MARK:- Actions
@IBAction func pageChanged(_ sender: UIPageControl) {
    scrollView.contentOffset = CGPoint(
        x: scrollView.bounds.size.width *
        CGFloat(sender.currentPage), y: 0)
}
```

This works the other way around: when the user taps in the Page Control, its `currentPage` property gets updated. You use that to calculate a new `contentOffset` for the scroll view.

- In the storyboard, for the **Landscape** scene, **Control-drag** from the Scroll View to the view controller and select **delegate**.
- Also **Control-drag** from the Page Control to the view controller and select **pageChanged**: under Sent Events.
- Try it out, the page control and the scroll view should now be in sync.

The transition from one page to another after tapping in the page control is still a little abrupt, though. An animation would help here.

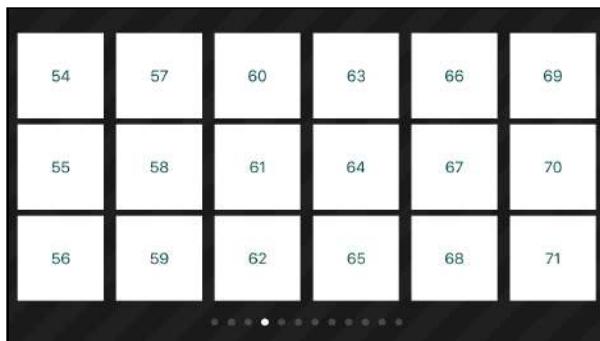
Exercise: See if you can animate what happens in `pageChanged(_ :)`.



You can simply wrap the code from the action method in an animation block:

```
@IBAction func pageChanged(_ sender: UIPageControl) {
    UIView.animate(withDuration: 0.3, delay: 0,
        options: [.curveEaseInOut], animations: {
            self.scrollView.contentOffset = CGPoint(
                x: self.scrollView.bounds.size.width *
                CGFloat(sender.currentPage), y: 0)
    },
    completion: nil)
}
```

You're using a version of the `UIView` animation method that allows you to specify options because the “Ease In, Ease Out” timing (`.curveEaseInOut`) looks good here.



We've got paging!

- This is a good time to commit.

Download the artwork

First, let's give the buttons a nicer look.

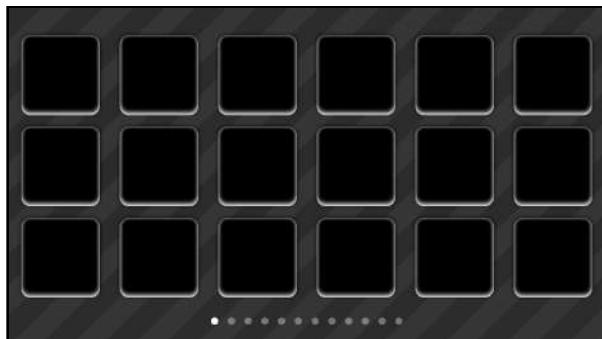
Set button background

- Replace the button creation code in `tileButtons()` (in `LandscapeViewController.swift`) with:

```
let button = UIButton(type: .custom)
button.setBackgroundImage(UIImage(named: "LandscapeButton"),
    for: .normal)
```

Instead of a regular button, you now make a `.custom` one, and you give it a background image instead of a white background and a title.

If you run the app, it will look like this:



The buttons now have a custom background image

Displaying button images

Now you have to download the artwork images, if they haven't already been downloaded and cached by the table view, and put them on the buttons.

Problem: You're dealing with `UIButtons` here, not `UIImageViews`, so you cannot simply use that handy extension from earlier. Fortunately, the code is very similar!

► Add a new method to `LandscapeViewController.swift`:

```
private func downloadImage(for searchResult: SearchResult,  
                           andPlaceOn button: UIButton) {  
    if let url = URL(string: searchResult.imageSmall) {  
        let task = URLSession.shared.downloadTask(with: url) {  
            [weak button] url, response, error in  
  
                if error == nil, let url = url,  
                    let data = try? Data(contentsOf: url),  
                    let image = UIImage(data: data) {  
                    DispatchQueue.main.async {  
                        if let button = button {  
                            button.setImage(image, for: .normal)  
                        }  
                    }  
                }  
            }  
        task.resume()  
    }  
}
```

This looks very much like what you did in the `UIImageView` extension. First you get a URL instance with the link to the 60×60-pixel artwork, and then you create a download task. Inside the completion handler you put the downloaded file into a

`UIImage`, and if all that succeeds, use `DispatchQueue.main.async` to place the image on the button.

- Add the following line to `tileButtons()` to call this new method, right after where you create the button:

```
downloadImage(for: result, andPlaceOn: button)
```

And that should do it. Run the app and you'll get some cool-looking buttons:



Showing the artwork on the buttons

Note: The Xcode warning about `result` is gone, but now it gives the same message for the `index` variable. Xcode doesn't like it if you declare variables but don't use them. You'll use `index` again later in this app but in the meantime, you can replace it by the `_` wildcard symbol to stop Xcode from complaining.

Clean up

It's always a good idea to clean up after yourself, in life as well as in programming. Imagine this: what would happen if the app is still downloading images and the user flips back to portrait mode?

At that point, the `LandscapeViewController` is deallocated but the image downloads keep going. That is exactly the sort of situation that can crash your app if not handled properly.

To avoid ownership cycles, you capture the button with a weak reference. When `LandscapeViewController` is deallocated, so are the buttons. So, the completion handler's captured button reference automatically becomes `nil`. The `if` let inside

the `DispatchQueue.main.async` block will now safely skip `button.setImage(for)`. No harm done. That's why you wrote `[weak button]`.

However, to conserve resources, the app should really stop downloading these images because they are not needed. Otherwise, it's just wasting bandwidth and battery life, and users don't take too kindly to apps that do this.

- Add a new property to `LandscapeViewController.swift`:

```
private var downloads = [URLSessionDownloadTask]()
```

This array will keep track of all the active `URLSessionDownloadTask` objects.

- Add the following line to the end of `downloadImage(for:andPlaceOn:)`, right after where you resume the download task:

```
downloads.append(task)
```

- And finally, add a `deinit` method to cancel any operations that are still on the way:

```
deinit {
    print("deinit \(self)")
    for task in downloads {
        task.cancel()
    }
}
```

This will stop the download for any button whose image was still pending or in transit. Good job, partner!

- Commit your changes.

Exercise: Despite what the iTunes web service promises, not all of the artwork is truly 60×60 pixels. Some of it is bigger, some are not even square, and so, it might not always fit nicely in the button. Your challenge is to use the image sizing code from *MyLocations* to always resize the image to 60×60 points before you put it on the button. Note that we're talking points here, not pixels — on Retina devices, the image should actually end up being 120×120 or even 180×180 pixels in size.

Note: In this section you learned how to create a grid-like view using a `UIScrollView`. iOS comes with a versatile class, `UICollectionView`, that lets you do the same thing — and much more! — without having to resort to the sort of math you did in `tileButtons()`. To learn more about `UICollectionView`, check out the website: raywenderlich.com/tag/collection-view

You can find the project files for this chapter under **44 – Landscape** in the Source Code folder.



Chapter 45: Refactoring

By Eli Ganim

Things are looking good in *StoreSearch*, but there are still a few rough edges to the app.

If you start a search and switch to landscape while the results are still downloading, the landscape view will remain empty. You can reproduce this situation by artificially slowing down your network connection using the Network Link Conditioner tool.

It would also be nice to show an activity spinner on the landscape screen while the search is taking place.

You will polish off some of these rough edges in this chapter and cover the following:

- **Refactor the search:** Refactor the code to put the search logic into its own class so that you have centralized access to the search state and results.
- **Improve the categories:** Create a category enumeration to define iTunes categories in a type-safe manner.
- **Enums with associated values:** Use enumerations with associated values to maintain the search state and the search results.
- **Spin me right round:** Add an activity indicator to the landscape view.
- **Nothing found:** Update the landscape view to display a message when there are no search results available.
- **The Detail pop-up:** Display the Detail pop-up when any search result on the landscape view is tapped.



Refactoring the search

So how can `LandscapeViewController` tell what state the search is in? Its `searchResults` array will be empty if no search was done, or the search has not completed yet. Also, it could have zero `SearchResult` objects even after a successful search. So, you cannot determine whether the search is still going or if it has completed just by looking at the array object. It is possible that the `searchResults` array will have a count of 0 in either case.

You need a way to determine whether a search is still going on. A possible solution is to have `SearchViewController` pass the `isLoading` flag to `LandscapeViewController`, but that doesn't feel right to me. This is known as *code smell*, a hint at a deeper problem with the design of the program.

Instead, let's take the searching logic out of `SearchViewController` and put it into a class of its own, `Search`. Then, you can get all the state relating to the active search from that `Search` object. Time for some more refactoring!

The Search class

- If you want, create a new branch for this in Git.

This is a pretty comprehensive change to the code and there is always a risk that it won't work as you hoped. By making the changes in a new branch, you can commit your changes without messing up the master branch. Plus, you can revert back to the master branch if the changes don't work out. Making new branches in Git is quick and easy, so it's good to get into the habit.

- Create a new file using the **Swift File** template. Name it **Search**.
- Change the contents of **Search.swift** to:

```
import Foundation

class Search {
    var searchResults: [SearchResult] = []
    var hasSearched = false
    var isLoading = false

    private var dataTask: URLSessionDataTask? = nil

    func performSearch(for text: String, category: Int) {
        print("Searching...")
    }
}
```

You've given this class three public properties, one private property, and a method. This stuff should look familiar because it comes straight from `SearchViewController`. You'll be removing code from that class and putting it into this new `Search` class.

The `performSearch(for:category:)` method doesn't do much yet but that's OK. First you need to make `SearchViewController` work with this new `Search` object and when it compiles without errors, you will move all the logic over. Baby steps!

Moving code over

Let's make the changes to `SearchViewController.swift`. Xcode will probably give a bunch of errors and warnings while you're making these changes, but it will all work out in the end.

- In `SearchViewController.swift`, remove the declarations for the following properties:

```
var searchResults: [SearchResult] = []
var hasSearched = false
var isLoading = false
var dataTask: URLSessionDataTask?
```

And replace them with this one:

```
private let search = Search()
```

The new `Search` object not only describes the state and results of the search, it will also encapsulate all the logic for talking to the iTunes web service. You can now remove a lot of code from the view controller.

- Move the following methods over to `Search.swift`:
 - `iTunesURL(searchText:category:)`
 - `parse(data:)`
- Make these methods `private`. They are only important to `Search` itself, not to any other classes from the app, so it's good to "hide" them.
- Back in `SearchViewController.swift`, replace the `performSearch()` method with the following (tip: set aside the old code in a temporary file because you'll need it again later).

```
func performSearch() {
    search.performSearch(for: searchBar.text!,
```

```
    category: segmentedControl.selectedSegmentIndex)

    tableView.reloadData()
    searchBar.resignFirstResponder()
}
```

This simply makes the `Search` object do all the work. Of course, you still reload the table view — to show the activity spinner — and hide the keyboard.

There are a few places in the code that still use the old `searchResults` array even though that no longer exists. You should change them to use the `searchResults` property from the `Search` object instead. Likewise for `hasSearched` and `isLoading`.

- For example, change `tableView(_:numberOfRowsInSection:)` to:

```
func tableView(_ tableView: UITableView,
              numberOfRowsInSection section: Int) -> Int {
    if search.isLoading {
        return 1 // Loading...
    } else if !search.hasSearched {
        return 0 // Not searched yet
    } else if search.searchResults.count == 0 {
        return 1 // Nothing Found
    } else {
        return search.searchResults.count
    }
}
```

Similar to the above, find the other places in code where the relevant properties have moved and make the necessary changes. If you aren't sure of where to make the changes, look for Xcode errors — for this step, once you make all the changes correctly, the code will compile again without any errors.

- In `showLandscape(with:)`, change the line that sets the `searchResults` property on the new view controller from:

```
controller.searchResults = search.searchResults
```

To:

```
controller.search = search
```

This line will give an error after you make the change, but you'll fix that next.

The `LandscapeViewController` still has a property for a `searchResults` array so you have to change that to use the `Search` object as well.

- In **LandscapeViewController.swift**, remove the `searchResults` instance variable and replace it with:

```
var search: Search!
```

- In `viewWillLayoutSubviews()`, change the call to `tileButtons()` into:

```
tileButtons(search.searchResults)
```

OK, that's the first round of changes. Build the app to make sure there are no compiler errors.

Adding the search logic back in

The app itself doesn't do much anymore because you removed all the searching logic. So let's put that back in.

- In **Search.swift**, replace `performSearch(for:category:)` with the following (you can use that temporary file from earlier, but be careful to make the proper changes):

```
func performSearch(for text: String, category: Int) {
    if !text.isEmpty {
        dataTask?.cancel()

        isLoading = true
        hasSearched = true
        searchResults = []

        let url = iTunesURL(searchText: text, category: category)

        let session = URLSession.shared
        dataTask = session.dataTask(with: url, completionHandler: {
            data, response, error in
            // Was the search cancelled?
            if let error = error as NSError?, error.code == -999 {
                return
            }

            if let httpResponse = response as? HTTPURLResponse,
               httpResponse.statusCode == 200, let data = data {
                self.searchResults = self.parse(data: data)
                self.searchResults.sort(by: <)

                print("Success!")
                self.isLoading = false
                return
            }
        })
    }
}
```

```
    print("Failure! \(response!)")
    self.hasSearched = false
    self.isLoading = false
)
dataTask?.resume()
}
```

This is basically the same logic as before, except all the user interface code has been removed. The purpose of `Search` is just to perform a search, it should not do any UI stuff. That's the job of the view controller.

- Run the app and search for something. When the search finishes, the Console shows a “Success!” message but the table view does not reload and the spinner keeps spinning for eternity.

The `Search` object currently has no way to tell the `SearchViewController` that it is done. You could solve this by making `SearchViewController` a delegate of the `Search` object, but for situations like these, closures are much more convenient.

The SearchComplete closure

Let's create your own closure!

- Add the following line to `Search.swift`, above the `class` line:

```
typealias SearchComplete = (Bool) -> Void
```

The `typealias` declaration allows you to create a more convenient name for a data type, in order to save some keystrokes and to make the code more readable.

Here, you declare a type for your own closure, named `SearchComplete`. This is a closure that returns no value (it is `Void`) and takes one parameter, a `Bool`. If you think this syntax is weird, then I'm right there with you, but that's the way it is.

From now on, you can use the name `SearchComplete` to refer to a closure that takes a `Bool` parameter and returns no value.

Closure types

Whenever you see a `->` in a type definition, the type is intended for a closure, function, or method.

Swift treats these three things as mostly interchangeable. Closures, functions, and methods are all blocks of source code that possibly take parameters and return a



value. The difference is that a function is really just a closure with a name, and a method is a function that lives inside an object.

Some examples of closure types:

- `() -> ()` is a closure that takes no parameters and returns no value.
- `Void -> Void` is the same as the previous example. `Void` and `()` mean the same thing.
- `(Int) -> Bool` is a closure that takes one parameter, an `Int`, and returns a `Bool`.
- `Int -> Bool` is the same as the above. If there is only one parameter, you can leave out the parentheses.
- `(Int, String) -> Bool` is a closure taking two parameters, an `Int` and a `String`, and returning a `Bool`.
- `(Int, String) -> Bool?` as above, but now returns an optional `Bool` value.
- `(Int) -> (Int) -> Int` is a closure that returns another closure that returns an `Int`. Freaky! Swift treats closures like any other type of object, so you can also pass them as parameters and return them from functions.

► Make the following changes to `performSearch(for:category:)`:

```
func performSearch(for text: String, category: Int,
                   completion: @escaping SearchComplete) {           // new
    if !text.isEmpty {
        .
        .
        .
        dataTask = session.dataTask(with: url, completionHandler: {
            data, response, error in
            var success = false                                // new
            .
            .
            if let httpResponse = response as? . . . {
                .
                .
                self.isLoading = false
                success = true                               // instead of return
            }

            if !success {                                     // new
                self.hasSearched = false
                self.isLoading = false
            }
        }                                                // new
        // New code block – add the next three lines
        DispatchQueue.main.async {
            completion(success)
        }
    }
    dataTask?.resume()
```

```
    }
```

You've added a third parameter named `completion` that is of type `SearchComplete`. Whoever calls `performSearch(for:category:completion:)` can now supply their own closure, and the method will execute the code that is inside that closure when the search completes.

Note: The `@escaping` annotation is necessary for closures that are not used immediately. It tells Swift that this closure may need to capture variables such as `self` and keep them around for a little while until the closure can finally be executed, in this case, when the search is done.

Instead of returning early from the closure upon success, you now set the `success` variable to `true` replacing the `return` statement. The value of `success` is used for the `Bool` parameter of the `completion` closure, as you can see inside the call to `DispatchQueue.main.async` at the bottom.

To perform the code from the closure, you simply call it as you'd call any function or method: `closureName(parameters)`. You call `completion(true)` upon success and `completion(false)` upon failure. This is done so that the `SearchViewController` can reload its table view or, in the case of an error, show an alert view.

► In `SearchViewController.swift`, replace `performSearch()` with:

```
func performSearch() {
    search.performSearch(for: searchBar.text!,
        category: segmentedControl.selectedSegmentIndex,
        completion: { success in          // Begin new code
            if !success {
                self.showNetworkError()
            }
            self.tableView.reloadData()
        })                                // End new code

    tableView.reloadData()
    searchBar.resignFirstResponder()
}
```

You now pass a closure to `performSearch(for:category:completion:)`. The code in this closure gets called after the search completes, with the `success` parameter being either `true` or `false`. A lot simpler than making a delegate, right? The closure is always called on the main thread, so it's safe to use UI code here.

- Run the app. You should be able to search again.

That's the first part of this refactoring complete. You've extracted the relevant code for searching out of the `SearchViewController` and placed it into its own object, `Search`. The view controller now only does view-related things, which is exactly how it is supposed to work.

- You've made quite a few extensive changes, so it's a good idea to commit.

Improving the categories

The idea behind Swift's strong typing is that the data type of a variable should be as descriptive as possible. Right now, the category to search for is represented by a number, 0 to 3, but is that the best way to describe a category to your program?

If you see the number 3, does that mean "e-book" to you? It could be anything... And what if you use 4 or 99 or -1, what would that mean? These are all valid values for an `Int` but not for a category. The only reason the category is currently an `Int` is because `segmentedControl.selectedIndex` is an `Int`.

Representing the category as an enum

There are only four possible search categories, so this sounds like a job for an enum!

- Add the following to `Search.swift`, *inside* the class brackets:

```
enum Category: Int {  
    case all = 0  
    case music = 1  
    case software = 2  
    case ebooks = 3  
}
```

This creates a new enumeration type named `Category` with four possible values. Each of these has a numeric value associated with it, called the **raw value**.

Contrast this with the `AnimationStyle` enum you made before:

```
enum AnimationStyle {  
    case slide  
    case fade  
}
```

That enum does not associate numbers with its values — it doesn't say : Int behind the enum name. For `AnimationStyle` it doesn't matter that `slide` is really number 0 and `fade` is number 1, or whatever the values might be. All you care about is that a variable of type `AnimationStyle` can either be `.slide` or `.fade`, a numeric value is not important.

For the `Category` enum, however, you want to connect its four values to the four possible indices of the Segmented Control. If segment 3 is selected, you want this to correspond to `.ebooks`. That's why the items from the `Category` enum have associated numbers.

Using the Category enum

- ▶ Change the method signature of `performSearch(for:category:completion:)` to use this new type:

```
func performSearch(for text: String, category: Category,  
                   completion: @escaping SearchComplete) {
```

The `category` parameter is no longer an `Int`. It is not possible to pass it the value 4 or 99 or -1 anymore. It must always be one of the values from the `Category` enum. This reduces a potential source of bugs and it has made the program more expressive. Whenever you have a limited list of possible values that can be turned into an enum, it's worth doing!

- ▶ Also change `iTunesURL(searchText:category:)` because that also assumed `category` would be an `Int`:

```
private func iTunesURL(searchText: String,  
                      category: Category) -> URL {  
    let kind: String  
    switch category {  
        case .all: kind = ""  
        case .music: kind = "musicTrack"  
        case .software: kind = "software"  
        case .ebooks: kind = "ebook"  
    }  
  
    let encodedText = . . .
```

The `switch` now looks at the various cases from the `Category` enum instead of the numbers 0 to 3. Note that the `default` case is no longer needed because the `category` parameter cannot have any other values.

This code works, but to be honest I'm not entirely happy with it. I've said before that any logic that is related to an object should be an integral part of that object. In other words, an object should do as much as it can itself.

Converting the category into a “kind” string that goes into the iTunes URL is a good example. That sounds like something the `Category` enum itself could do.

Swift enums can have their own methods and properties. So, let's take advantage of that and improve the code even more.

► Add the `type` property to the `Category` enum:

```
enum Category: Int {
    case all = 0
    case music = 1
    case software = 2
    case ebooks = 3

    var type: String {
        switch self {
            case .all: return ""
            case .music: return "musicTrack"
            case .software: return "software"
            case .ebooks: return "ebook"
        }
    }
}
```

Swift enums cannot have instance variables, only computed properties. `type` has the exact same `switch` statement that you just saw, except that it switches on `self`, the current value of the enumeration object.

► In `iTunesURL(searchText:category:)` you can now simply write:

```
private func iTunesURL(searchText: String,
                      category: Category) -> URL {
    let kind = category.type
    let encodedText = . . .
```

That's a lot cleaner. Everything that has to do with categories now lives inside its own enum, `Category`.

Converting an Int to Category

You still need to tell `SearchViewController` about this, because it needs to convert the selected segment index into a proper `Category` value.

- In **SearchViewController.swift**, change the first part of `performSearch()` to:

```
func performSearch() {
    if let category = Search.Category(
        rawValue: segmentedControl.selectedSegmentIndex) {
        search.performSearch(for: searchBar.text!,
            category: category, completion: {
                . .
            })
        .
    }
}
```

To convert the `Int` value from `selectedSegmentIndex` to an item from the `Category` enum, you use the built-in `init(rawValue:)` method. This may fail — for example, when you pass in a number that isn't covered by one of `Category`'s cases, i.e. anything that is outside the range 0 to 3. That's why `init(rawValue:)` returns an optional that needs to be unwrapped with `if let` before you can use it.

Note: Because you placed the `Category` enum inside the `Search` class, its full name is `Search.Category`. In other words, `Category` lives inside the `Search` *namespace*. It makes sense to bundle up these two things because they are so closely related.

- Build and run to see if the different categories still work.

Enums with associated values

Enums are pretty useful for restricting something to a limited range of possibilities, like what you did with the search categories. But they are even more powerful than you might have expected, as you'll find out...

Like all objects, the `Search` object has a certain amount of *state*. For `Search`, this is determined by its `isLoading`, `hasSearched`, and `searchResults` variables.

These three variables describe four possible states:

State	hasSearched	isLoading	searchResults
No search has been performed yet (this is also the state after an error)	false	false	Empty array
The search is in progress	true	true	Empty array
No results were found	true	false	Empty array
There are search results	true	false	Contains at least one SearchResult object

The Search object is in only one of these states at a time, and when it changes from one state to another, there is a corresponding change in the app's UI. For example, upon a change from “searching” to “have results,” the app hides the activity spinner and loads the results into the table view.

The problem is that this state is scattered across three different variables. It's tricky to see what the current state is just by looking at these variables.

Consolidate search state

You can improve upon things by giving Search an explicit state variable. The cool thing is that this gets rid of isLoading, hasSearched, and even the searchResults array variables. Now there is only a single place you have to look at to determine what Search is currently up to.

► In **Search.swift**, remove the following instance variables:

```
var searchResults: [SearchResult] = []
var hasSearched = false
var isLoading = false
```

► In their place, add the following enum, which goes inside the class again:

```
enum State {
    case notSearchedYet
    case loading
    case noResults
    case results([SearchResult])
}
```

This enumeration has a case for each of the four states listed above. It does not need raw values, so the cases don't have numbers — do note that the state .notSearchedYet is also used for when there is an error.

The `.results` case is special: it has an *associated value* — an array of `SearchResult` objects.

This array is only important when the search is successful. In all the other cases, there are no search results and the array is empty — see the state table above. By making it an associated value, you'll only have access to this array when `Search` is in the `.results` state. In the other states, the array simply does not exist.

Using the new state enum

Let's see how this works.

- First add a new instance variable:

```
private(set) var state: State = .notSearchedYet
```

This keeps track of `Search`'s current state. Its initial value is `.notSearchedYet` — obviously no search has happened yet when the `Search` object is first constructed.

This variable is `private`, but only half so. It's not unreasonable for other objects to want to ask `Search` what its current state is. In fact, the app won't work unless you allow this.

But you don't want those other objects to be able to *change* the value of `state`; they are only allowed to read the state value. With `private(set)` you tell Swift that reading is OK for other objects, but assigning (or setting) new values to this variable may only happen inside the `Search` class.

- Change `performSearch(for:category:completion:)` to use this new variable:

```
func performSearch(for text: String, category: Category,
                   completion: @escaping SearchComplete) {
    if !text.isEmpty {
        dataTask?.cancel()
        // Remove the next 3 lines and replace with the following
        state = .loading

        dataTask = session.dataTask(with: url, completionHandler: {
            data, response, error in

            var newState = State.notSearchedYet           // add this

            if let httpResponse = response . . . {
                // Replace all code within this if block with following
                var searchResults = self.parse(data: data)
                if searchResults.isEmpty {
                    newState = .noResults
                }
            }
        })
    }
}
```

```
        } else {
            searchResults.sort(by: <)
            newState = .results(searchResults)
        }
        success = true
    }
    // Remove "if !success" block
    DispatchQueue.main.async {
        self.state = newState                         // add this
        completion(success)
    }
}
dataTask?.resume()
}
```

Instead of the old variables `isLoading`, `hasSearched`, and `searchResults`, this code now only changes `state`.

Note: You don't update `state` directly, but instead, use a new local variable `newState`. Then at the end, in the `DispatchQueue.main.async` block, you transfer the value of `newState` to `self.state`. The reason for doing this the long way round is that `state` must only be changed by the main thread, or it can lead to a nasty and unpredictable bug known as a *race condition*.

When you have multiple threads trying to use the same variable at the same time, the app may do unexpected things and crash. In our app, the main thread will try to use `search.state` to display the activity spinner in the table view — and that can happen at the same time as `URLSession`'s completion handler, which runs in a background thread. We have to make sure these two threads don't get in each other's way!

Here's how the new logic works:

There is a lot that can go wrong between performing the network request and parsing the JSON. By setting `newState` to `.notSearchedYet` (which doubles as the error state) and `success` to `false` at the start of the completion handler, you assume the worst — always a good idea when doing network programming — unless there is evidence otherwise.

That evidence comes when the app is able to successfully parse the JSON and create an array of `SearchResult` objects. If the array is empty, `newState` becomes `.noResults`.

The interesting part is when the array is *not* empty. After sorting it like before, you do `newState = .results(searchResults)`. This gives `newState` the value `.results` and also associates the array of `SearchResult` objects with it. You no longer need a separate instance variable to keep track of the array; the array object is intrinsically attached to the value of `newState`.

Finally, you copy the value of `newState` into `self.state`. As mentioned before, this needs to happen on the main thread to prevent race conditions.

Updating other classes to use the state enum

That completes the changes in `Search.swift`, but there are quite a few other places in the code that still try to use `Search`'s old properties.

- In `SearchViewController.swift`, replace `tableView(_:numberOfRowsInSection:)` with:

```
func tableView(_ tableView: UITableView,
              numberOfRowsInSection section: Int) -> Int {
    switch search.state {
    case .notSearchedYet:
        return 0
    case .loading:
        return 1
    case .noResults:
        return 1
    case .results(let list):
        return list.count
    }
}
```

This is pretty straightforward. Instead of trying to make sense out of the separate `isLoading`, `hasSearched`, and `searchResults` variables, this simply looks at the value from `search.state`. The `switch` statement is ideal for situations like this.

The `.results` case requires a bit more explanation. Because `.results` has an array of `SearchResult` objects associated with it, you can *bind* this array to a temporary variable, `list`, and then use that variable inside the case to read how many items are in the array. That's how you make use of the associated value.

This pattern, using a `switch` statement to look at `state`, is going to become very common in your code.

- Replace `tableView(_:cellForRowAt:)` with:

```
func tableView(_ tableView: UITableView,  
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    switch search.state {  
    case .notSearchedYet:  
        fatalError("Should never get here")  
  
    case .loading:  
        let cell = tableView.dequeueReusableCell(  
            withIdentifier: TableView.CellIdentifiers.loadingCell,  
            for: indexPath)  
  
        let spinner = cell.viewWithTag(100) as!  
            UIActivityIndicatorView  
        spinner.startAnimating()  
        return cell  
  
    case .noResults:  
        return tableView.dequeueReusableCell(  
            withIdentifier:  
            TableView.CellIdentifiers.nothingFoundCell,  
            for: indexPath)  
  
    case .results(let list):  
        let cell = tableView.dequeueReusableCell(  
            withIdentifier:  
            TableView.CellIdentifiers.searchResultCell,  
            for: indexPath) as! SearchResultCell  
  
        let searchResult = list[indexPath.row]  
        cell.configure(for: searchResult)  
        return cell  
    }  
}
```

The same thing happens here. The various `if` statements have been replaced by a `switch` and `case` statements for the four possibilities.

Note that `numberOfRowsInSection` returns 0 for `.notSearchedYet` and no cells will ever be asked for. But because a `switch` must always be exhaustive, you also have to include a case for `.notSearchedYet` in `cellForRowAt`. Since it would be a bug if the code ever got there, you can use the built-in `fatalError()` function to help catch such a situation.

- Next up is `tableView(_:willSelectRowAt:)`:

```
func tableView(_ tableView: UITableView,  
    willSelectRowAt indexPath: IndexPath) -> IndexPath? {  
    switch search.state {
```

```
    case .notSearchedYet, .loading, .noResults:
        return nil
    case .results:
        return indexPath
    }
}
```

It's only possible to tap on rows when the state is `.results`. So for all the other cases, this method returns `nil`. And for the `.results` case, you don't need to bind the `results` array because you're not using it for anything here.

► And finally, change `prepare(for:sender:)` to:

```
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "ShowDetail" {
        if case .results(let list) = search.state {
            let detailViewController = segue.destination
                as! DetailViewController
            let indexPath = sender as! IndexPath
            let searchResult = list[indexPath.row]
            detailViewController.searchResult = searchResult
        }
    }
}
```

Here you only care about the `.results` case, so writing an entire switch statement is a bit much. For situations like this, you can use the special `if case` statement to look at a single case.

There is one more change to make in `LandscapeViewController.swift`.

► Change the `if firstTime` block in `viewWillLayoutSubviews()` to:

```
if firstTime {
    firstTime = false

    switch search.state {
        case .notSearchedYet:
            break
        case .loading:
            break
        case .noResults:
            break
        case .results(let list):
            tileButtons(list)
    }
}
```

This uses the same pattern as before. If the state is `.results`, it binds the array of `SearchResult` objects to the temporary constant `list` and passes it along to `tileButtons()`. The reason you don't use a `if case` condition here is because you'll be adding additional code to the other cases soon. But, because these cases are currently empty, they must contain a `break` statement.

- Build and run to see if the app still works — it should!

Enums with associated values are one of the most exciting features of Swift. Here you used them to simplify the way the Search state is expressed. No doubt you'll find many other great uses for them in your own apps!

- This is a good time to commit your changes.

Spin me right round

If you rotate to landscape while the search is still taking place, the app really ought to show an animated spinner to let the user know that an action is taking place. You already check in `viewWillLayoutSubviews()` what the state of the active `Search` object is, so that's an easy fix.

Show an activity indicator in landscape mode

- In `LandscapeViewController.swift`, add a new method to display an activity indicator:

```
private func showSpinner() {
    let spinner = UIActivityIndicatorView(style: .large)
    spinner.center = CGPoint(x: scrollView.bounds.midX + 0.5,
                             y: scrollView.bounds.midY + 0.5)
    spinner.tag = 1000
    view.addSubview(spinner)
    spinner.startAnimating()
}
```

This creates a new `UIActivityIndicatorView` object — a big white one —, puts it in the center of the screen, and starts animating it.

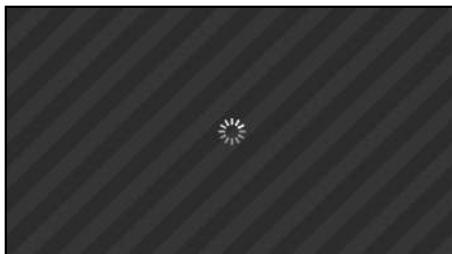
You give the spinner the tag 1000, so you can easily remove it from the screen once the search is done.

- In `viewWillLayoutSubviews()` change the `.loading` case in the `switch` statement to call this new method:



```
case .loading:  
    showSpinner()
```

- Run the app. After starting a search, quickly rotate the phone to landscape. You should now see a spinner:



A spinner indicates a search is still taking place

Note: In the new method you add `0.5` to the spinner's center position. This kind of spinner is 37 points wide and high, which is not an even number. If you were to place the center of this view at the exact center of the screen at (284, 160) then it would extend 18.5 points to either end. The top-left corner of that spinner will be at coordinates (265.5, 141.5), making it look all blurry.

It's best to avoid placing objects at fractional coordinates. By adding 0.5 to both the X and Y position, the spinner is placed at (266, 142) and everything looks sharp. Pay attention to this when working with the `center` property and objects that have odd widths or heights.

Hide the landscape spinner when results are found

This is all great, but the spinner doesn't disappear when the actual search results are received. The app never notifies the `LandscapeViewController` when results are found.

There is a variety of ways you can choose to tell the `LandscapeViewController` that the search results have come in, but let's keep it simple.

- In `LandscapeViewController.swift`, add these two new methods:

```
// MARK:- Public Methods  
func searchResultsReceived() {  
    hideSpinner()
```

```
switch search.state {
    case .notSearchedYet, .loading, .noResults:
        break
    case .results(let list):
        tileButtons(list)
    }
}

private func hideSpinner() {
    view.viewWithTag(1000)?.removeFromSuperview()
}
```

The private `hideSpinner()` method looks for the view with tag 1000 — the activity spinner — and then tells that view to remove itself from the screen.

You could have kept a reference to the spinner and used that, but for a simple situation such as this you might as well use a tag.

Because no one else has any strong references to the `UIActivityIndicatorView`, this instance will be deallocated. Note that you have to use optional chaining because `viewWithTag()` can potentially return `nil`.

The `searchResultsReceived()` method should be called from somewhere, of course, and that somewhere is the `SearchViewController`.

► In `SearchViewController.swift`'s `performSearch()` method, add the following line into the closure, below `self.tableView.reloadData()`:

```
self.landcapeVC?.searchResultsReceived()
```

The sequence of events here is quite interesting. When the search begins there is no `LandscapeViewController` object yet because the only way to start a search is from portrait mode.

But by the time the closure is invoked, the device may have rotated and if that happened `self.landcapeVC` will contain a valid reference.

Upon rotation, you also gave the new `LandscapeViewController` a reference to the active `Search` object. Now you just have to tell it that search results are available so it can create the buttons and fill them up with images.

Of course, if you're still in portrait mode by the time the search completes, then `self.landcapeVC` is `nil` and the call to `searchResultsReceived()` will simply be ignored due to the optional chaining — you could have used `if let` here to unwrap the value of `self.landcapeVC`, but optional chaining has the same effect and is shorter to write.



- Try it out. That works pretty well, eh?

Exercise: Verify that network errors are also handled correctly when the app is in landscape orientation. Find a way to create, or fake, a network error and see what happens in landscape mode. Hint: if you don't want to use the Network Link Conditioner, the `sleep(5)` function will put your app to sleep for 5 seconds. Put that in the completion handler to give yourself some time to flip the device around.

Nothing found

You're not done yet. If there are no matches found, you should also tell the user about this if they're in landscape mode.

- First, add the following method to `LandscapeViewController.swift`:

```
private func showNothingFoundLabel() {
    let label = UILabel(frame: CGRect.zero)
    label.text = "Nothing Found"
    label.textColor = UIColor.white
    label.backgroundColor = UIColor.clear

    label.sizeToFit()

    var rect = label.frame
    rect.size.width = ceil(rect.size.width/2) * 2 // make even
    rect.size.height = ceil(rect.size.height/2) * 2 // make even
    label.frame = rect

    label.center = CGPoint(x: scrollView.bounds.midX,
                           y: scrollView.bounds.midY)
    view.addSubview(label)
}
```

You first create a `UILabel` object and give it text and a color. The `backgroundColor` property is set to `UIColor.clear` to make the label transparent.

The call to `sizeToFit()` tells the label to resize itself to the optimal size. You could have given the label a frame that was big enough to begin with, but this is just as easy. This also helps when you're translating the app to a different language, in which case you may not know beforehand how large the label needs to be.

The only trouble is that you want to center the label in the view and as you saw

before, that gets tricky when the width or height are odd — something you don't necessarily know in advance. So here you use a little trick to always force the dimensions of the label to be even numbers:

```
width = ceil(width/2) * 2
```

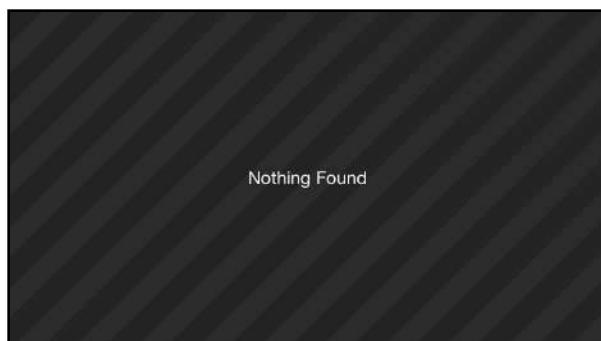
If you divide a number such as 11 by 2 you get 5.5. The `ceil()` function rounds up 5.5 to make 6, and then you multiply by 2 to get a final value of 12. This formula always gives you the next even number if the original is odd. You only need to do this because these values have type `CGFloat`. If they were integers, you wouldn't have to worry about fractional parts.

Note: Because you're not using a hardcoded number such as 480 or 568 but `scrollView.bounds` to determine the width of the screen, the code to center the label works correctly on all screen sizes.

- Inside the `switch` statement in `viewWillLayoutSubviews()`, call the new method from the case for `.noResults`:

```
case .noResults:  
    showNothingFoundLabel()
```

- Run the app and search for something ridiculous (`ewdasuq3sadf843` will do). When the search is done, flip to landscape.



Yup, nothing found here either

It doesn't work properly yet if you flip to landscape while the search is taking place. Of course, you also need to put some logic in `searchResultsReceived()`.

- Change the switch statement in that method to:

```
switch search.state {  
    case .notSearchedYet, .loading:  
        break  
    case .noResults:  
        showNothingFoundLabel()  
    case .results(let list):  
        tileButtons(list)  
}
```

Now you should have all your bases covered.

The Detail pop-up

The landscape view is that much more functional after all the refactoring and changes. But there's still one more thing left to do. The landscape search results are not buttons for nothing.

The app should show the Detail pop-up when you tap an item, like this:



The pop-up in landscape mode

This is fairly easy to achieve. When adding the buttons you can give them a *target-action* — a method to call when the Touch Up Inside event is received. Just like in Interface Builder, except now you hook up the event to the action method programmatically.

- First, still in **LandscapeViewController.swift** add the method to be called when a button is tapped:

```
@objc func buttonPressed(_ sender: UIButton) {  
    performSegue(withIdentifier: "ShowDetail", sender: sender)  
}
```

Even though this is an action method, you didn't declare it as `@IBAction`. That is only necessary when you want to connect the method to something in Interface Builder. Here you make the connection via code, so you can skip the `@IBAction` annotation.

Also note that the method has the `@objc` attribute — as you learned previously with *MyLocations*, you need to tag any method that is identified via a `#selector` with the `@objc` attribute. So, that would seem to indicate that you'll be calling this new method using a `#selector`, right?

Pressing the button simply triggers a segue, and you'll get to the segue part in a moment. But first, you should hook up the buttons to the above method.

- Add the following two lines to the button creation code in `tileButtons()`:

```
button.tag = 2000 + index
button.addTarget(self, action: #selector(buttonPressed),
                 for: .touchUpInside)
```

First you give the button a tag, so you know to which index in the `.results` array this button corresponds. That's needed in order to pass the correct `SearchResult` object to the Detail pop-up.

Also, if you replaced the `index` variable in the `for` loop earlier with a wildcard because of the Xcode compiler warning, this would be the time to revert that change.

Tip: You added 2000 to the index because tag 0 is used on all views by default, so asking for a view with tag 0 might actually return a view that you didn't expect. To avoid this kind of confusion, you simply start counting from 2000.

You also tell the button it should call the `buttonPressed()` method when it gets tapped.

- Next, add the `prepare(for:sender:)` method to handle the segue:

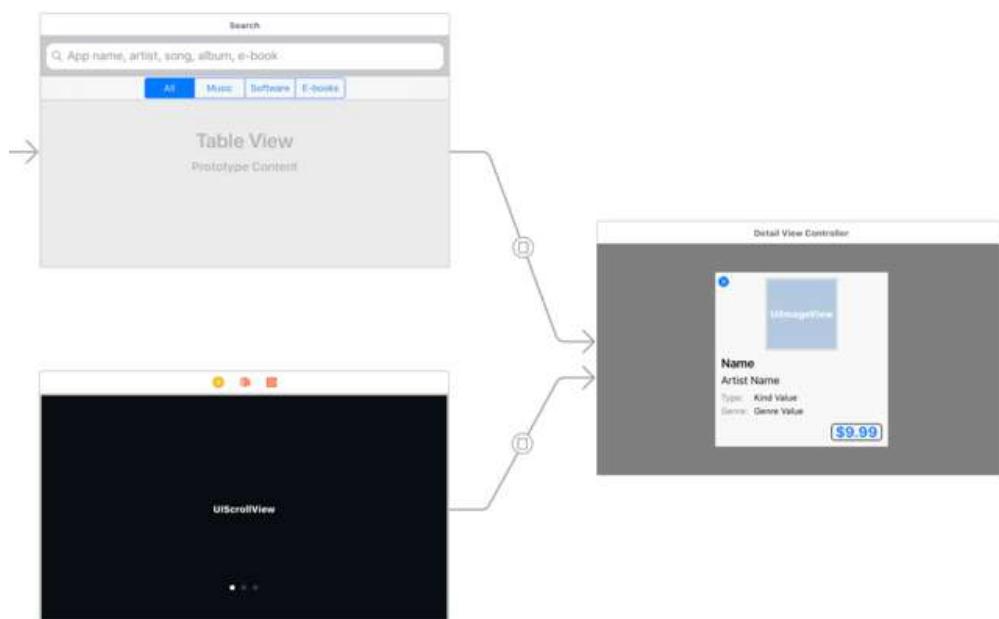
```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                     sender: Any?) {
    if segue.identifier == "ShowDetail" {
        if case .results(let list) = search.state {
            let detailViewController = segue.destination
                as! DetailViewController
            let searchResult = list[(sender as! UIButton).tag - 2000]
            detailViewController.searchResult = searchResult
        }
    }
}
```

This is almost identical to `prepare(for:sender:)` from `SearchViewController`, except now you don't get the index of the `SearchResult` object from an index-path, but from the button's tag minus 2000.

Of course, none of this will work unless you actually have a segue in the storyboard.

- Go to the Landscape scene in the storyboard and Control-drag from the yellow circle at the top to the Detail View Controller. Make it a **Present Modally** segue with the identifier set to **ShowDetail**.

The storyboard should look like this now:



The storyboard after connecting the Landscape view to the Detail pop-up

- Run the app and check it out.

Cool! But what happens when you rotate back to portrait with a Detail pop-up showing? Unfortunately, it sticks around. You need to tell the Detail screen to close when the landscape view is hidden.

- In `SearchViewController.swift`, in `hideLandscape(with:)`, add the following lines to the `animate(alongsideTransition:)` animation closure:

```
if self.presentedViewController != nil {  
    self.dismiss(animated: true, completion: nil)  
}
```

In the Console output you should see that the `DetailViewController` is properly deallocated when you rotate back to portrait.

► If you're happy with the way the code works, then let's commit it. If you also made a branch, then merge it back into the master branch.

You can find the project files for this chapter under **45 – Refactoring** in the Source Code folder.



Chapter 46: Internationalization

Eli Ganim

So far, the apps you've made in this book have all been in English. No doubt the United States is the single biggest market for apps, followed closely by Asia. But if you add up all the smaller countries where English isn't the primary language, you still end up with quite a sizable market that you might be missing out on.

Fortunately, iOS makes it very easy to add support for other languages to your apps, a process known as *internationalization*. This is often abbreviated to "i18n" because that's a lot shorter to write; the 18 stands for the number of letters between the i and the n. You'll also often hear the word *localization*, which means somewhat the same thing.

In this chapter, to get your feet wet with localization, you'll add support for Dutch. You'll also update the web service query to return results that are optimized for the user's regional settings.

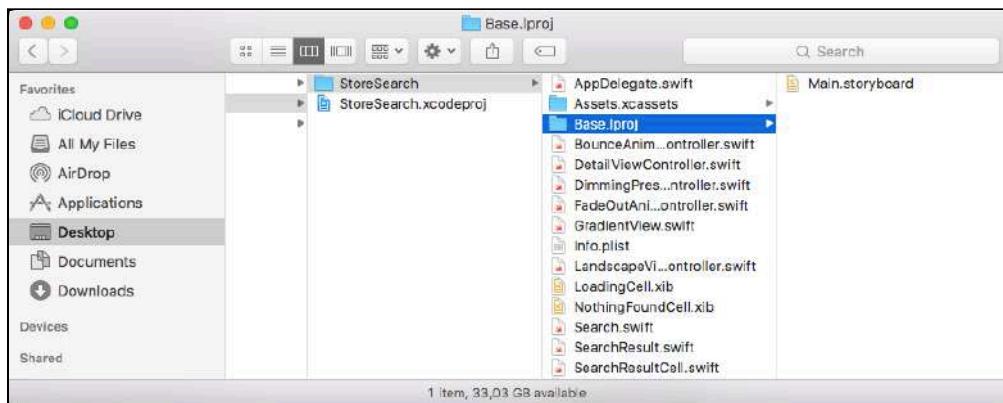
You'll cover the following items:

- **Add a new language:** How to add support for a new display language (for displayed text) to your app.
- **Localize on-screen text:** How to localize text values used in code.
- **InfoPlist.strings:** Localize Info.plist file settings such as the app name.
- **Regional Settings:** Modify the web query to send the device language and region to get localized search results.



Adding a new language

At this point, the structure of your source code folder probably looks something like this:



The files in the source code folder

There is a subfolder named **Base.lproj** that contains at least the storyboard, **Main.storyboard**. The Base.lproj folder is for files that can be localized. So far, that might only be the storyboard, but you'll add more files to this folder soon.

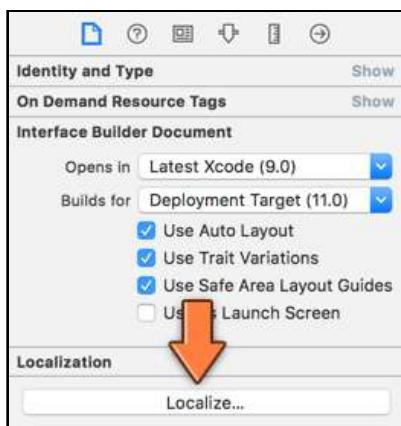
When you add support for another language, a new **XX.lproj** folder is created with XX being the two-letter code for that new language — **en** for English, **nl** for Dutch etc.

Localizing a nib file

Let's begin by localizing a simple file, the **NothingFoundCell.xib**. Often nib files contain text that needs to be translated. You can simply make a new copy of the existing nib file for a specific language and put it in the right .lproj folder. When the iPhone is using that language, it will automatically load the translated nib.

- Select **NothingFoundCell.xib** in the Project navigator. Switch to the **File inspector** pane on the right.

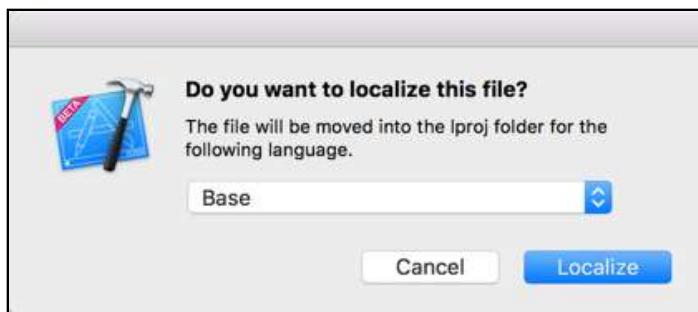
Because the NothingFoundCell.xib file isn't in any XX.lproj folders, it does not have any localizations yet.



The NothingFoundCell has no localizations

- Click the **Localize...** button in the Localization section.

Xcode asks for confirmation because this involves moving the file to a new folder:



Xcode asks whether it's OK to move the file

- Choose **English (not Base)** and click **Localize** to continue.

Look in Finder and you will see there is a new **en.lproj** — for English — folder and **NothingFoundCell.xib** has been moved to that folder:



Xcode moved NothingFoundCell.xib to the en.lproj folder

The **File inspector** for **NothingFoundCell.xib** now lists English as one of the localizations.



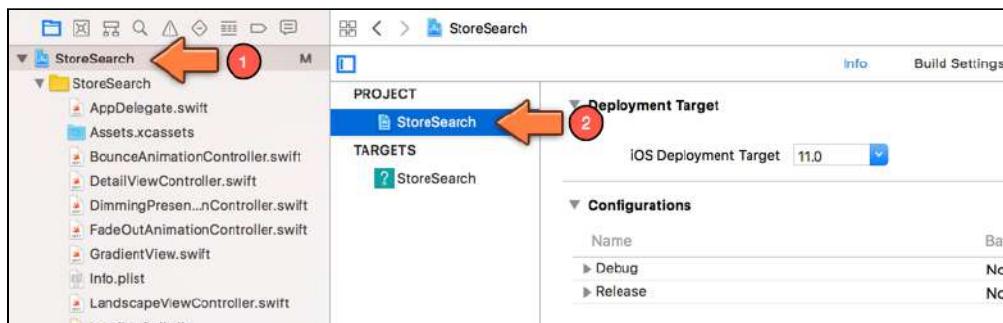
The Localization section now contains an entry for English

Adding support for a new language

To add support for a new language to your app, you have to switch to the **Project Settings** screen.

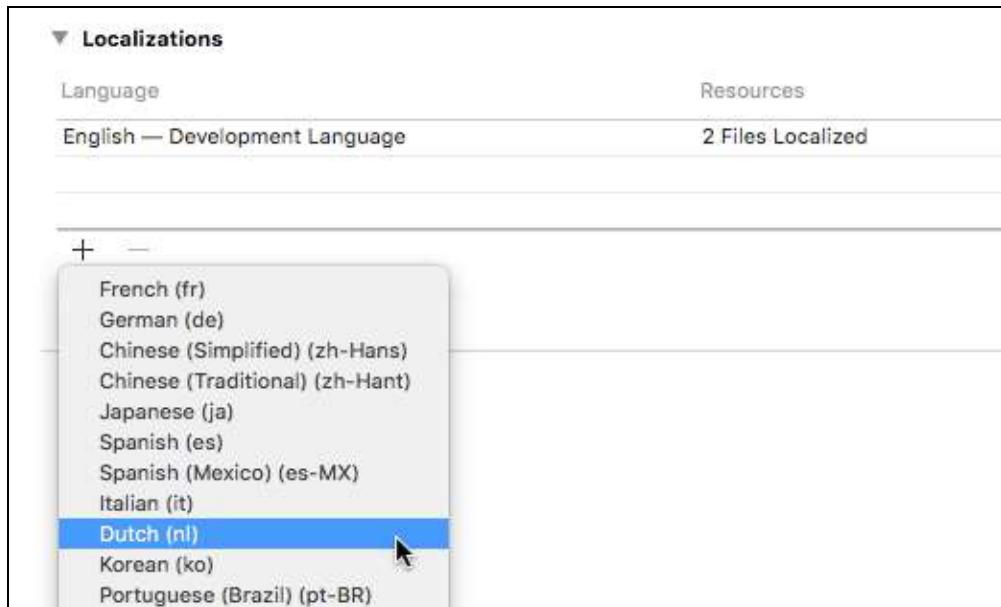
- Click on **StoreSearch** at the top of the Project navigator to open the settings page. From the central sidebar, choose **StoreSearch** under **PROJECT** (*not* under TARGETS).

If the central sidebar isn't visible, click the small blue icon at the top of the sidebar area to open it.



The Project Settings

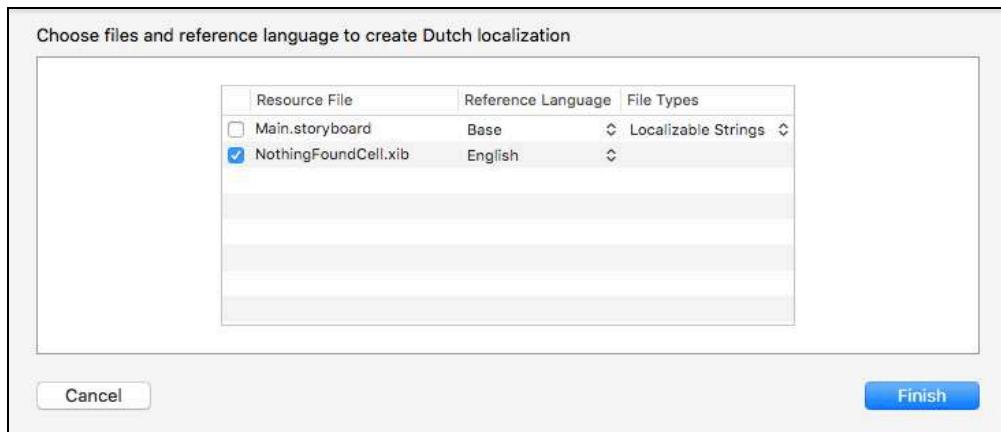
- In the Info tab, under the **Localizations** section press the + button:



Adding a new language

- From the pop-up menu choose **Dutch (nl)**.

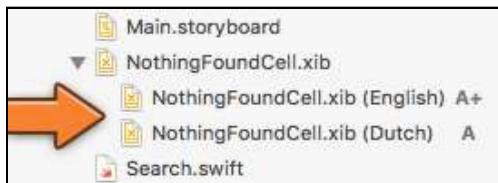
Xcode now asks which resources you want to localize. Uncheck everything except for **NothingFoundCell.xib** and click **Finish**.



Choosing the files to localize

If you look in Finder again you'll notice that a new subfolder has been added, **nl.lproj**, and that it contains another copy of NothingFoundCell.xib.

That means there are now two nib files for `NothingFoundCell`. You can also see this in the Project navigator:

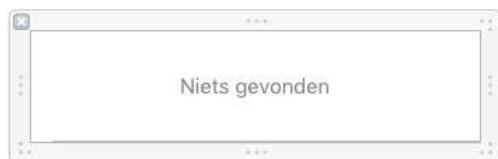


NothingFoundCell.xib has two localizations

Editing a language specific nib

Let's edit the Dutch version of this nib.

- Click on **NothingFoundCell.xib (Dutch)** to open it in Interface Builder.
- Change the label text to **Niets gevonden**.



That's how you say it in Dutch

It is perfectly all right to resize or move around items in a translated nib. You could make the whole nib look completely different if you wanted to — but that's probably a bad idea. Some languages, such as German, have very long words and in those cases you may have to tweak label sizes and fonts to get everything to fit.

If you run the app now, nothing will have changed. You have to switch the Simulator to use the Dutch language first. However, before you do that, you really should remove the app from the simulator, clean the project, and do a fresh build.

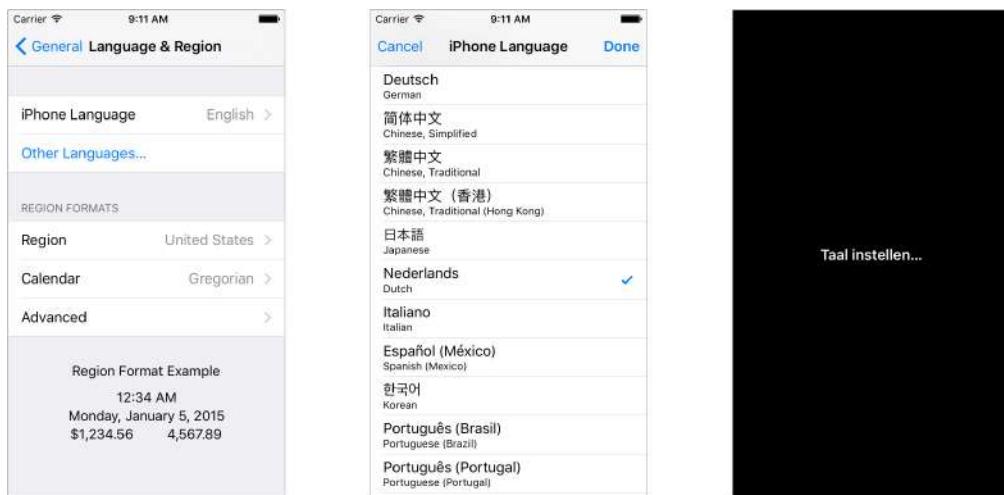
The reason for this is that the nibs were previously not localized. If you were to switch the simulator's language now, the app might still use the old, non-localized versions of the nibs, or it might not. It's better to be safe than tear your hair out wondering what went wrong, right?

Note: For this reason, it's a good idea to already put all your nib files and storyboards in the **en.lproj** folder — or in **Base.lproj**, which we'll discuss shortly — when you create them. Even if you don't intend to internationalize

your app any time soon, you don't want your users to run into the same problem later on. It's not nice to ask your users to uninstall the app — and lose their data — in order to be able to switch languages.

Switching device language

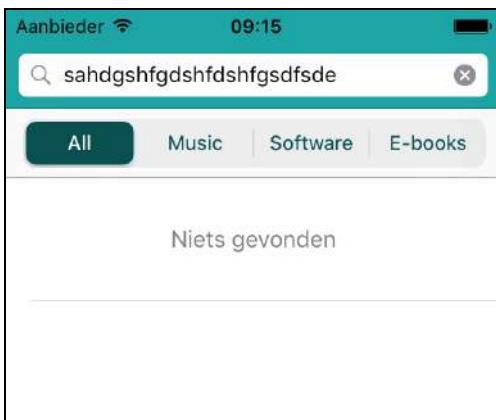
- Remove the app from the Simulator. Do a clean (**Product** ▶ **Clean** or **Shift-⌘-K**) and re-build the app.
- Open the **Settings** app in the Simulator and go to **General** ▶ **Language & Region** ▶ **iPhone Language**. From the list pick **Nederlands** (Dutch).



Switching languages in the Simulator

The Simulator will take a moment to switch between languages. This terminates the app if it was still running.

- Search for some nonsense text and the app will now respond in Dutch:



I'd be surprised if that did turn up a match

Pretty cool, just by placing some files in the **en.lproj** and **nl.lproj** folders, you have internationalized the app! You're going to keep the Simulator in Dutch for a while because the other nibs need translating too.

Note: If the app crashes for you at this point, then the following might help. Quit Xcode. Reset the Simulator and then quit it. In Finder, go to your **Library** folder, **Developer/Xcode** and throw away the entire **DerivedData** folder. Empty your trashcan. Then open the *StoreSearch* project again and give it another try. Also, don't forget to switch the Simulator back to **Nederlandse**.

Base internationalization

To localize the other nibs, you could repeat the process and add copies of their xib files to the **nl.lproj** folder. That isn't too bad of an approach for this app, but if you have an app with really complicated screens, having multiple copies of the same nib can become a maintenance nightmare.

Whenever you need to change something on that screen, you need to update all of those nibs. There's a risk that you might overlook one or more nib files and they'll be out-of-sync. That's just asking for bugs — in languages that you probably don't speak!

To prevent this from happening, you can use *base internationalization*. With this feature enabled, you don't copy the entire nib, but only the text strings. This is what the **Base.lproj** folder is for.

Let's translate the other nibs.

- Select **LoadingCell.xib** in the Project navigator. In the **File inspector** press the **Localize...** button. This time use **Base** as the language:



Choosing the Base localization as the destination

Verify with Finder that **LoadingCell.xib** got moved into the **Base.lproj** folder.

- The Localization section in the **File inspector** for **LoadingCell.xib** now contains three options: Base (with a checkmark), English, and Dutch. Put a checkmark in front of **Dutch**:

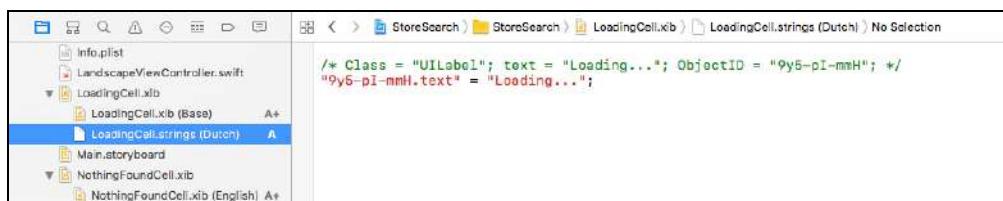


Adding a Dutch localization

In Finder you can see that **nl.proj** doesn't get a copy of the nib, but a new file does get added: **LoadingCell.strings**.

- Click the disclosure triangle in front of **LoadingCell.xib** to expand it in the Project navigator and select the **LoadingCell.strings (Dutch)** file.

You should see something like the following:



The Dutch localization is a strings file

There is still only one nib, the one from the Base localization. The Dutch translation consists of a “strings” file with just the text from the labels, buttons, and other controls.

This particular strings file contains:

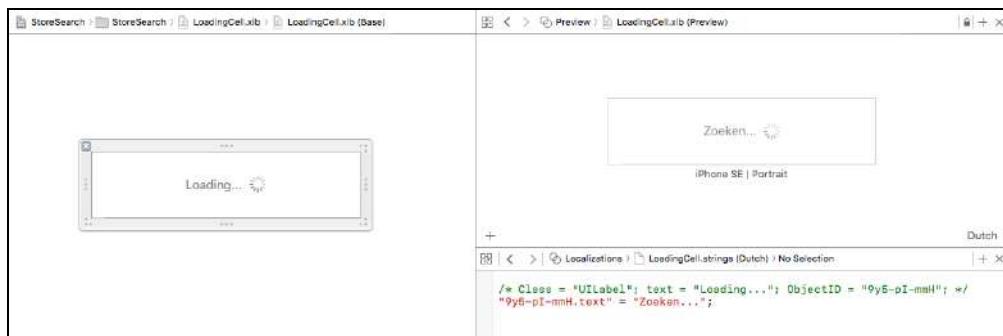
```
/* Class = "UILabel"; text = "Loading..."; ObjectID = "hU7-Dc-hSi"; */
"hU7-Dc-hSi.text" = "Loading...";
```

The green bit is a comment, just like in Swift. The second line says that the **text** property of the object with ID “hU7-Dc-hSi” contains the text **Loading...**

The ID is an internal identifier that Xcode uses to keep track of the objects in your nibs; your own nib probably has a different ID than mine. You can see this ID in the Identity inspector for the label.

► Change the text from **Loading...** to **Zoeken...**

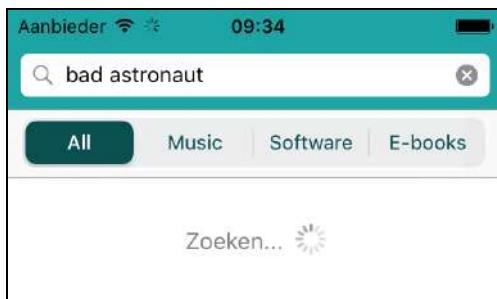
Tip: You can use the Assistant editor in Interface Builder to get a preview of your localized nib. Switch to **LoadingCell.xib (Base)** and open the Assistant editor. From the Jump bar at the top, choose **Preview**. In the bottom-right corner it says English. Click this to switch to a Dutch preview.



The Assistant editor shows a preview of the translation

If you open a second assistant pane (with the +) and set that to **Localizations**, you can edit the translations and see what they look like at the same time. Very handy!

- Do a Product ▶ Clean (to be safe) and run the app again.



The localized loading text

Note: If you don't see the "Zoeken..." text then do the same dance again: quit Xcode, throw away the DerivedData folder, reset the Simulator.

- Repeat the steps to add a Dutch localization for **Main.storyboard**. It already has a Base localization so you simply have to put a check in front of **Dutch** in the File inspector.

For the Search View Controller screen, two things need to change: the placeholder text in the Search Bar and the labels on the Segmented Control.

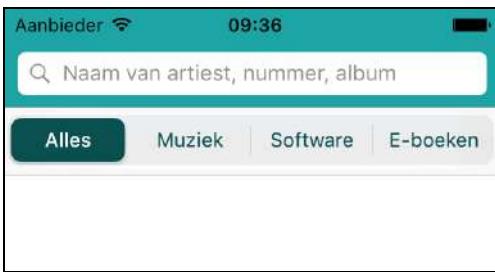
- In **Main.strings (Dutch)** change the placeholder text to **Naam van artiest, nummer, album**.

```
"68e-CH-NSs.placeholder" = "Naam van artiest, nummer, album";
```

The segment labels will become: **Alles, Muziek, Software, and E-boeken**.

```
"Sjk-fv-Pca.segmentTitles[0]" = "Alles";
"Sjk-fv-Pca.segmentTitles[1]" = "Muziek";
"Sjk-fv-Pca.segmentTitles[2]" = "Software";
"Sjk-fv-Pca.segmentTitles[3]" = "E-boeken";
```

Of course, your object IDs are going to be different - so don't rely on the IDs to find the right values in the file.



The localized SearchViewController

- For the Detail pop-up, you only need to change the **Type:** label to say **Soort:**

```
"DCQ-US-EVg.text" = "Soort:";
```

You don't need to change these:

```
"ZYp-Zw-Fg6.text" = "Genre:";  
"yz2-Gh-kzt.text" = "Kind Value";  
"Ph9-wm-1LS.text" = "Artist Name";  
"JVj-dj-Iz8.text" = "Name";  
"7sM-UJ-kWH.text" = "Genre Value";  
"x0H-GC-bHs.normalTitle" = "$9.99";
```

These labels can remain the same because you will replace them with values from the SearchResult object anyway. Also, “Genre” is the same in both languages.

Note: If you wanted to, you could even remove the text that doesn’t need localization from the strings file. If a localized version for a specific resource is missing for the user’s language, iOS will fall back to the one from the Base localization.



The pop-up in Dutch

Thanks to Auto Layout, the labels automatically resize to fit the translated text. A common issue with localization is that English words tend to be shorter than words in other languages, so you have to make sure your labels are big enough to accommodate any language. With Auto Layout that is a piece of cake.

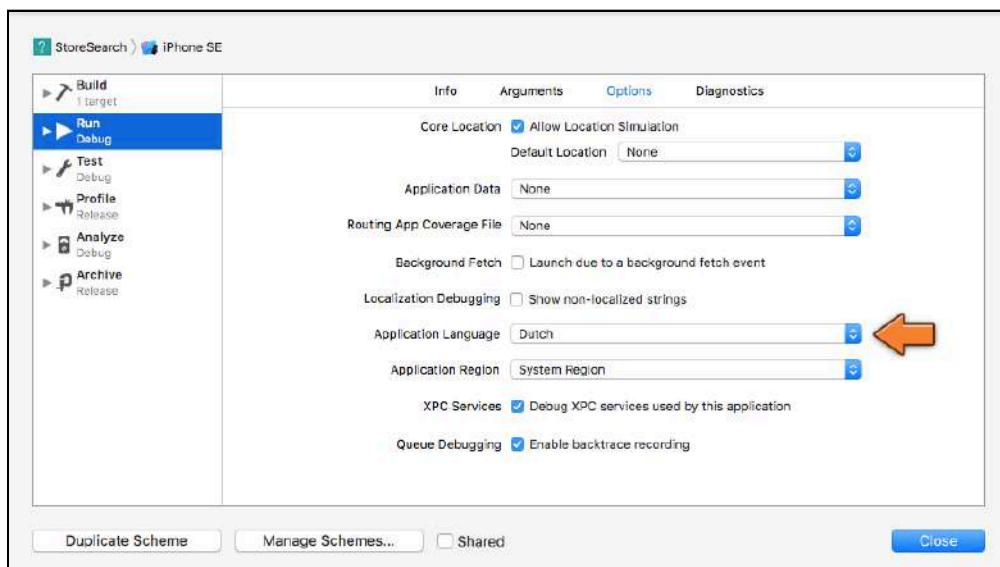
The Landscape View Controller doesn't have any text to translate.

- There is no need to give **SearchResultCell.xib** a Dutch localization — there is no on-screen text in the nib itself — but do give it a Base localization. This prepares the app for the future, should you need to localize this nib at some point.

When you're done, there shouldn't be any **xib** files outside the **.lproj** folders.

That's it for the nibs and the storyboard. Not so bad, was it? I'd say all these changes are commit-worthy.

Tip: You can also test localizations by changing the settings for the active scheme. Click on **StoreSearch** in the active scheme selector in the Xcode toolbar — next to the Simulator name — and choose **Edit Scheme**.



In the **Options** tab you can change the **Application Language** and **Region** settings. That's a bit quicker than restarting the Simulator.

Localizing on-screen text

Even though the nibs and storyboard have been translated, not all of the text is. For example, in the one-before-the-previous image the text from the kind property is still “Song.”

While in this case you could probably get away with it — everyone in the world probably knows what the word “Song” means — not all of the text from the type property will be understood by non-English speaking users.

Localizing text used in code

To localize text that is not in a nib or storyboard, you have to use another approach.

- In **SearchResult.swift**, make sure the Foundation framework is imported:

```
import Foundation
```

- Then replace the type property with:

```
var type:String {
    let kind = self.kind ?? "audiobook"
    switch kind {
        case "album":
            return NSLocalizedString("Album",
                                   comment: "Localized kind: Album")
        case "audiobook":
            return NSLocalizedString("Audio Book",
                                   comment: "Localized kind: Audio Book")
        case "book":
            return NSLocalizedString("Book",
                                   comment: "Localized kind: Book")
        case "ebook":
            return NSLocalizedString("E-Book",
                                   comment: "Localized kind: E-Book")
        case "feature-movie":
            return NSLocalizedString("Movie",
                                   comment: "Localized kind: Feature Movie")
        case "music-video":
            return NSLocalizedString("Music Video",
                                   comment: "Localized kind: Music Video")
        case "podcast":
            return NSLocalizedString("Podcast",
                                   comment: "Localized kind: Podcast")
        case "software":
            return NSLocalizedString("App",
                                   comment: "Localized kind: Software")
        case "song":
```

```
    return NSLocalizedString("Song",
                           comment: "Localized kind: Song")
  case "tv-episode":
    return NSLocalizedString("TV Episode",
                           comment: "Localized kind: TV Episode")
  default:
    return kind
}
```

Tip: Rather than typing in the above, you can use Xcode's powerful Regular Expression Replace feature to make those changes in just a few seconds.

Go to the **Find navigator** — fourth tab, the one with the magnifying glass icon, on the left sidebar — and change its mode from Find to **Replace > Regular Expression**.

In the search box type: **return "(.+)"** and press **return** to search.

In the replacement box type:

```
return NSLocalizedString("$1", comment: "Localized kind: $1")
```

This looks for any lines that match the pattern *return "something"*. Whatever that *something* is will be put in the \$1 placeholder of the replacement text.

Make sure only the relevant search results from **SearchResult.swift** are selected — you don't want to make this change to all of the search results! Click **Replace** to finish.

Thanks to Scott Gardner for the tip!

The structure of type is still the same as before, but instead of doing:

```
return "Album"
```

It now does:

```
return NSLocalizedString("Album", comment: "Localized kind:
Album")
```

Slightly more complicated, but also a lot more flexible.

`NSLocalizedString()` takes two parameters: the text to return, "Album", and a comment, "Localized kind: Album".



Here is the cool thing: if your app includes a file named **Localizable.strings** for the user's language, then `NSLocalizedString()` will look up the text ("Album") and return the translation as specified in Localizable.strings.

If no translation for that text is present, or there is no Localizable.strings file, then `NSLocalizedString()` simply returns the text as-is.

► Run the app again. The "Type:" field in the pop-up — or "Soort:" in Dutch — should still show the same text values as before because you haven't translated anything yet.

First, you need to create an empty Localizable.strings file.

► Right-click on the yellow **StoreSearch** folder in the Project navigator, select **New File...**, select the **Strings File** template under **iOS - Resources** and tap **Next**. Save the file as **Localizable.strings**.

► Select **Localizable.strings**, in the File inspector (on the right) click **Localize...**, select **English** from the dropdown, and click **Localize**.

This creates an empty English **Localizable.strings** file. You need to use a command line tool named **genstrings** to populate the file with text strings from your source files. This requires a trip to the Terminal.

Generating localizable text strings

► Open a Terminal, `cd` to the folder that contains the *StoreSearch* project. You want to go into the folder that contains the actual source files. On my system that is:

```
cd ~/Desktop/StoreSearch/StoreSearch
```

Then, type the following command:

```
genstrings *.swift -o en.lproj
```

This looks at all your source files (`*.swift`) and writes the text strings from those source files to the **Localizable.strings** file in the **en.lproj** folder.

If you open the Localizable.strings file now, this is what it should contain:

```
/* Localized Kind: Album */
"Album" = "Album";

/* Localized Kind: Software */
"App" = "App";
```

```
/* Localized kind: Audio Book */
"Audio Book" = "Audio Book";

/* Localized kind: Book */
"Book" = "Book";

/* Localized kind: E-Book */
"E-Book" = "E-Book";

/* Localized kind: Feature Movie */
"Movie" = "Movie";

/* Localized kind: Music Video */
"Music Video" = "Music Video";

/* Localized kind: Podcast */
"Podcast" = "Podcast";

/* Localized kind: Song */
"Song" = "Song";

/* Localized kind: TV Episode */
"TV Episode" = "TV Episode";
```

The things between the /* and */ symbols are the comments you specified as the second parameter of `NSLocalizedString()`. They give the translator some context about where the string is supposed to be used in the app.

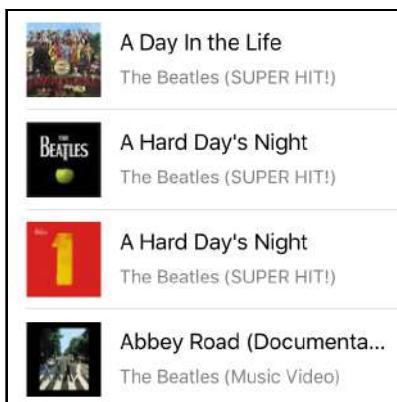
Tip: It's a good idea to make these comments as detailed as you can. In the words of fellow raywenderlich.com author Scott Gardner:

"The comment to the translator should be as detailed as necessary to not only state the words to be transcribed, but also the perspective, intention, gender frame of reference, etc. Many languages have different words based on these considerations. I translated an app into Chinese Simplified once and it took multiple passes to get it right because my original comments were not detailed enough."

► Change the “Song” line to:

```
"Song" = "SUPER HIT!";
```

► Now run the app again and search for music. For any search result that is a song, it will now say “SUPER HIT!” instead of “Song”.



Where it used to say Song it now says SUPER HIT!

Of course, changing the text in the English localization doesn't make much sense. Reverse the change to Song and then we'll do it properly.

- In the **File inspector**, add a Dutch localization for this file. This creates a copy of Localizable.strings in the **nl.lproj** folder.
- Change the translations in the Dutch version of **Localizable.strings** to:

```
"Album" = "Album";
"App" = "App";
"Audio Book" = "Audioboek";
"Book" = "Boek";
"E-Book" = "E-Boek";
"Movie" = "Film";
"Music Video" = "Videoclip";
"Podcast" = "Podcast";
"Song" = "Liedje";
"TV Episode" = "TV serie";
```

If you run the app again, the product types will all be in Dutch. Nice!

Always use `NSLocalizedString()` from the beginning

There are a bunch of other strings in the app that need translation as well. You can search for anything that begins with " but it would have been a lot easier if you had used `NSLocalizedString()` from the start. Then all you would've had to do was run the `genstrings` tool and you'd get all the strings.

Now you have to comb through the source code and add `NSLocalizedString()` to all the text strings that will be shown to the user — mea culpa!

You should really get into the habit of always using `NSLocalizedString()` for strings that you want to display to the user, even if you don't care about internationalization right away.

Adding support for other languages is a great way for your apps to become more popular, and going back through your code to add `NSLocalizedString()` is not much fun. It's better to do it right from the start!

Here are the other strings that need to be `NSLocalizedString`-ified:

```
// DetailViewController, updateUI()
artistNameLabel.text = "Unknown"
priceText = "Free"

// LandscapeViewController, showNothingFoundLabel()
label.text = "Nothing Found"

// SearchResultCell, configure(for)
artistNameLabel.text = "Unknown"

// SearchViewController, showNetworkError()
title: "Whoops...",
message: "There was an error reading from the iTunes Store.
          Please try again.",
title: "OK"
```

► Add `NSLocalizedString()` around these strings. Don't forget to use descriptive comments!

For example, when instantiating the `UIAlertController` in `showNetworkError()`, you could write:

```
let alert = UIAlertController(
    title: NSLocalizedString("Whoops...", comment: "Error alert: title"),
    message: NSLocalizedString("There was an error reading from the iTunes Store. Please try again.", comment: "Error alert: message"),
    preferredStyle: .alert)
```

Note: You don't need to use `NSLocalizedString()` with your `print()`'s. Debug output is really intended only for you, the developer, so it's best if it is in English, or whatever happens to be your native language.

- Run the **genstrings** tool again. Give it the same arguments as before. It will put a clean file with all the new strings in the **en.lproj** folder.

Unfortunately, there really isn't a good way to make genstrings merge new strings into existing translations. It will overwrite your entire file and throw away any changes that you made. There is a way to make the tool append its output to an existing file, but then you end up with a lot of duplicate strings.

Tip: Always regenerate only the file in en.lproj and then copy over the missing strings to your other Localizable.strings files. You can use a tool such as FileMerge or Kaleidoscope to compare the two files to find the new strings. There are also several third-party tools on the Mac App Store that are a bit friendlier to use than genstrings.

You might also get a warning similar to the following if you weren't consistent in using the same comment for the same word when it appears in multiple places:

```
Warning: Key "Unknown" used with multiple comments "Artist name
label: Unknown" & "Artist name: Unknown"
```

If you check the Localizable.strings file, you'll notice that you don't have two instances of the word "Unknown" (or whatever the word was that generated the error) in the file. But there are two comments for the same word.

You can easily fix this — if you wanted to — by going back and using the same comment for the same word and then running the genstrings tool again.

- Add these new translations to the Dutch **Localizable.strings**:

```
"Nothing Found" = "Niets gevonden";
"There was an error reading from the iTunes Store. Please try
again." = "Er ging iets fout bij het communiceren met de iTunes
winkel. Probeer het nog eens.";
"Unknown" = "Onbekend";
"Whoops..." = "Foutje...";
```

It may seem a little odd that such a long string as "There was an error reading from the iTunes Store. Please try again." would be used as the lookup key for a translated string, but there really isn't anything wrong with it.



By the way, the semicolons at the end of each line are not optional. If you forget a semicolon, the Localizable.strings file cannot be compiled and the build will fail.

Some people write code for NSLocalizedStrings like this:

```
let s = NSLocalizedString("ERROR_MESSAGE23",
    comment: "Error message on screen X")
```

The Localizable.strings file would then look like:

```
/* Error message on screen X */
"ERROR_MESSAGE23" = "Does not compute!";
```

This works, but is harder to read. It requires that you always have an English Localizable.strings as well.

Note also that the text "Unknown" occurred only once in Localizable.strings even though it shows up in two different places in the source code. Each piece of text only needs to be translated once.

Localizing dynamically constructed strings

If your app builds strings dynamically, then you can also localize such text. For example, in **SearchResultCell.swift**, `configure(for:)` you do:

```
artistNameLabel.text = String(format: "%@ %@",  
    searchResult.artistName, searchResult.kindForDisplay())
```

► Internationalize this as follows:

```
artistNameLabel.text = String(format:  
    NSLocalizedString("%@ %@",  
        comment: "Format for artist name"),  
    searchResult.artistName, searchResult.kindForDisplay())
```

After running **genstrings** again, this shows up in Localizable.strings as:

```
/* Format for artist name */
"%@ %@", "%1$@ (%2$@);
```

If you wanted to, you could change the order of these parameters in the translated file. For example:

```
"%@ %@", "%2$@ van %1$@";
```

It will turn the artist name label into something like this:



The “kind” now comes first, the artist name last

In this instance it's better to use a special key rather than the literal string to find the translation. It's thinkable that your app will employ the format string "%@ (%@)" in some other place and you may want to translate that completely differently there.

I'd call it something like ARTIST_NAME_LABEL_FORMAT instead (this goes in the Dutch Localizable.strings):

```
/* Format for artist name label */
"ARTIST_NAME_LABEL_FORMAT" = "%2$@ van %1$@";
```

You also need to add this key to the English version of Localizable.strings:

```
/* Format for artist name label */
"ARTIST_NAME_LABEL_FORMAT" = "%1$@ (%2$@)";
```

Don't forget to change the code as well:

```
artistNameLabel.text = String(format:
    NSLocalizedString("ARTIST_NAME_LABEL_FORMAT",
        comment: "Format for artist name label"),
    searchResult.artistName, searchResult.kindForDisplay())
```

Data-driven localization

There is one more thing I'd like to improve. Remember how in **SearchResult.swift** the type property is this enormous switch statement? That's “smelly” to me. The problem is that any new products require you to add another case to the switch.

For situations like these, it's better to use a *data-driven* approach. Here, that means you place the product types and their human-readable names in a data structure, a dictionary, rather than a code structure.

► Add the following dictionary to **SearchResult.swift**, above the class (you may want to copy-paste this from type as it's almost identical):

```
private let typeForKind = [
    "album": NSLocalizedString("Album",
```

```
        comment: "Localized kind: Album"),
"audiobook": NSLocalizedString("Audio Book",
                               comment: "Localized kind: Audio Book"),
"book": NSLocalizedString("Book",
                         comment: "Localized kind: Book"),
"ebook": NSLocalizedString("E-Book",
                           comment: "Localized kind: E-Book"),
"feature-movie": NSLocalizedString("Movie",
                                    comment: "Localized kind: Feature Movie"),
"music-video": NSLocalizedString("Music Video",
                                 comment: "Localized kind: Music Video"),
"podcast": NSLocalizedString("Podcast",
                            comment: "Localized kind: Podcast"),
"software": NSLocalizedString("App",
                             comment: "Localized kind: Software"),
"song": NSLocalizedString("Song",
                         comment: "Localized kind: Song"),
"tv-episode": NSLocalizedString("TV Episode",
                               comment: "Localized kind: TV Episode"),
]
]
```

Now the code for type becomes really short:

```
var type: String {
    let kind = self.kind ?? "audiobook"
    return typeForKind[kind] ?? kind
}
```

It's nothing more than a simple dictionary lookup.

The `??` is the nil coalescing operator. Remember that dictionary lookups always return an optional, just in case the key you're looking for — `kind` in this case — does not exist in the dictionary. That could happen if the iTunes web service added new product types. If the dictionary gives you `nil`, the `??` operator simply returns the original value of `kind`.

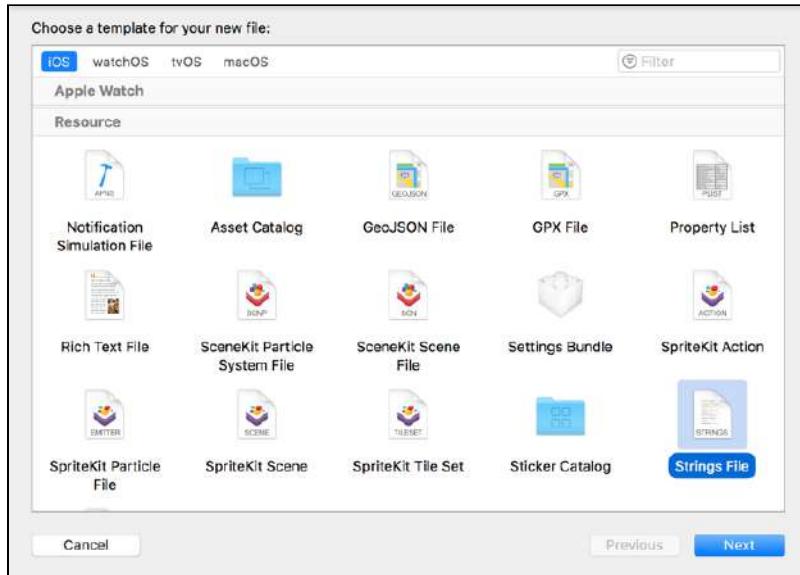
InfoPlist.strings

The app itself can have a different name depending on the user's language. The name that is displayed on the iPhone's home screen comes from the **Bundle name** setting in **Info.plist** or if present, the **Bundle display name** setting.

To localize the strings from `Info.plist`, you need a file named **InfoPlist.strings**.



- Add a new file to the project. In the template chooser scroll down to the **Resource** group and choose **Strings File**. Name it **InfoPlist.strings** (the capitalization matters!).



Adding a new Strings file to the project

- Open **InfoPlist.strings** and press the **Localize...** button from the File inspector. Choose the **English** localization.
- Also add a **Dutch** localization for this file.
- Open the Dutch version and add the following line:

```
CFBundleDisplayName = "StoreZoeker";
```

The key for the “Bundle display name” setting is `CFBundleDisplayName`.

- Run the app and close it so you can see its icon. The Simulator’s springboard should now show the translated app name:



Even the app’s name is localized!

If you switch the Simulator back to English, the app name is StoreSearch again (and of course, all the other text is back to English as well).

Regional settings

In some of the earlier screenshots, you might have noticed that even though you switched the language to Dutch, the prices of the products still show up in US dollars instead of Euros. That's for two reasons:

1. The language settings are independent of the regional settings. How currencies and numbers are displayed depends on the region settings, not the language.
2. The app does not specify anything about country or language when it sends the requests to the iTunes store, so the web service always returns prices in US dollars.

Fixing web request to include language and region

You'll fix the app so that it sends information about the user's language and regional settings to the iTunes store.

► In **Search.swift**, change the `iTunesURL(searchText:category:)` method as follows:

```
private func iTunesURL(searchText: String,
                      category: Category) -> URL {
    // Add the following 3 lines
    let locale = Locale.autoupdatingCurrent
    let language = locale.identifier
    let countryCode = locale.regionCode ?? "US"

    // Modify the URL string
    let urlString = "https://itunes.apple.com/search?" +
        "term=\(encodedText)&limit=200&entity=\(kind)" +
        "&lang=\(language)&country=\(countryCode)"

    let url = URL(string: urlString)
    print("URL: \(url!)")           // Add this
    return url!
}
```

The regional settings are also referred to as the user's *locale* and of course there is an object to represent it — `Locale`. You get a reference to the `autoupdatingCurrent` locale.

This `Locale` object is called "autoupdating" because it always reflects the current state of the user's locale settings. In other words, if the user changes their regional information while the app is running, the app will automatically use these new settings the next time it does something with the `Locale` object.

From the `Locale` object you get the language and the country code. You then put these two values into the URL using the `&lang=` and `&country=` parameters. Because `Locale.regionCode` may be `nil`, we use `?? "US"` as a failsafe.

The `print()` lets you see what exactly the URL will be.

- Run the app and do a search. Xcode should output something like the following if you have English set as the language:

```
https://itunes.apple.com/search?  
term=bird&limit=200&entity=&lang=en_US&country=US
```

It added "en_US" as the language identifier and just "US" as the country. For products that have descriptions (such as apps) the iTunes web service will return the English version of the description. The prices of all items will have USD as the currency.

Note: It's also possible you got an error message, which happens when the locale identifier returns something nonsensical such as `nl_US`. This is due to the combination of language and region settings on your Mac or the Simulator. If you also change the region (see below), the error should disappear. The iTunes web service does not support all combinations of languages and regions — so an improvement to the app would be to check the value of `language` against a list of allowed languages. I'll leave that as an exercise for you.

Testing for region changes

- In the Simulator, switch to the **Settings** app to change the regional settings. Go to **General** ▶ **Language & Region** ▶ **Region**. Select **Netherlands**.

If the Simulator is still in Dutch, then it is under **Algemeen** ▶ **Taal en Regio** ▶ **Regio**. Change it to **Nederland**. If the language is not set to Dutch, then set the language to Dutch now.



- Run StoreSearch again and repeat the search.

Xcode now says:

```
https://itunes.apple.com/search?  
term=bird&limit=200&entity=&lang=nl_NL&country=NL
```

The language and country are both now set to NL — for the Netherlands. If you tap on a search result you'll see that the price is now in Euros:



The price according to the user's region settings

Of course, you have to thank `NSNumberFormatter` for this. It now knows the region settings are from the Netherlands, so it uses a comma for the decimal point.

And because the web service now returns "EUR" as the currency code, the number formatter puts the Euro symbol in front of the amount. You can get a lot of functionality for free if you know which classes to use!

That's it as far as internationalization goes. It takes only a small bit of effort, but it definitely pays back.

You can put the Simulator back to English now.

- It's time to commit because you're going to make some big changes in the next section.

If you've also been tagging the code, you can call this v0.9, as you're rapidly approaching the 1.0 version that is ready for release.

You can find the project files for this chapter under **46 – Internationalization** in the Source Code folder.

Chapter 47: The iPad

Eli Ganim

Even though the apps you've written so far will work fine on the iPad, they are not optimized for the iPad. There isn't much difference between the iPhone and the iPad. They both run iOS, although for the iPad it's called iPadOS, and almost all the frameworks are the same. But the iPad has a much bigger screen — 768×1024 points for the regular iPad, 834x1112 points for the 10.5-inch iPad Pro, 1024×1366 points for the 12.9-inch iPad Pro — and that makes all the difference.

Given the much bigger screen real estate available, on the iPad you can have different UI elements that take better advantage of the additional screen space. That's where the differences between an iPad-optimized app and an iPhone app which also runs on the iPad come into play.

In this chapter, you will cover the following:

- **Universal apps:** A brief explanation of universal apps and how to switch from universal mode to support a specific platform only.
- **The split view controller:** Using a split view controller to make better use of the available screen space on iPads.
- **Improve the detail pane:** Re-using the Detail screen from the iPhone version, with some adjustments, to display detail information on iPad.
- **Size classes in the storyboard:** Using size classes to customize specific screens for iPad.
- **Your own popover:** Create a menu popover to be displayed on the iPad.
- **Send e-mail from the app:** Send a support e-mail from within the app using the iOS e-mail functionality.



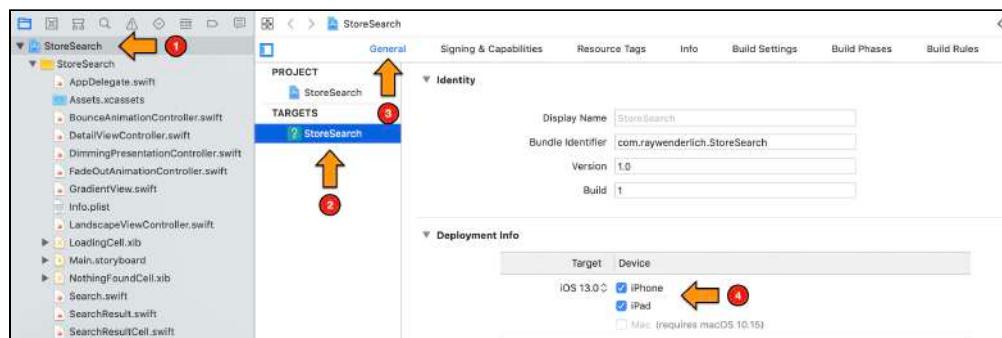
- **Landscape on iPhone Plus:** Handle landscape mode correctly for iPhone Plus devices since they act like a mini iPad in landscape mode.
- **Dark Mode support:** Support dark mode if the user chooses to activate it.

Universal apps

All new apps you create with Xcode are universal apps by default. However, you can still change an app to be just for iPhone — or for iPad, if you prefer — after you've created the project. You will *not* be doing that for *StoreSearch*. However, in case you want to know how to change your app from a universal app to one which supports a particular platform, here's how you do it:

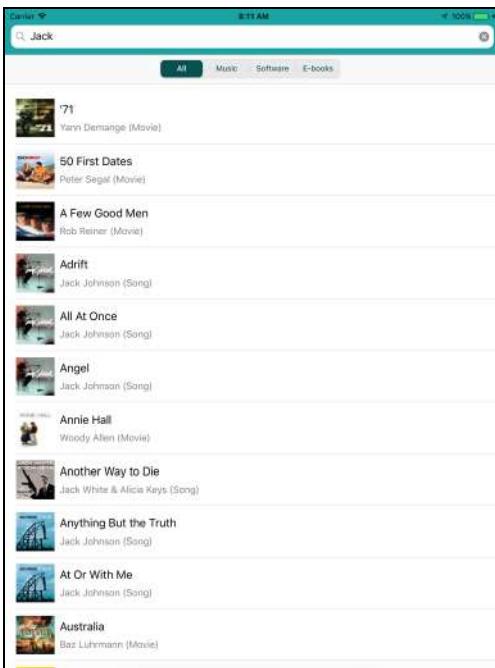
- Go to the **Project Settings** screen and select the **StoreSearch** target.

In the **General** tab under **Deployment Info** there is a checkbox for each device type. Both **iPhone** and **iPad** should be selected by default. That's where you want it to be. But, if you wanted to, you could uncheck **iPad**.



How to change device support

- While you will **not** make any changes to the setting above, if you haven't tried this before, it's a good idea to try running on an iPad simulator now. Be aware that the iPad Simulator is huge, so you may need to use the **Window > Scale** option from the Simulator menu to make it fit on your computer.



StoreSearch in the iPad Simulator

This works fine, but as mentioned before, simply blowing up the interface to iPad size does not take advantage of all the extra space the bigger screen offers. Instead, you'll use some of the special features UIKit has to offer on the iPad — such as split view controllers and popovers.

The split view controller

On the iPhone, with a few exceptions such as when you embed view controllers inside another, a view controller generally manages the whole screen.

On the iPad, because the display is so much bigger, it is common for view controllers to manage just a section of the screen. Often, you will want to combine different types of content on the same screen.

A excellent example of this is the split view controller. It has two panes: A smaller one on the left — the “master” pane — usually containing a list of items, and a larger right pane — the “detail” pane — showing more information about the thing you have selected in the master list. Each pane has its own view controller.

If you've used an iPad before, then you've seen the split view controller in action. It's used in many standard apps such as Mail and Settings.



The split view controller in landscape and portrait orientations

If the iPad is in landscape mode, the split view controller has enough room to show both panes at the same time. However, in portrait mode, only the detail view controller is visible, and the app provides a button that will slide the master pane into view. Or, you can swipe the screen to reveal/hide it.

In this section, you'll convert the app to use a split view controller. This has some consequences for the organization of the user interface.

Check the iPad orientations

Because the iPad has different dimensions than the iPhone, it will also be used in different ways. Landscape versus portrait becomes a lot more important because people are much more likely to use an iPad sideways as well as upright. Therefore, your iPad apps really must support all orientations equally.

This implies that an iPad app shouldn't make landscape show a completely different UI than portrait. So, what you did with the iPhone version of the app won't fly on the iPad — you can no longer show the `LandscapeViewController` when the user rotates the device. That feature goes out of the window.

- Open **Info.plist**. There will be a **Supported interface orientations** item with three items under it, and a **Supported interface orientations (iPad)** item with four items under it.

Supported interface orientations	Array	(3 items)
Item 0	String	Portrait (bottom home button)
Item 1	String	Landscape (left home button)
Item 2	String	Landscape (right home button)
Supported interface orientations (iPad)	Array	(4 items)
Item 0	String	Portrait (bottom home button)
Item 1	String	Portrait (top home button)
Item 2	String	Landscape (left home button)
Item 3	String	Landscape (right home button)

The supported device orientations in Info.plist

The iPad has its own supported orientations. On the iPhone, you usually don't want to enable Upside Down, but on the iPad you do. If the settings do not correspond to the above, make sure to change them to match the screenshot.

Next, run the app on the iPad simulator and verify that the app always rotates so that the search bar is on top, no matter what orientation you put the iPad in.

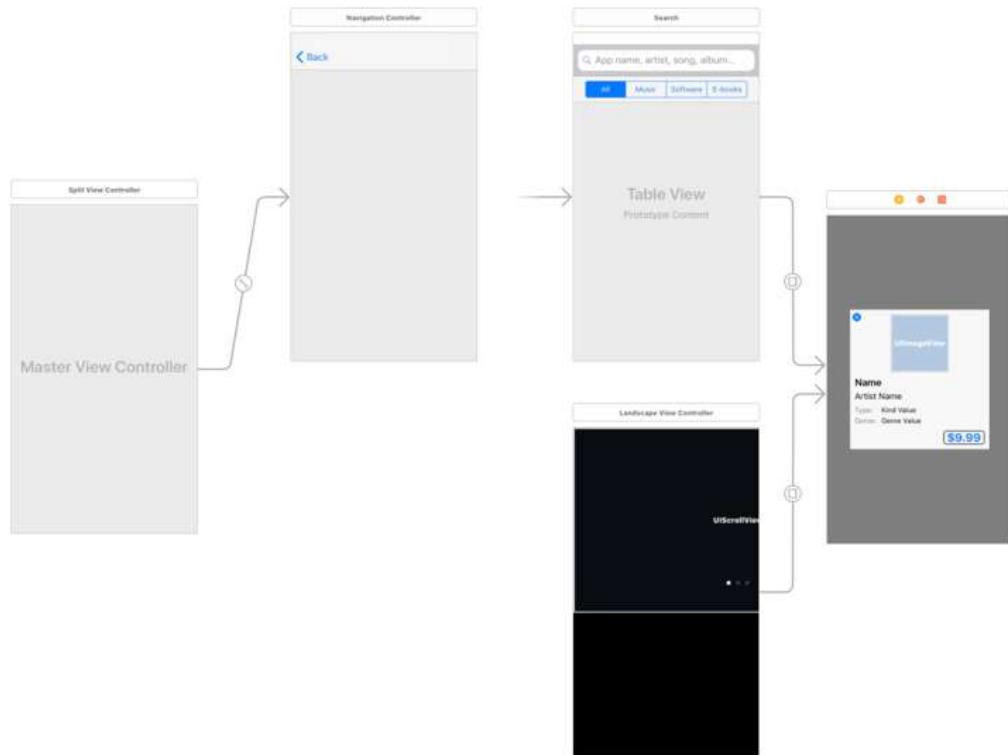
Now, let's put that split view controller into the app.

Add a split view controller

On the latest Xcode versions, you can simply add a split view controller object to the storyboard. The split view is only visible on the iPad. On the iPhone, it stays hidden. This is a lot simpler than in previous iOS versions where you had to make two different storyboard files, one for the iPhone and one for the iPad. Now you just design your entire UI in a single storyboard, and it magically works across all device types.

- Open **Main.storyboard**. If you are still in landscape mode, switch back to portrait mode now.
- Drag a new **split view controller** on to the canvas.
- The split view controller comes with several scenes pre-attached. Remove the white View Controller. Also, remove the one that says Root View Controller. Keep just the Master View Controller and the Navigation Controller.

Here's how the final result should look:



The storyboard with the new split view controller and Navigation Controller

A split view controller has a relationship segue with two child view controllers, one for the smaller master pane on the left and one for the bigger detail pane on the right.

The obvious candidate for the master pane is the `SearchViewController`, and the `DetailViewController` will go — where else? — into the detail pane.

► Control-drag from the split view controller to the Search scene. Choose **Relationship Segue – master view controller**.

This puts a new arrow between the split view and the Search screen. This arrow used to be connected to the navigation controller.

You won't put the detail view controller directly into the split view's detail pane. It's better to wrap it inside a navigation controller first. That is necessary for portrait mode where you need a button to slide the master pane into view. What better place for this button than a navigation bar?

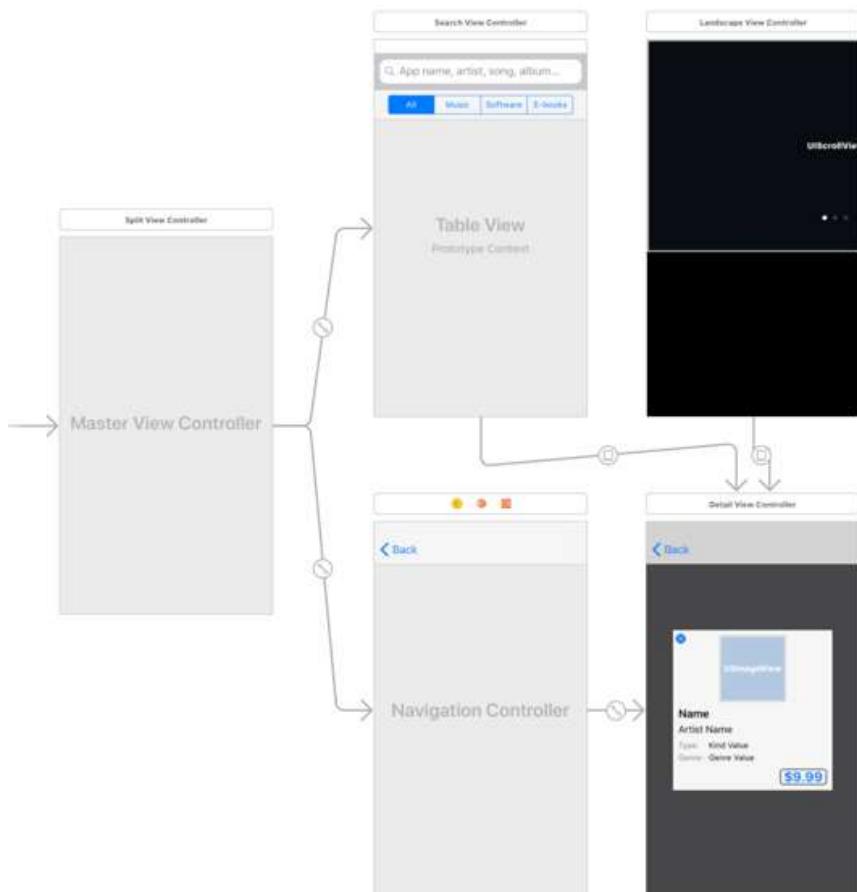
► Control-drag from the split view controller to the navigation controller. Choose **Relationship Segue – detail view controller**.

► Control-drag from the navigation controller to the detail view controller. Make this a **Relationship Segue – root view controller**.

The split view must become the initial view controller, so it gets loaded by the storyboard first.

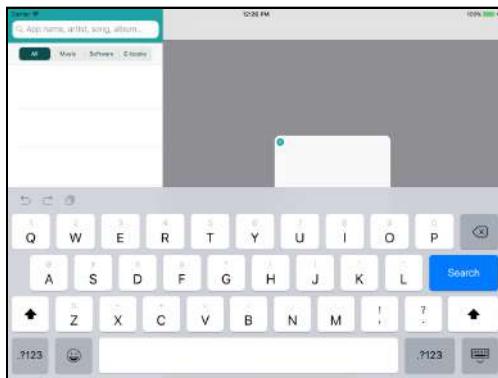
► Pick up the arrow that currently points to the Search scene — tap on the arrow to select it first and then drag — and drag it over to the split view controller. You can also check the **Is Initial View Controller** option in the Attributes inspector for the split view controller instead of dragging the arrow.

Now, everything is connected:



The master and detail panes are connected to the split view

That should be enough to get the app up and running with a split view — albeit with some issues:



The app in a split view controller

It will still take a bit of effort to make everything look good and work well, but this was the first step.

If you play with the app, you'll notice that it still uses the logic from the iPhone version, and that doesn't always work so well now that the UI sits in a split view. For example, tapping the price button from the new Detail pane crashes the app.

You'll fix the app throughout this chapter to make sure it doesn't do anything funny on the iPad!

Fixing the master pane

The master pane works fine in landscape, but in portrait mode, it's not visible. You can make it appear by swiping from the left edge of the screen — try it. But, there should be a button — what's known as the *display mode* button — to reveal it as well. The split view controller takes care of most of this logic, but you still need to put that button somewhere.

That's why you put `DetailViewController` in a navigation controller, so you can add this button — which is a `UIBarButtonItem` — to its navigation bar.

For the record, it's not mandatory to use a navigation controller for this. For example, you could also add a toolbar to the `DetailViewController` or use a different button altogether. But generally, a navigation controller is the easiest way to achieve this.

- Add the following properties to **AppDelegate.swift**, inside the class:

```
// MARK:- Properties
var splitVC: UISplitViewController {
    return window!.rootViewController as! UISplitViewController
}

var searchVC: SearchViewController {
    return splitVC.viewControllers.first as! SearchViewController
}

var detailNavController: UINavigationController {
    return splitVC.viewControllers.last as! UINavigationController
}

var detailVC: DetailViewController {
    return detailNavController.topViewController
        as! DetailViewController
}
```

These four computed properties refer to the various view controllers in the app:

- **splitVC**: The top-level view controller.
- **searchVC**: The Search screen in the master pane of the split view.
- **detailNavController**: The **UINavigationController** in the detail pane of the split view.
- **detailVC**: The Detail screen inside the **UINavigationController**.

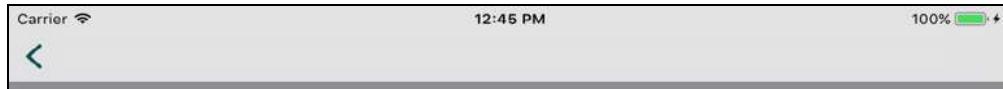
By making properties for these view controllers, you can easily refer to them without having to go digging through the view hierarchy as you did for the previous apps.

- Add the following line to **application(_:didFinishLaunchingWithOptions:)**:

```
detailVC.navigationItem.leftBarButtonItem =
    splitVC.displayModeButtonItem
```

This looks up the Detail screen and puts a button into its navigation item for switching between the split view display modes. Because the **DetailViewController** is embedded in a **UINavigationController**, this button will automatically end up in the navigation bar.

If you run the app now, all you get in portrait mode is a back arrow:



The display mode button

It would be better if this back button said “Search.” You can fix that by giving the view controller from the master pane a title.

- In **SearchViewController.swift**, add the following line to `viewDidLoad()`:

```
title = NSLocalizedString("Search", comment: "split view master button")
```

Of course, you’re using `NSLocalizedString()` because this is text that appears to the user. Hint: The Dutch translation is “Zoeken.”

- Run the app, and now you should have a proper button for bringing up the master pane in portrait mode:



The display mode button has a title

Exercise: On the iPad, rotating to landscape doesn’t bring up the special landscape view controller anymore. That’s good because we don’t want to use it in the iPad version of the app, but you haven’t changed anything in the code. Can you explain what stops the landscape view from appearing?

Answer: The clue is in `SearchViewController`’s `willTransition()`. This shows the landscape view when the new vertical size classes becomes *compact*. But on the iPad, both the horizontal and vertical size class are always *regular*, regardless of the device orientation. As a result, nothing happens upon rotation.

Improving the detail pane

The detail pane needs some more work. It just doesn't look very good yet. Also, tapping a row in the search results should fill in the split view's detail pane, not bring up a new pop-up.

You're using `DetailViewController` for both purposes — pop-up and detail pane. So, let's give it a Boolean that determines how it should behave. On the iPhone, it will be a pop-up. On the iPad, it will not.

To pop-up or not to pop-up

- Add the following instance variable to `DetailViewController.swift`:

```
var ispop-up = false
```

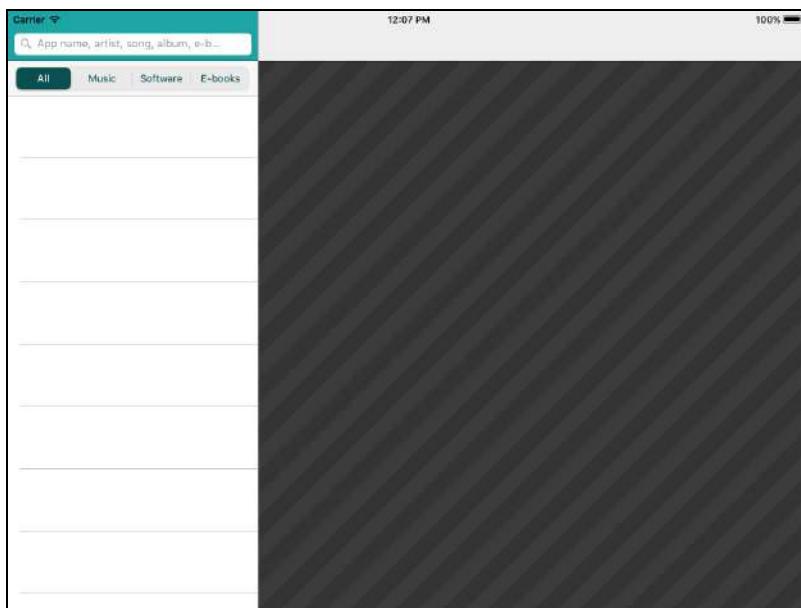
- In `viewDidLoad()` replace the four lines dealing with the gesture recognizer set up and the one setting up the background color, with the following:

```
if ispop-up {
    let gestureRecognizer = UITapGestureRecognizer(target: self,
                                                action: #selector(close))
    gestureRecognizer.cancelsTouchesInView = false
    gestureRecognizer.delegate = self
    view.addGestureRecognizer(gestureRecognizer)

    view.backgroundColor = UIColor.clear
} else {
    view.backgroundColor = UIColor(patternImage:
        UIImage(named: "LandscapeBackground")!)
}
pop-upView.isHidden = true
```

With the gesture recognizer code inside the `if ispop-up` check, tapping the background has no effect on the iPad. Likewise for the line that sets the background color to `clearColor`.

The `else` branch always hides the pop-up view until a `SearchResult` is selected in the table view. The background gets a pattern image to make things look a little nicer — it's the same image you used with the landscape view on the iPhone.



Making the detail pane look better

Initially, this means the `DetailViewController` doesn't show anything except for the patterned background. So, you need `SearchViewController` to tell the `DetailViewController` that a new `SearchResult` has been selected.

Previously, on an iPhone, `SearchViewController` created a new instance of `DetailViewController` every time you tapped a row, but now, on an iPad, it will need to use the existing instance from the split view's detail pane instead. But how does the `SearchViewController` know what that instance is?

You will have to give it a reference to the `DetailViewController`. A good place for that is in `AppDelegate` where you create those instances.

► First, add this new property to `SearchViewController.swift`:

```
weak var splitViewDetail: DetailViewController?
```

Notice that you make this property `weak`. The `SearchViewController` isn't responsible for keeping the `DetailViewController` alive since that's the job of the split view controller. It would work fine without `weak` but specifying it makes the relationship clearer.

The variable is an optional because it will be `nil` when the app runs on an iPhone.

- Add the following line to `application(_: didFinishLaunchingWithOptions:)` in `AppDelegate.swift`:

```
searchVC.splitViewDetail = detailVC
```

- To change what happens when the user taps a search result on the iPad, replace `tableView(_: didSelectRowAt:)` in `SearchViewController.swift` with:

```
func tableView(_ tableView: UITableView,  
    didSelectRowAt indexPath: IndexPath) {  
    searchBar.resignFirstResponder()  
  
    if view.window!.rootViewController!.traitCollection  
        .horizontalSizeClass == .compact {  
        tableView.deselectRow(at: indexPath, animated: true)  
        performSegue(withIdentifier: "ShowDetail",  
                     sender: indexPath)  
  
    } else {  
        if case .results(let list) = search.state {  
            splitViewDetail?.searchResult = list[indexPath.row]  
        }  
    }  
}
```

On the iPhone, this still does the same as before — pop up a new Detail screen — but on the iPad, it assigns the `SearchResult` object to the existing `DetailViewController` that lives in the detail pane.

Note: To determine whether the app is running on an iPhone, you look at the horizontal size class of the window's root view controller, which is the `UISplitViewController`. On iPhone, the horizontal size class is always *compact* — well, almost always since there are some exceptions, more about that shortly. On the iPad, it is always *regular*.

The reason you're looking at the size class from the root view controller and not `SearchViewController` is that the latter's size class is always horizontally *compact*. This is true even on iPad because it sits inside the split view's master pane.

These changes by themselves don't update the contents of the labels in the `DetailViewController`. So, let's make that happen.



The ideal place to update the labels is in a *property observer* on the `searchResult` variable. After all, the user interface needs to be updated right after you put a new `SearchResult` object into this variable.

- Change the declaration of `searchResult` in `DetailViewController.swift`:

```
var searchResult: SearchResult! {
    didSet {
        if isViewLoaded {
            updateUI()
        }
    }
}
```

You've seen this pattern a few times before. You provide a `didSet` observer to perform certain functionality when the value of a property changes. After `searchResult` has changed, you call the `updateUI()` method to set the text on the labels.

Notice that you first check whether the controller's view is already loaded. `searchResult` may be given an object when the `DetailViewController` hasn't loaded its view yet — which is exactly what happens in the iPhone version of the app. In that case, you don't want to call `updateUI()` as there is no user interface yet to update. The `isViewLoaded` check ensures this property observer only gets used when on an iPad.

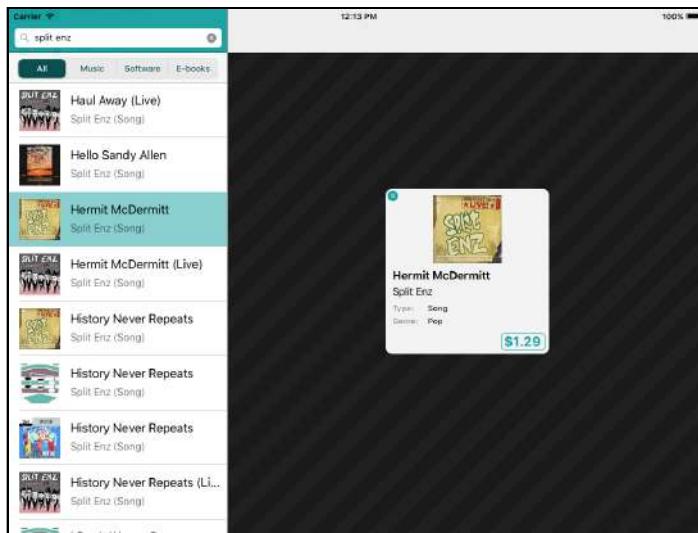
- Add the following line to the bottom of `updateUI()`:

```
popUpView.isHidden = false
```

This makes the view visible when on the iPad. Recall that in `viewDidLoad()` you hid the pop-up because there was nothing to show yet.

- Run the app. Now the detail pane should show details about the selected search result. Notice that the row in the table stays selected as well.





The detail pane shows additional info about the selected item

Fix the Detail pop-up for iPhone

One small problem: The Detail pop-up no longer works properly on the iPhone because `ispop-up` is always false — try it.

- In `prepare(for:sender:)` in **SearchViewController.swift**, add the line:

```
detailViewController.ispop-up = true
```

- Do the same thing in **LandscapeViewController.swift**. Verify that the Detail screen works properly in all situations.

Display the app name on Detail pane

It would be nice if the app showed its name in the navigation bar above the detail pane. Currently, all that space seems wasted. Ideally, this would use the localized name of the app.

You could use `NSLocalizedString()` and put the name into the `Localizable.strings` files, but considering that you already put the localized app name in **InfoPlist.strings** it would be handy if you could use that. As it happens, you can.

- In **DetailViewController.swift**, add this line to the `else` clause in `viewDidLoad()`:

```
if let displayName = Bundle.main.  
    localizedInfoDictionary?["CFBundleDisplayName"] as? String {  
    title = displayName  
}
```

The `title` property is used by the `UINavigationController` to put the title text in the navigation bar. You set it to the value of the `CFBundleDisplayName` setting from the localized version of `Info.plist`, i.e., the translations from `InfoPlist.strings`.

Because `NSBundle`'s `localizedInfoDictionary` can be `nil` you need to unwrap it. The value stored under the `"CFBundleDisplayName"` key may also be `nil`. And finally, the `as?` cast to turn the value into a `String` can also potentially fail. If you're counting along, that is three things that can go wrong in this single line of code.

That's why it's called *optional chaining*: You can check a chain of optionals in a single statement. If any of them is `nil`, the code inside the `if` is skipped. That's a lot shorter than writing three separate `if` statements!

If you were to run the app right now, no title would show up because you did not actually put a translation for `CFBundleDisplayName` in the English version of `InfoPlist.strings`.

- Add the following line to **InfoPlist.strings (English)**:

```
CFBundleDisplayName = "StoreSearch";
```

bordered width=60%

That looks good, but there are a few other small improvements to make.

Removing input focus on iPad

On the iPhone, it made sense to give the search bar the input focus, so the keyboard appeared immediately after launching the app. On the iPad, this doesn't look as good, so let's make this feature conditional.

- In `viewDidLoad()` in **SearchViewController.swift**, enclose the call to `becomeFirstResponder()` in a condition:

```
if UIDevice.current.userInterfaceIdiom != .pad {  
    searchBar.becomeFirstResponder()  
}
```



To figure out whether the app is running on an iPhone or on an iPad, you look at the current `userInterfaceIdiom`. This is either `.pad` or `.phone` — an iPod touch counts as a phone in this case.

Hiding the master pane in portrait mode

In portrait mode, after you tap a search result, the master pane stays visible and obscures about half of the detail pane. It would be better to hide the master pane when the user makes a selection.

- Add the following method to `SearchViewController.swift`:

```
private func hideMasterPane() {
    UIView.animate(withDuration: 0.25, animations: {
        self.splitViewController!.preferredDisplayMode =
            .primaryHidden
    }, completion: { _ in
        self.splitViewController!.preferredDisplayMode = .automatic
    })
}
```

Every view controller has a built-in `splitViewController` property that is non-nil if the view controller is currently inside a `UISplitViewController`.

You can tell the split view to change its display mode to `.primaryHidden` to hide the master pane. You do this in an animation block, so the master pane disappears with a smooth animation.

The trick is to restore the preferred display mode to `.automatic` after the animation completes. Otherwise, the master pane stays hidden even in landscape!

- Add the following lines to `tableView(_:didSelectRowAt:)` in the `else` clause, right after the `if case .results` block:

```
if splitViewController!.displayMode != .allVisible {
    hideMasterPane()
}
```

The `.allVisible` mode only applies in landscape, so this says, “if the split view is not in landscape, hide the master pane when a row gets tapped.”

- Try it out. Put the iPad in portrait, do a search and tap a row. Now the master pane will slide away when you tap a row in the table.



Congrats! You have successfully repurposed the Detail pop-up to also work as the detail pane of a split view controller. Whether this is possible in your own apps depends on how different you want the user interfaces of the iPhone and iPad versions to be.

If you're lucky, you may be able to use the same view controllers for both versions of the app. Often, though, you might find that the iPad user interface for your app is different enough from the iPhone's that you have to make all new ones.

The Apple Developer Forums

When I first wrote this chapter, how to hide the master pane was not explained anywhere in the official `UISplitViewController` documentation. Needless to say, I had trouble getting it to work properly.

Desperate, I turned to the Apple Developer Forums and asked my question there. Within a few hours, I received a reply from a fellow developer who ran into the same problem and who found a solution — thanks, user “timac”!

So, if you're stuck, don't forget to look at the Apple Developer Forums for a solution: <https://forums.developer.apple.com>

Size classes in the storyboard

Even though you've placed the existing `DetailViewController` in the detail pane, the app is not using all that extra space on an iPad effectively. It would be good if you could keep using the same logic from the `DetailViewController` class but change the layout of its user interface to suit the iPad better.

If you like suffering, you could do `if UIDevice.current.userInterfaceIdiom == .pad` in `viewDidLoad()` and move all the labels around programmatically. But there is a better way. This is exactly the sort of thing size classes were invented for!

► Open `Main.storyboard` and take a look at the **View as:** pane.

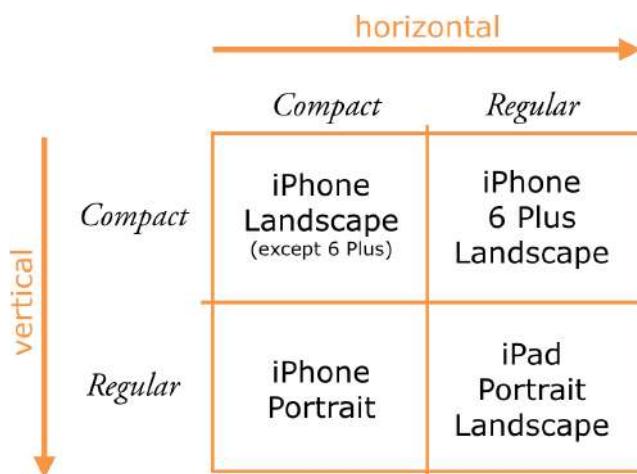


Size classes in the View as: pane

Notice how it says **iPhone 8 (wC hR)**? The **wC** and **hR** are the size class for this particular device: The size class for the width is *compact* (wC), and the size class for the height is *regular* (hR).

Recall that there are two possible size classes: *Compact* and *regular*. You can assign one of these values to the horizontal axis (width) and one to the vertical axis (height).

Here is the diagram again:



Horizontal and vertical size classes

- Use the **View as** pane to switch to **iPad Pro (9.7")**. Not only are the view controllers larger now, but you'll see the size class has changed to **wR hR**, or *regular* in both width and height.

 **View as: iPad Pro 9.7" (wR hR)**

The size classes for the iPad

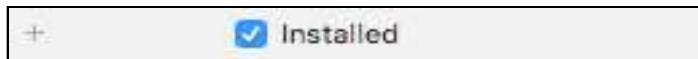
We want to make the Detail pop-up bigger when the app runs on the iPad. However, if you make any edits to the storyboard right now, these edits will also affect the design of the app in iPhone mode. Fortunately, there is a way to make edits that apply to a specific size class only.

You can tell Interface Builder that you only want to change the layout for the *regular* width size class (**wR**), but leave *compact* width alone (**wC**). Now those edits will only affect the appearance of the app on the iPad.

Uninstalling an item for a specific size class

The Detail pane doesn't need a close button on the iPad. It is not a pop-up, so there's no reason to dismiss it. Let's remove that button from the storyboard.

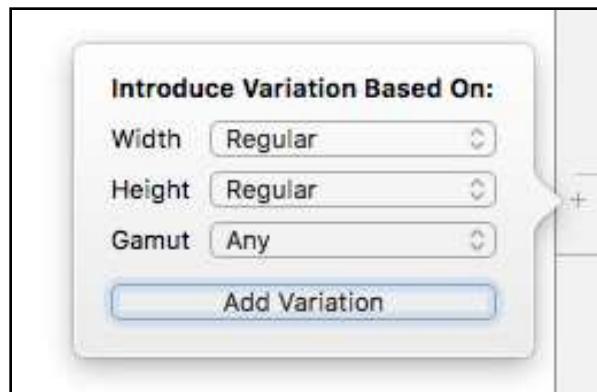
- Select the **Close Button**. Go to the Attributes inspector and scroll to the bottom, to the **Installed** option.



The installed checkbox

This option lets you remove a view from a specific size class while leaving it visible in other size classes.

- Click the tiny + button to the left of Installed. This brings up a menu. Choose **Width: Regular, Height: Regular** and click on **Add Variation**:



Adding a variation for the regular, regular size class

This adds a new line with a second Installed checkbox:



The option can be changed on a per-size class basis

- Uncheck Installed for **wR hR**. Now the Close button disappears from the scene — if the storyboard is in View as: iPad mode, of course.

The Close button still exists, but it is not installed in this size class. You can still see the button in the Document Outline, but it is grayed out:

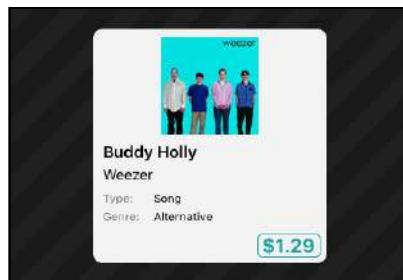


The Close Button is still present but grayed out

- Use the **View as:** panel to switch back to **iPhone 8**.

Notice how the Close button is back in its original position. You've only removed it from the storyboard design for the iPad. That's the power of size classes!

- Run the app, and you'll see that the Close button really is gone on the iPad:



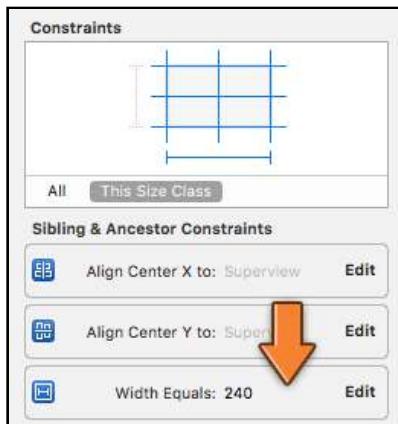
No more close button in the top-left corner

Change the storyboard layout for a given size class

Using the same principle as above, you can change the layout of the Detail screen to be completely different between the iPhone and iPad versions. For example, you can change the Detail pop-up to be bigger on an iPad.

- In the storyboard, switch to the **iPad Pro** layout again.

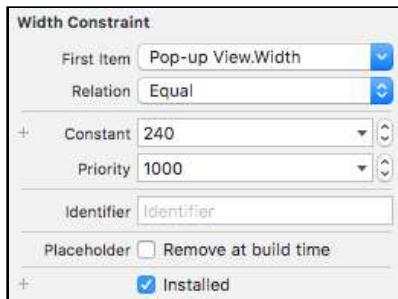
- Select the **Pop-up View** and go to the **Size inspector**. The **Constraints** section shows the constraints for this view:



The Size inspector lists the constraints for the selected view

The **Width Equals: 240** constraint has an **Edit** button. If you click that, a pop-up appears that lets you change the width. However, that will change this constraint for *all* size classes. You want to change it for the iPad only. So, do the following.

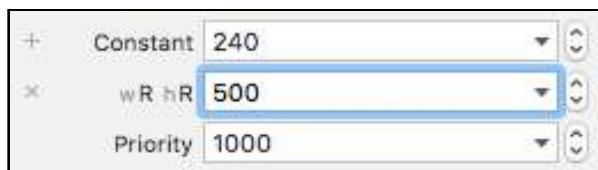
- Double-click **Width Equals: 240**. This brings up the **Size inspector** for just that constraint:



The Size inspector for the Width constraint

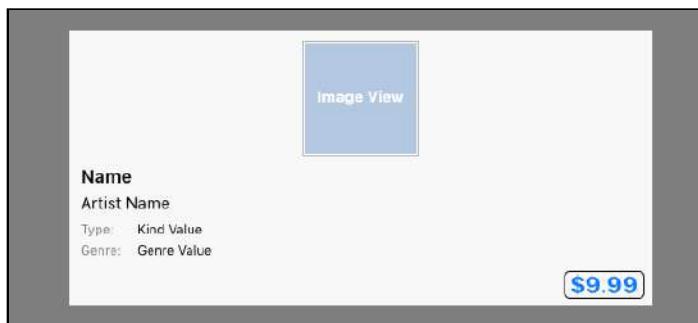
At this point, if you just type in a new value for Constant, the constraint will become larger for all size classes again.

- Click the **+** button next to Constant. In the pop-up choose **Width: Regular, Height: Regular** and click **Add Variation**. This adds a second row. Type **500** into the new **wR hR** field.



Adding a size class variation for the Constant

Now the pop-up view is a lot wider. Next up, you'll rearrange and resize the labels to take advantage of the extra space.



The Pop-up View after changing the Width constraint

- In the same way, change the **Width** and **Height** constraints of the **Image View** to **180**.
- Select the **Vertical Space** constraint between the **Name** label and the **Image View** and go to its **Size inspector**. Add a new variation for Constant and type **28** into the **wR hR** field.
- Repeat this procedure for the other **Vertical Space** constraints. Each time use the + button to add a new rule for **Width: Regular, Height: Regular** and make the new Constant 20 points taller than the existing value.

Remember, if the constraints are difficult to pinpoint, then select the view they're attached to instead and use the Size inspector to find the actual constraints.

- Make the **Vertical Space** at the top of the **Image View** 20 points.
- And finally, put the **\$9.99 button** at 20 points from the sides instead of 8.

You should end up with something that looks like this:

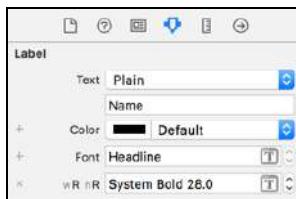


The pop-up view after changing the vertical spaces

To double-check, switch back to iPhone 8 and make sure that the Detail pane is restored to its original dimensions. If not, then you may have changed one of the original constraints instead of making a variation for the iPad's size class.

In the iPad's version of the Detail pane, the text is now tiny compared to the pop-up background. So, let's change the fonts. That works in the same fashion. You add a customization for this size class with the + button, then change the property. You can customize any attribute that has a small + in front of it for different size classes.

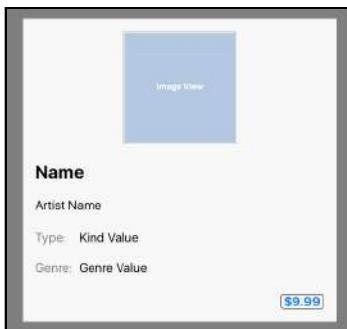
- Select the **Name** label. In the **Attributes inspector** click the + in front of **Font** to add a new variant. Choose the **System Bold** font, size **28**.



Adding a size class variation for the label's font

- Change the font of the other labels to **System**, size **20**. You can do this in one go by making a multiple-selection.
- Change all the “leading” **Horizontal Space** constraints to **20** for this size class.

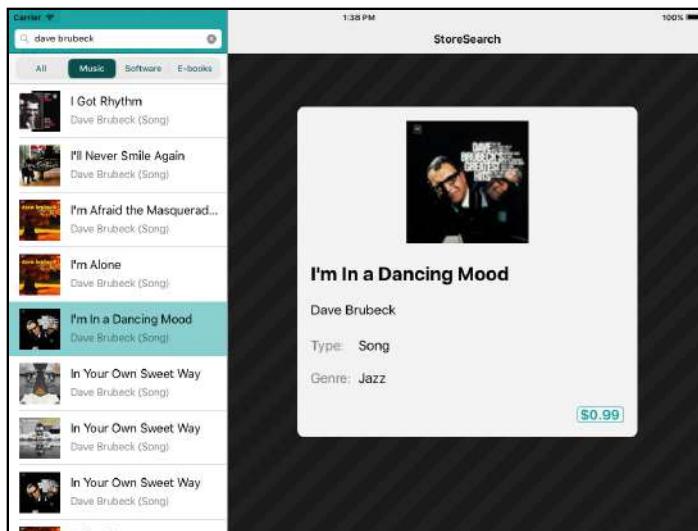
The final layout should look like this:



The layout for the Pop-up View on iPad

Switch back to iPhone 8 to make sure all the constraints are still correct there.

- Run the app, and you should have a much bigger detail view:



The iPad now uses different constraints for the detail pane

Exercise: The first time the detail pane shows its contents they appear quite abruptly because you simply set the `isHidden` property of `pop-upView` to `false`, which causes it to appear instantaneously. See if you can make it show up using a cool animation.

- This is probably a good time to try the app on the iPhone again. The changes you've made should be compatible with the iPhone version, but it's smart to make sure.

If you’re satisfied everything works as it should, then commit the changes.

Slide over and split-screen on iPad

iOS has a very handy split-screen feature that lets you run two apps side-by-side. It works on pretty much all the 64-bit iPads (with a few caveats). Because you used size classes to build the app’s user interface, split-screen support should work flawlessly.

Try it out: Run the app on the iPad Air 2 or iPad Pro simulator. Swipe up from the bottom of the screen to have your dock appear on screen. Drag an app icon from the dock on to the right (or left) edge of the iPad screen, and it should snap on, giving you two apps running side-by-side. You can drag the divider bar to adjust the size occupied by each app. Thanks to size classes, the layout of *StoreSearch* will automatically adapt to the allotted space.

The **View as:** panel has a button **Vary for Traits**. You can use this to change how a view controller acts when it is part of such a split-screen.

Your own popover

Anyone who has ever used an iPad before is no doubt familiar with popovers, the floating panels that appear when you tap a button in a navigation bar or toolbar. They are a very handy UI element.

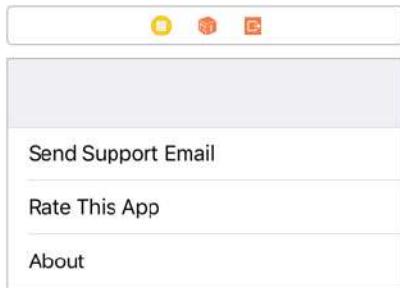
A popover is nothing more than a view controller that is presented in a special way. In this section, you’ll create a popover for a simple menu.

Adding the menu items

- In the storyboard, first, switch back to **iPhone 8** because in iPad mode the view controllers are huge and take up too much space.
- Drag a new **Table View Controller** on to the canvas and place it next to the Detail screen.
- Change the table view to **Grouped** style and give it **Static Cells**.



- Add these rows (change the cell style to **Basic**):



The design for the new table view controller

This just puts three items in the table. You will only do something with the first one in this book. Feel free to implement the functionality of the other two by yourself.

Displaying as popover

To show the new view controller inside a popover, you first have to add a button to the navigation bar to trigger the popover.

- From the Objects Library drag a new **Bar Button Item** into the **Detail View Controller's Navigation Item** — you can find it in the Document Outline. Make sure the Bar Button Item is in the **Right Bar Button Items** group.

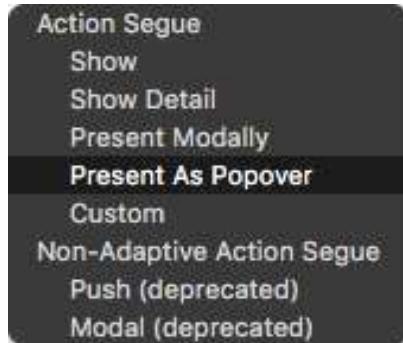


The new bar button item in the Navigation Item

- Change the bar button's **System Item** to **Action**.

This button won't show up on the iPhone because there the Detail pop-up doesn't sit in a navigation controller.

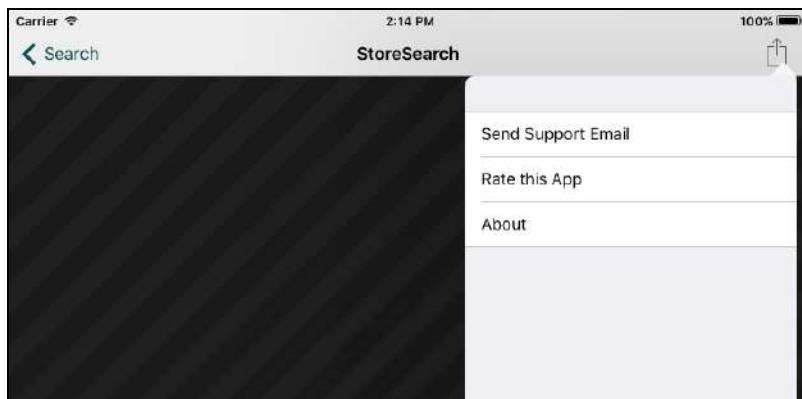
- Control-drag from the bar button (in the Document Outline) to the Table View Controller to make segue. Choose the segue type of **Action Segue – Present As Popover**.



The new bar button item in the Navigation Item

- Give the segue the identifier **ShowMenu**.

If you run the app and press the menu button, the app should look like this:



That menu is a bit too tall

Setting the popover size

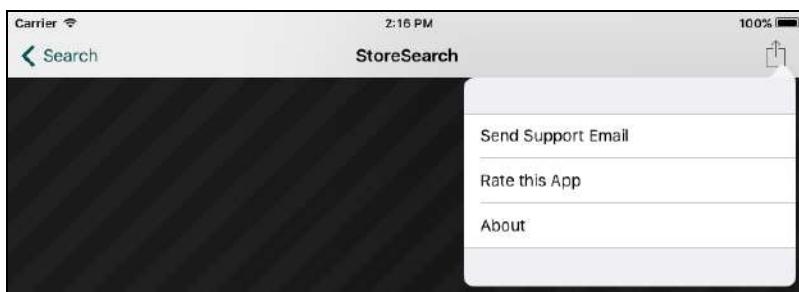
The popover doesn't really know how big its content view controller is, so it just picks a size and that's just ugly. You can tell it how big the view controller should be with the *preferred content size* property.

- In the **Attributes inspector** for the **Table View Controller**, in the **Content Size** boxes type Width: 320, Height: 204.



Changing the preferred width and height of the popover

Now the size of the menu popover looks a lot more appropriate:



The menu popover with a size that fits

When a popover is visible, all other controls on the screen become inactive. The user has to tap outside the popover to dismiss it to use the rest of the screen again. You can make exceptions to this by setting the popover's `passthroughViews` property.

Sending e-mail from the app

Now, let's make the "Send Support Email" menu option work. Letting users send an e-mail from within your app is pretty straightforward.

iOS provides the `MFMailComposeViewController` class that takes care of everything for you. It lets the user type an e-mail and then sends the e-mail using the mail account that is set up on the device.

All you have to do is create an `MFMailComposeViewController` object and present it on the screen.

The question is: Who will be responsible for this mail compose controller? It can't be the popover because that view controller will be deallocated once the popover goes away.

Instead, you will let the `DetailViewController` handle the sending of the e-mail, mainly because this is the screen that brings up the popover — via the segue from its bar button item — in the first place. `DetailViewController` is the only object that knows anything about the popover.

The `MenuViewController` class

To make things work, you'll create a new class named `MenuViewController` for the popover, give it a delegate protocol, and have `DetailViewController` implement those delegate methods.

- Add a new file to the project using the **Cocoa Touch Class** template. Name it **MenuViewController**, subclass of **UITableViewController**.
- Remove all the data source methods from this file; you don't need those for a table view with static cells. Also, remove all the commented out boilerplate code.
- In the storyboard, change the **Class** of the popover's table view controller to **MenuViewController**.
- Add a new protocol to **MenuViewController.swift**, outside the class:

```
protocol MenuViewControllerDelegate: class {
    func menuViewControllerSendEmail(_ controller: MenuViewController)
}
```

- Also, add a property for this protocol inside the class:

```
weak var delegate: MenuViewControllerDelegate?
```

Like all delegate properties, this is weak because you don't want `MenuViewController` to "own" the object that implements the delegate methods.

- Finally, add `tableView(_:didSelectRowAt:)` to handle taps on the rows from the table view:

```
// MARK:- Table View Delegates
override func tableView(_ tableView: UITableView,
                      didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)

    if indexPath.row == 0 {
        delegate?.menuViewControllerSendEmail(self)
    }
}
```

Setting the MenuViewController delegate

Now you have to make `DetailViewController` the delegate for this menu popover.

- Switch to `DetailViewController.swift` and add the following extension to the bottom of the source file to conform to the new protocol:

```
extension DetailViewController: MenuViewControllerDelegate {  
    func menuViewControllerSendEmail(_: MenuViewController) {  
    }  
}
```

Currently, the code is just a stub. You'll fill in the implementation code in a bit.

- Next, add the following navigation code to the class:

```
// MARK:- Navigation  
override func prepare(for segue: UIStoryboardSegue,  
                      sender: Any?) {  
    if segue.identifier == "ShowMenu" {  
        let controller = segue.destination as! MenuViewController  
        controller.delegate = self  
    }  
}
```

This tells the `MenuViewController` object who its delegate is.

Run the app and tap **Send Support Email**. Notice how the popover doesn't disappear yet. You'll have to manually dismiss it before you can show the mail compose sheet.

Showing the mail compose view

- The `MFMailComposeViewController` lives in the `MessageUI` framework — import that in `DetailViewController.swift`:

```
import MessageUI
```

- Then, add the following code to `menuViewControllerSendEmail()` (in the extension at the end):

```
dismiss(animated: true) {  
    if MFMailComposeViewController.canSendMail() {  
        let controller = MFMailComposeViewController()  
        controller.setSubject("Support Request",  
                           comment: "Email subject")  
        controller.setToRecipients(["your@email-address-here.com"])  
        self.present(controller, animated: true, completion: nil)  
}
```

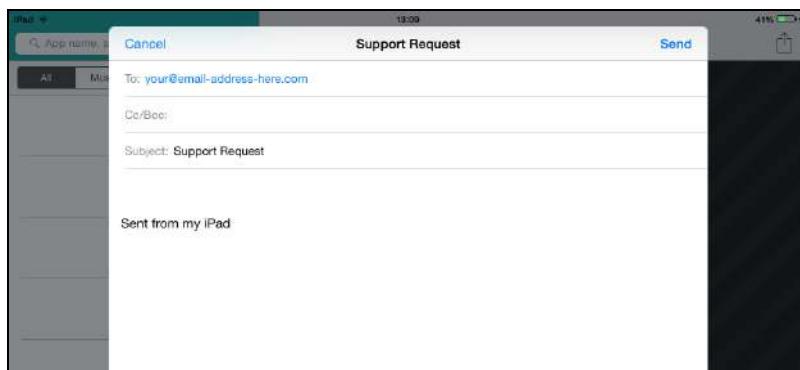
```
}
```

The code first calls `dismiss(animated:)` to hide the popover. This method takes a completion closure that until now you've always left `nil`. Here you implement the closure — using trailing syntax — to bring up the `MFMailComposeViewController` after the popover has faded away.

It's not a good idea to present a new view controller while the previous one is still in the process of being dismissed. This is why you wait to show the mail compose sheet until after the popover has done animating.

To use the `MFMailComposeViewController` object, you have to give it the subject of the e-mail and the e-mail address of the recipient. You probably should put your own e-mail address there!

► Run the app and pick the Send Support Email menu option. The standard e-mail compose sheet should slide up — if you are on a device. This won't work on the Simulator at all, sorry.



The e-mail interface

Note: If you run the app on a device and don't see the e-mail sheet, you may not have set up any e-mail accounts on your device.

The mail compose view delegate

Notice that the Send and Cancel buttons don't actually appear to do anything. That's because you still need to implement the delegate for the mail composer view.

- Add a new extension to **DetailViewController.swift**:

```
extension DetailViewController:  
    MFMailComposeViewControllerDelegate {  
        func mailComposeController(_ controller:  
            MFMailComposeViewController, didFinishWith result:  
            MFMailComposeResult, error: Error?) {  
            dismiss(animated: true, completion: nil)  
        }  
    }
```

The `result` parameter says whether the mail was successfully sent or not. This app doesn't really care about that, but you could show an alert in case of an error if you wanted. Check the documentation for the possible result codes.

- In the `menuViewControllerSendEmail()` method, add the following line, after the controller is created:

```
controller.mailComposeDelegate = self
```

- Now, if you press Cancel or Send, the mail compose sheet gets dismissed.

Modal sheet presentation styles

Did you notice that the mail sheet did not take up the entire screen area in landscape, but when you rotate to portrait it (almost) does? That is called a **page sheet**.

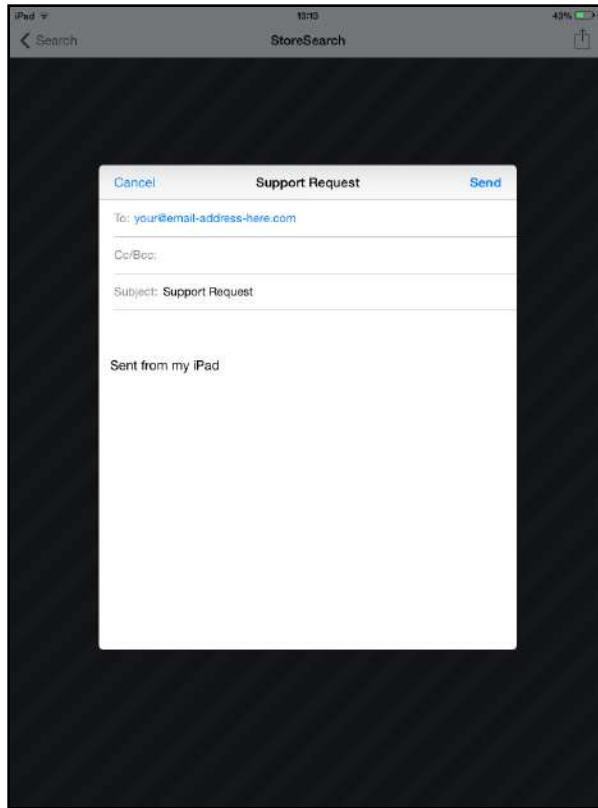
On the iPhone, if you presented a modal view controller, it always takes over the entire screen, but on the iPad, you have several options.

The page sheet is probably the nicest option for the `MFMailComposeViewController`, but let's experiment with the other ones as well, shall we?

- In `menuViewControllerSendEmail()`, add the following line:

```
controller.modalPresentationStyle = .formSheet
```

The `modalPresentationStyle` property determines how a modal view controller is presented on the iPad. You've switched it from the default page sheet to a **form sheet**, which looks like this:



The email interface in a form sheet

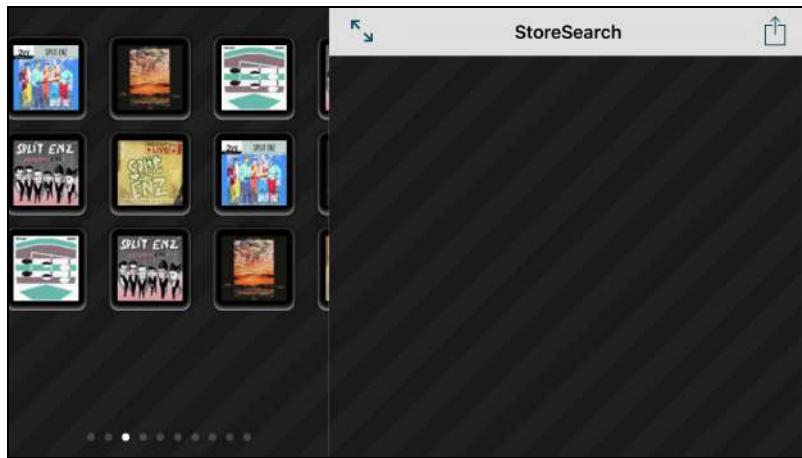
A form sheet is smaller than a page sheet, so it takes up less room on the screen. There is also a “full screen” presentation style that always covers the entire screen, even in landscape. Try it out!

Landscape on iPhone Plus

The iPhone Plus is a strange beast. It mostly works like any other iPhone, but sometimes it gets ideas and pretends to be an iPad.

- Run the app on the **iPhone 8 Plus** Simulator, do a search, and rotate to landscape.

The app will look something like this:



The landscape screen appears in the split view's master pane

The app tries to both show the split view controller and the special landscape view at the same time. That's not going to work.

The iPhone Plus devices are so big that they're almost small iPads. The designers at Apple decided that in landscape orientation, the Plus should behave like an iPad, and therefore it shows the split view controller.

What's the trick? Size classes, of course! On a landscape iPhone Plus, the horizontal size class is *regular*, not *compact*. But the vertical size class is still compact, just like on the smaller iPhone models.

Showing split view correctly for iPhone Plus

To stop the `LandscapeViewController` from showing up, you have to make the rotation logic smarter.

► In `SearchViewController.swift`, change `willTransition(to:with:)` to:

```
override func willTransition(to newCollection:  
    UITraitCollection, with coordinator:  
    UIViewControllerTransitionCoordinator) {  
    super.willTransition(to: newCollection, with: coordinator)  
  
    let rect = UIScreen.main.bounds  
    if (rect.width == 736 && rect.height == 414) || // portrait  
        (rect.width == 414 && rect.height == 736) { // landscape  
        if presentedViewController != nil {  
            dismiss(animated: true, completion: nil)
```

```
        }
    } else if UIDevice.current.userInterfaceIdiom != .pad {
    switch newCollection.verticalSizeClass {
        case .compact:
            showLandscape(with: coordinator)
        case .regular, .unspecified:
            hideLandscape(with: coordinator)
    }
}
```

The bottom bit of this method is as before; it checks the vertical size class and decides whether to show or hide the `LandscapeViewController`.

You don't want to do this for the iPhone Plus, so you need to detect somehow that the app is running on the Plus. There are a couple of ways you can do this:

- Look at the width and height of the screen. The dimensions of the iPhone Plus are 736 by 414 points.
- Look at the hardware device name. There are APIs for finding this out, but you have to be careful. Often one type of iPhone can have multiple model names, depending on the cellular chipset used or other factors.

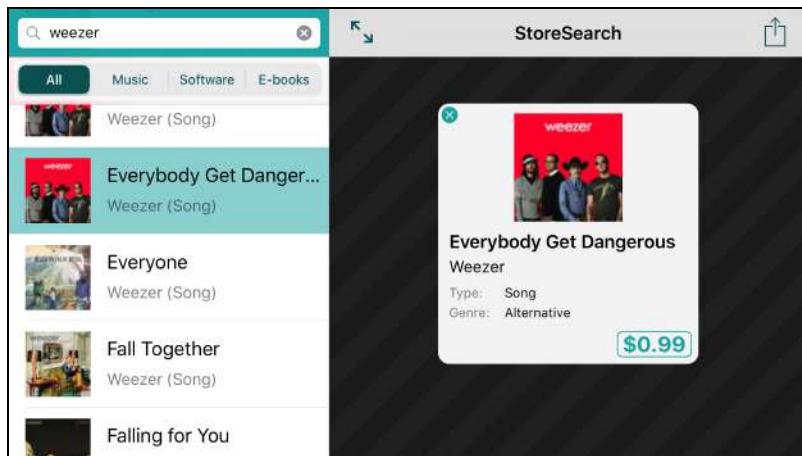
What about the size class? That sounds like it would be the obvious thing to tell the different devices apart. Unfortunately, looking at the size class *doesn't* work.

If the device is in portrait, the Plus has the same size classes as the other iPhone models. In other words, in portrait, you can't tell from the size class alone whether the app is running on a Plus or not. Only in landscape, and even then, if you have Display Zoom on, the Plus will no longer have a different size class. It will act like a regular iPhone.

The approach you're using in this app is to look at the screen dimensions. You need to check for both orientations because the screen bounds change depending on the orientation of the device.

Once you've detected the app runs on an iPhone Plus, you no longer show the landscape view, and you dismiss any Detail pop-up that may still be visible before you rotate to landscape.

► Try it out. Now the iPhone Plus shows a proper split view:



The app on the iPhone 8 Plus with a split-view

Adding size class based UI changes for iPhone Plus

Of course, the Detail pane now uses the iPhone-size design, not the iPad design.

That's because the size class for `DetailViewController` is now *regular* width, *compact* height. You didn't make a specific design for that size class, so the app uses the default design.

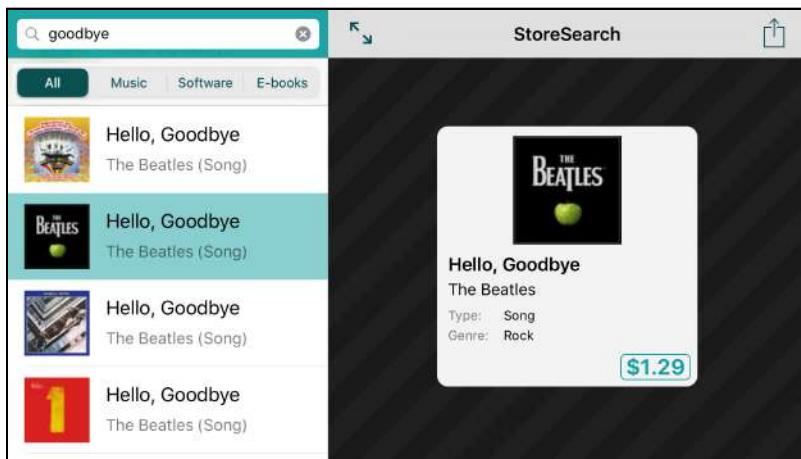
That's fine for the size of the Detail view, but it does mean the close button is visible again.

- Open the storyboard. Then, open the **View as:** panel and switch to the **iPhone 8 Plus** mode. Next, switch to landscape mode. This will help you get the size classes right when you add exceptions.
- Select the **Close Button** in the Detail scene. In the **Attributes inspector**, add a new row for **Installed** (for Width: Regular, Height: Compact) and uncheck it:



Adding a variation for size class width regular, height compact

- Select the **Center Y Alignment** constraint on **Pop-up View**. Change its **Constant** to **20**, but only for this size class. This moves the Detail panel down a bit.



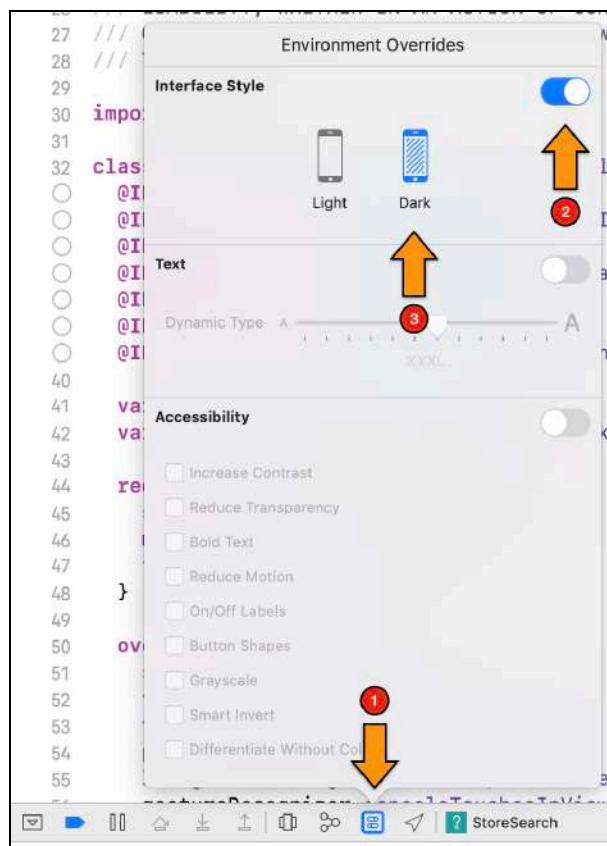
The finished StoreSearch app on the iPhone 6s Plus or 7 Plus

Dark mode support

iPhone has system-wide support for dark mode, which lets the user define that all supporting apps should show a dark color palette, rather than a light one. All of Apple's native apps support this mode already. You should support it in your app as it's easier on the eyes at night, and the users of your app would expect your app to also respect this setting. And it looks a lot better!

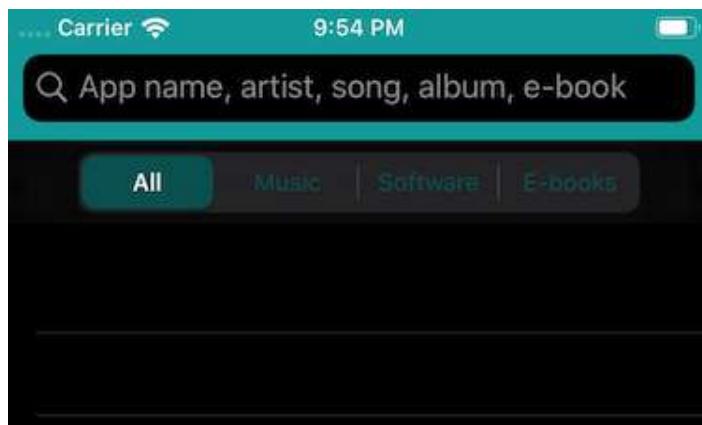
Follow these steps to see how your app looks like in dark mode:

- Run the app. In Xcode's debug bar, click the **Environment Overrides** button (or choose **Debug > View Debugging > Configure Environment Overrides...**).
- Click on the **Interface Style** toggle and choose **Dark**.



Enable dark mode in your app

Now if you switch to the simulator, you'll see the app looks like this:

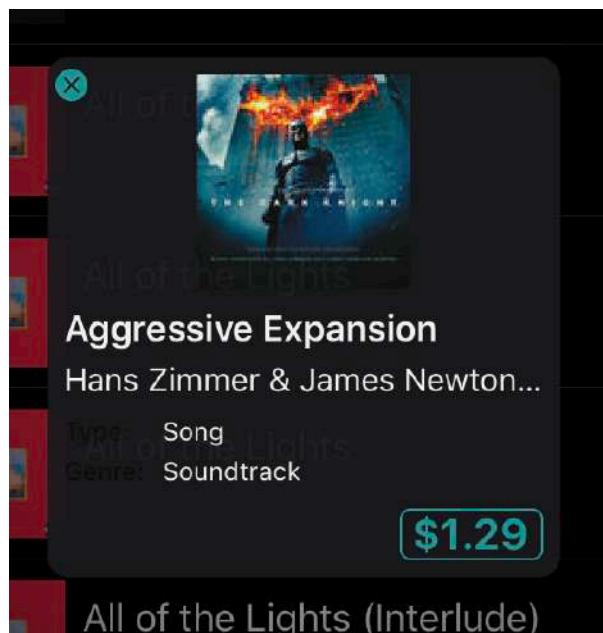


The app in dark mode

It already looks great, and you didn't have to write any extra lines of code! How is that possible?

The native controls you've been using to design the app, like the search bar, segment control, table view, etc. already support dark mode and know which colors to change. Also, as long as you're using system colors, like *System Background Color* or *Placeholder Text Color*, they will also automatically switch to their dark mode equivalent when switching modes. The only changes you'll need to make are in custom colors you defined manually.

For example, if you search for something and click on a result, you'll see that the details pop-up didn't adopt to dark mode perfectly:



The pop-up view in dark mode

The labels **Type** and **Genre** are in black on a dark gray background, so it's hard to make out the words. If you recall, when you created this view, you specifically set the color of these labels to black. The right thing to do is to choose one of the provided system colors. When you add a new control, the default color values are usually already set to one of the system colors. When you added these two labels, you were asked to choose non-default colors on purpose, so you would see that it breaks dark mode support.

It's time to fix it:

- Open the storyboard and go to the Detail scene.
- Click on the **Type:** label and change the color from black to **Default (Label Color)**. Do the same for the **Genre:** label.
- Run the app again and see how the content adopts its colors correctly when you switch dark mode on and off.

Dark mode in storyboard

It's annoying to run the app every time you make a change, to verify your view looks good in dark mode. Luckily Apple has incorporated dark mode in Interface Builder as well.

- Once again, open the storyboard and go to the Detail scene.
- Open the **View as** panel and under **Interface Style** choose **Dark Style**:

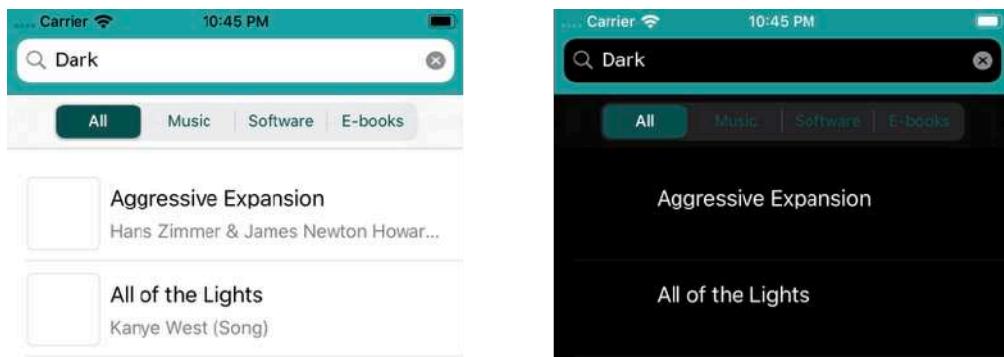


The pop-up view in dark mode

Now you know how to easily switch and make sure the UI looks perfect in both modes.

Providing dark mode assets

Most images should look fine in dark mode. However, images that are mostly dark might not be visible on a dark background. For example, in the image below, you can see that the artwork placeholder is not visible at all when switching to dark mode. The iPhone can't know how to convert your images to dark mode.



The pop-up view in dark mode

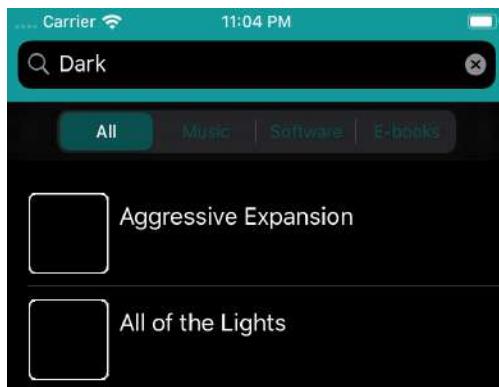
For that reason, you have the option to provide a dark mode version of an image.

- Open the assets catalog and choose the **Placeholder** image.
- In the right pane choose the **Attributes Inspector**.
- Under **Appearances** choose **Any, Dark**.

Additional placeholders will show up and you'll be able to put images that will only be displayed when the app is in dark mode.

- Drag the files PlaceholderDark@2x.png and PlaceholderDark@3x.png from the **ImagesDark** folder to the **2x** and **3x Dark Appearance** placeholders respectively.

Now when you switch from light to dark mode, the placeholder changes as well.



The pop-up view in dark mode

Supporting dark mode in code

The app looks great in dark mode by now, but there's still something that could be improved. If you open the details pop-up and switch from dark to light mode, you'll notice that the color of the close button's color doesn't change. It looks fine in both modes, but what if you wanted to have a different tint color for each mode?

For that, you'll need to detect whether dark mode is enabled before setting the tint color.

► Open **DetailViewController.swift**

- In the method `ViewDidLoad()` find the line `view.tintColor = . . .`
- Replace it with the following:

```
if (traitCollection.userInterfaceStyle == .light) {  
    view.tintColor = UIColor(red: 20/255, green: 160/255, blue:  
    160/255, alpha: 1)  
} else {  
    view.tintColor = UIColor(red: 140/255, green: 140/255, blue:  
    240/255, alpha: 1)  
}
```

Now the tint color of the pop-up dialog will update as you switch modes.

And that's it for the *StoreSearch* app! Congratulations for making it this far, it has been a long road.

► Celebrate by committing the final version of the source code and tagging it v1.0!

You can find the project files for this chapter under **47 – The iPad** in the Source Code folder.

Chapter 48: Distributing the App

Eli Ganim

What do you do with an app that is finished? Upload it to the App Store, of course! And with a little luck, make some big bucks...

Throughout this book, you've probably been testing the apps on the Simulator and occasionally on your device. That's great, but when the app is nearly done, you may want to let other people beta test it.

In this chapter, you'll learn how to beta test the *StoreSearch* app. After that, I'll also show you how to submit the app to the App Store, which is basically an extension of the same process.

By the way, I'd appreciate it if you don't actually submit the apps from this book. Let's not spam the App Store with dozens of identical *StoreSearch* or *Bull's Eye* apps.

This chapter will cover the following:

- **Join the Apple Developer program:** How to sign up for the paid Apple Developer Program.
- **Beta testing:** How to beta test your app using Apple's TestFlight service.
- **Submit to the App Store:** How to submit your app to Apple for review before being made available on the App Store.



Join the Apple Developer program

Once you're ready to make your creations available on the App Store, it's time to join the paid Apple Developer Program.

To sign up, go to developer.apple.com/programs/ and click the blue **Enroll** button.

On the sign-up page you'll need to enter your Apple ID. Your developer program membership will be tied to this account. It's OK to use the same Apple ID that you're already using with iTunes and your iPhone, but if you run a business you might want to create a new Apple ID to keep things separate.

You can enroll as an Individual or as an Organization. There is also an [Enterprise Program](#), but that's for big companies who want to distribute apps within their own organization only. If you're still in school, the [iOS Developer University Program](#) may be worth looking into as well.

You buy the Developer Program membership from the online Apple Store for your particular country. Once your payment is processed, you'll receive an activation code that you use to activate your account.

Signing up is usually pretty quick. In the worst case it may take a few weeks, as Apple will check your credit card details and if they find anything out of the ordinary — such as a misspelled name — your application may run into delays. So make sure to enter your credit card details correctly or you'll be in for an agonizing wait.

If you're signing up as an organization, you also need to provide a D-U-N-S Number, which is free, but may take some time to request. You cannot register as an organization if you have a single-person business such as a sole proprietorship or DBA ("doing business as"). In that case you need to sign up as an Individual.

You will have to renew your membership every year, but if you're serious about developing apps, then that \$99/year will be worth it.

Beta testing

You will be distributing your app for beta testing via Apple's TestFlight service.

TestFlight

In the early days of iOS development, the only way to send builds to testers was via what was known as Ad Hoc distribution. You had to register specific devices for Ad Hoc distribution — for which you needed to know the unique ID for the device — and there was a limit of 100 devices per developer account. You could only reset the devices in this list once per year, when you renewed your developer account.

Additionally, you had to go through a complicated manual signing process to sign builds for Ad Hoc distribution and you had to send these builds out to your users and hope that they could figure out how to install the builds on their devices and troubleshoot any installation issues by themselves, or provide you enough information to help them figure out what was going on.

All this changed with the introduction of Apple's TestFlight service.

TestFlight allows you to distribute your beta builds to 10,000 testers and all you need is just their e-mail address!

What's more, the process itself is fairly straightforward since you simply build your app in Xcode and upload to the App Store. The app binary would go through some processing at this point and once the processing is complete, you are able to offer the app for testing to your internal testers immediately.

In addition to the 10,000 external testers, you can also have internal testers from your organization. The internal tester count is limited to 25 people — each of whom can run your app on up to 30 devices — and they have to be part of your Apple Developer team. If you signed up as a single developer, this probably would not work for you since you can't add additional members to your team on the Apple Developer portal. On the other hand, if you signed up as an organization, then you can start testing immediately.

If you want to distribute to external testers — the 10,000 testers mentioned earlier — you do have to go through an initial review of your app by Apple. This process is usually quite fast and takes about a day. This generally has to be done only once per new beta build — for the very first beta build. After that, you can simply push out new builds without needing to wait for approval from Apple.

To add new users to beta test, you simply invite them using their e-mail address. They will receive an invitation e-mail which they can accept, or reject by simply ignoring the e-mail. If they accept the invitation, they are prompted to install the TestFlight app which will handle installing beta builds and notifying users of updates to beta builds from then on.

You can read more on TestFlight at: <https://developer.apple.com/testflight/>

Apple Developer portal

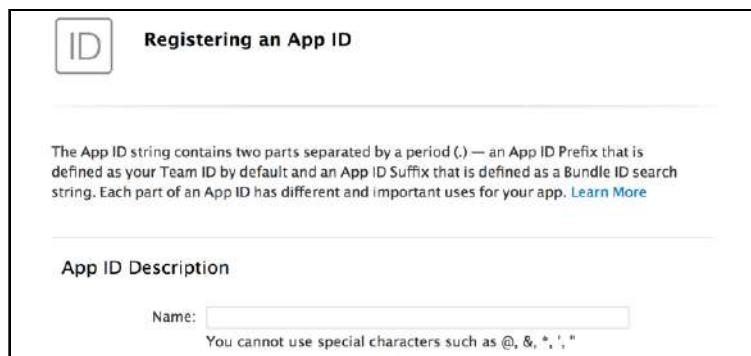
While the new TestFlight workflow for beta testing is miles ahead of what you had previously, it still requires you to do a bunch of things on several different Apple sites. You start out on the Apple Developer portal where you need to create an App ID for your new app.

- Open your favorite web browser and navigate to your account details page at developer.apple.com/account. You might need to sign in if you are not logged in. Once you are there, click on **Certificates, Identifiers & Profiles**.

Tip: If you have trouble using the site and you are not using Safari, try using Safari. Other browsers can throw up weird issues sometimes.

Note: Like any piece of software, the Apple Developer Portal changes every now and then. It's possible that by the time you read this, some of the options are in different places or have different names. The general flow should still be the same, though. And if you really get stuck, online help is usually available.

- Click on **App IDs** under **Identifiers** in the sidebar — you should get a list of existing app IDs. Press the + button on the top right to add a new App ID:



Creating a new App ID

- Fill in the **App ID Description** field. This can be anything you want — it's just for your reference on the Provisioning Portal.
- The **App ID Prefix** field contains the ID for your team. You cannot modify this value.
- Under **App ID Suffix**, select **Explicit App ID**. In the **Bundle ID** field you must enter the identifier that you used when you created the Xcode project. Here that is **com.raywenderlich.StoreSearch**.



The Bundle ID must match with the identifier from Xcode

If you want your app to support push notifications, In-App Purchases, or iCloud, then you can also configure that here. *StoreSearch* doesn't need any of that, so leave the other fields on the default settings.

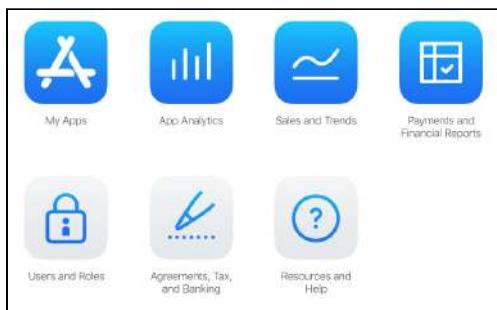
- Press **Continue** and then **Register** to create the App ID. The portal will now generate the App ID for you and add it to the list.

The full App ID is something like **U89ECKP4Y4.com.yourname.StoreSearch**. That number in front is your Apple Developer Team ID.

App Store Connect

Next, you need to add your app to App Store Connect.

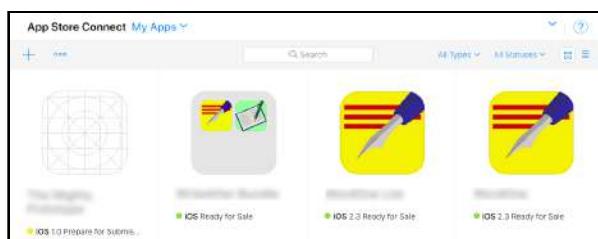
- Navigate to appstoreconnect.apple.com via your browser of choice. Again, try using Safari if you run into any issues with the App Store Connect site.
- Log in using the same Apple ID that you used to sign up for your Apple Developer account.
- The first screen you see will look something like the following — if you are not an administrator for the App Store Connect account, you might see fewer options on the screen than in the screenshot.



Initial App Store Connect screen

- If you've never been to App Store Connect before, then make sure to first visit the **Agreements, Tax, and Banking** section and fill out the forms. All that stuff has to be in order before your app can be distributed on the App Store.
- Select **My Apps** — this is the option you need to manage everything related to your apps. You create new app entries, edit existing ones, and manage your beta testing and app distribution tasks all from there.

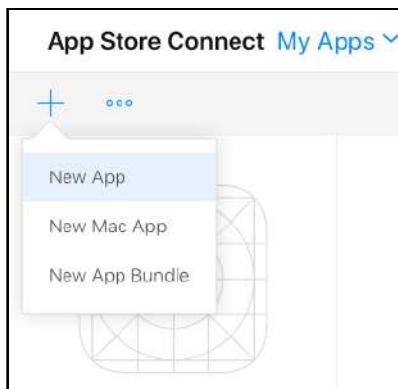
The page you are taken to lists your existing apps, if you have any. It also allows you to add new apps to App Store Connect.



The My Apps page on App Store Connect

Tip: If you are stuck and don't know what to do, you can use the help icon on the top right to access guides and videos which might allow you to figure out how to use the App Store Connect site.

- Click the plus (+) icon on the top left to add a new app and then select **New App** from the menu.



Add a new app on App Store Connect

- This should present you with a new dialog for entering the basic information necessary for an app.

The screenshot shows the "New App" configuration dialog. It includes fields for "Platforms" (with "iOS" checked and "tvOS" unchecked), "Name" (set to "StoreSearch"), "Primary Language" (set to "English (U.S.)"), and "Bundle ID" (with a tooltip explaining it must match Xcode and can't be changed). There are also fields for "SKU" and "Create" and "Cancel" buttons at the bottom.

Platforms	?
<input checked="" type="checkbox"/> iOS	<input type="checkbox"/> tvOS
Name	?
StoreSearch	
Primary Language	?
English (U.S.)	
Bundle ID	?
Choose	
The bundle ID must match the one you used in Xcode. It can't be changed after you upload your first build.	
SKU	?
Cancel Create	

New app information on App Store Connect

Select the checkbox for **iOS** (since yours is an iOS app), enter the name of your app, select the primary language from the dropdown and select the Bundle ID from the dropdown.

The Bundle ID would be the app ID you added on the Apple Developer Portal earlier. If you don't see the app ID you added, try refreshing your browser or waiting for a bit in case the information has not updated on the Apple servers. Generally, the information should be reflected almost immediately.

Tip: If you are not sure about what you are supposed to enter for a particular field, you can always click on the question mark icon next to each field to get a hint as to what you should enter. Also note the hint in the above screenshot — you **must** have a bundle ID matching the ID you have in Xcode for your project. Otherwise, the upload from Xcode will fail.

The last value you have to enter is the **SKU** (or “skew”), which stands for Stock-Keeping Unit. This one confuses people a bit since it can be any unique value *for your company*. Basically, Apple does not care what this value is — it's only used for reporting purposes. It has to be unique for your apps. So, for example, if you use 1001 as the SKU for your first app, you *can't* use 1001 as the SKU for your second app.

- Once you've filled in all the information, click **Create** and your new app will be added to App Store Connect — if there are no errors.

If your app name has been used before — by anybody, not just you — for another app, or if your Bundle ID is not unique, or if your SKU has been used before, you will get an error message at this point. You would have to fix these issues and try again if this happens. Generally, it's the app name which gives you problems — so it's always a good idea to figure out if the name you selected is in use before you try to add a new app to App Store Connect.

That's all you need to do at the App Store Connect end for the time being.

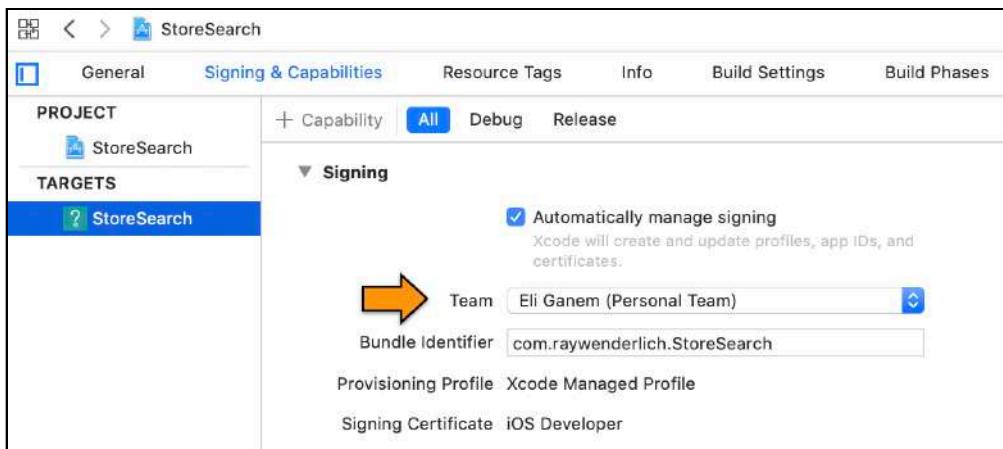
Upload for beta testing

Once you have your app on App Store Connect, you can upload the app for beta testing — and later submission to Apple — quite easily.

- Start Xcode, if you're not already running it, and then open the *StoreSearch* project.



- In the **Project Settings** screen, in the **General** tab, choose the correct **Team**. As you noticed when you created the App ID, the team ID is connected to your App ID. So make sure that you have the right team selected here, otherwise you will run into issues later when you try to upload the build ...



Choosing the team

- Change the device in the active scheme selector, on the Xcode toolbar, to **Generic iOS Device**.

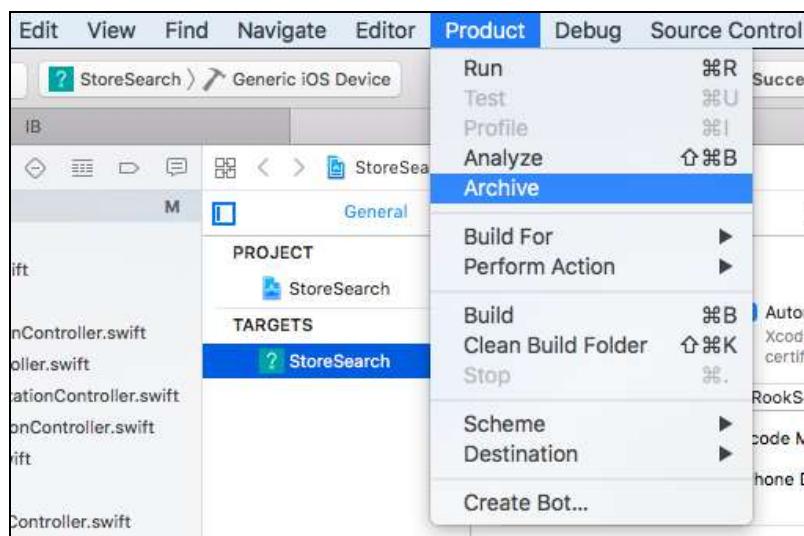


Selecting Generic iOS Device

Normally, when you build your app, you build for a specific Simulator or for a connected device because your intention at that point is to run the build on that particular Simulator or device.

But when you build for distribution, you have no idea which particular device a build would run on. So, you have to build using the generic device setting in order to ensure that the resulting app would be compiled correctly to run on all supported devices.

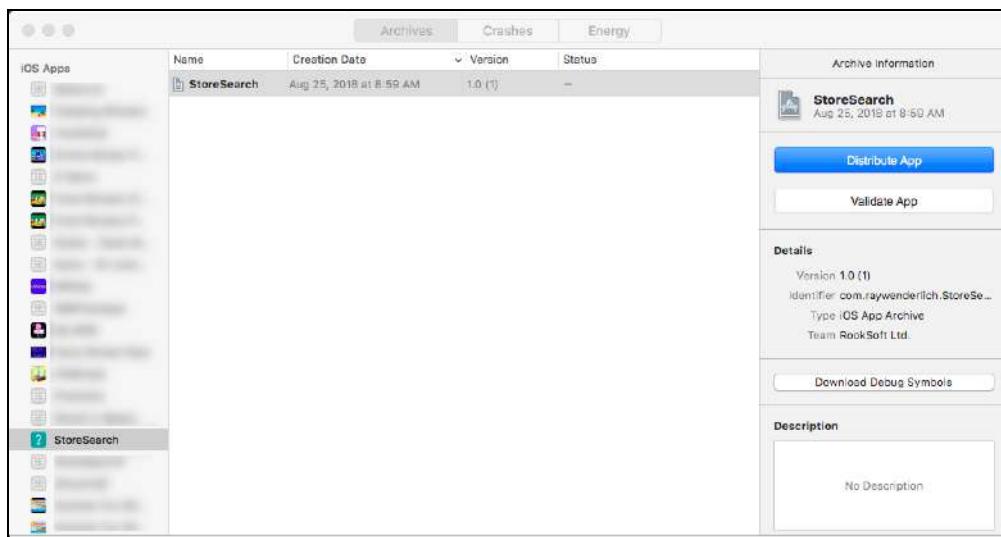
- Select **Product > Archive** from the Xcode menu.



Create app archive

If the Archive option is disabled, then you probably did not select the Generic iOS Device from the active scheme selector as per the previous step. You can only build an archive if you have the Generic iOS Device selected.

The app will compile the project and link it. If everything goes smoothly, Xcode should open the Organizer window and display the new archive which was just created.



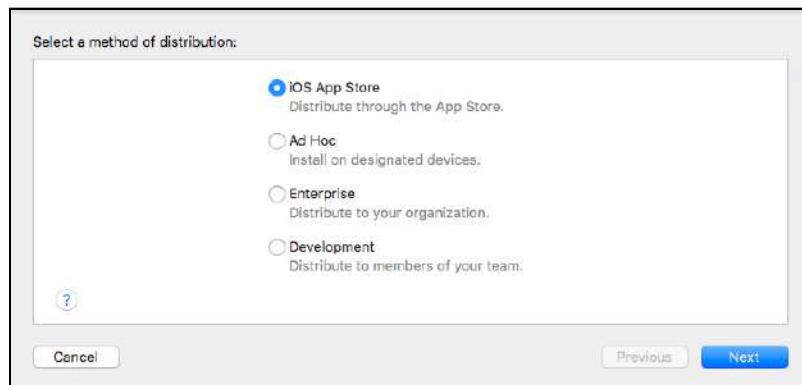
The Organizer window

You are now ready to upload the build to the App Store, as the big blue button on the right sidebar testifies.

You can simply click the big blue button, or, you can click the *Validate App* button below it to verify that your app passes all of Apple's initial validations. The validations are run even if you use the *Distribute App* button but the *Validate App* button is an easy way to check your app locally and verify that it passes muster before you upload it to the Apple servers.

► Click the **Distribute App** button.

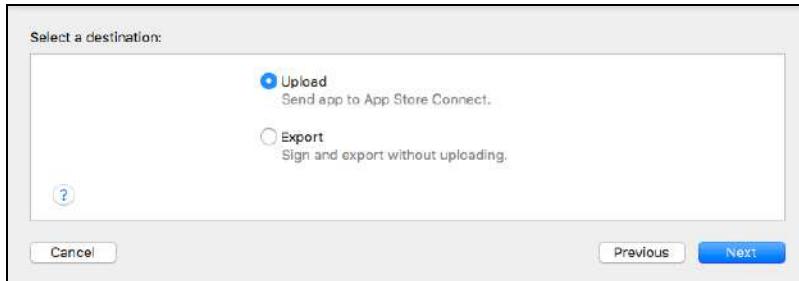
You will be asked for the method of distribution:



App distribution method

You would, of course, want to select **iOS App Store** for this one.

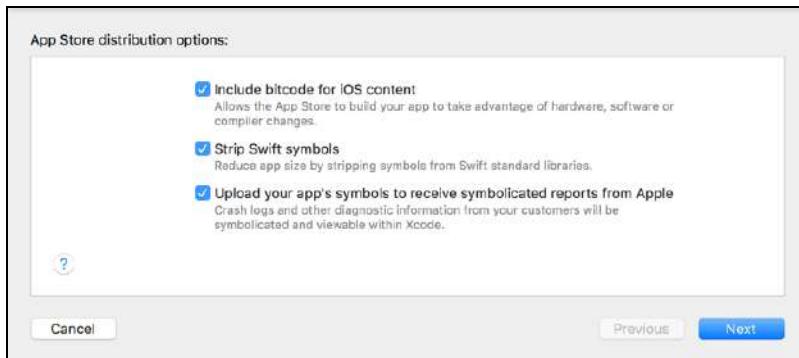
Then you need to select the destination:



Destination options

Here, you can opt to upload the binary file directly to the Apple servers or to create an export file that you can upload later. This option can be useful for situations where you are unable to upload via Xcode for some reason — it happens, more often than you might think.

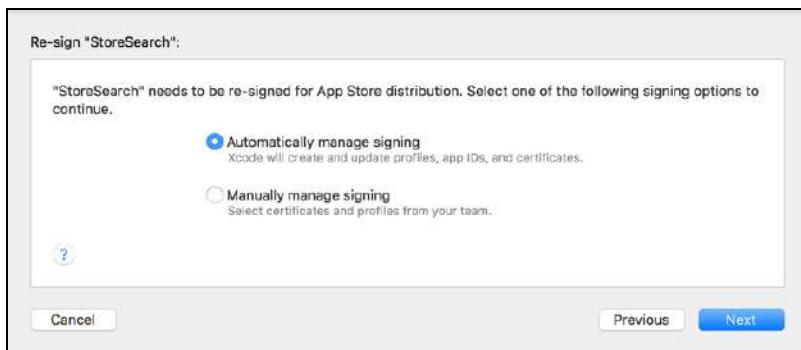
Next, you need to select some distribution options:



App Store distribution options

Each of the options has some helpful text describing what the option does — so you can basically go with the options that are suitable for you. If you are not sure, there should not be any harm in keeping all of the options checked.

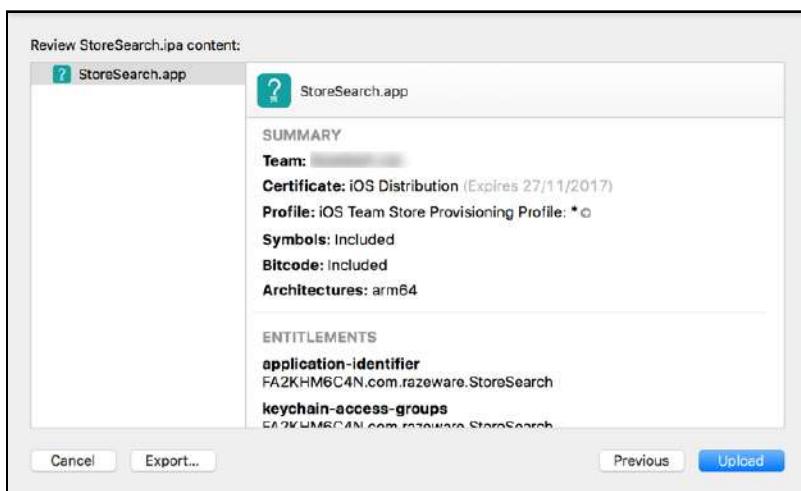
Tapping **Next** will take you to another dialog:



Code signing options

► Generally, it's best to go with **Automatically manage signing** unless you know what you are doing. The manual option gives you a lot more flexibility, but you also have to deal with the complexity that comes along with it — with great power comes great complexity.

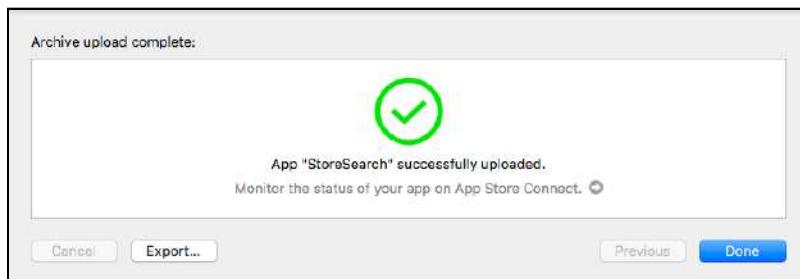
Tap **Next** to proceed and Xcode will work for a while signing your app and getting it ready for upload. When Xcode is ready, it should show you a screen similar to the following:



Ready to upload app

- Click **Upload** and Xcode will start uploading the binary for your app to the Apple servers. Depending on the size of your app — and the speed of your network connection — this might take a bit of time.

While the upload is in progress, you'll get a progress indicator and status messages indicating what is going on. If the upload completes successfully, you should get a message similar to the following:



App Store Connect app upload successful

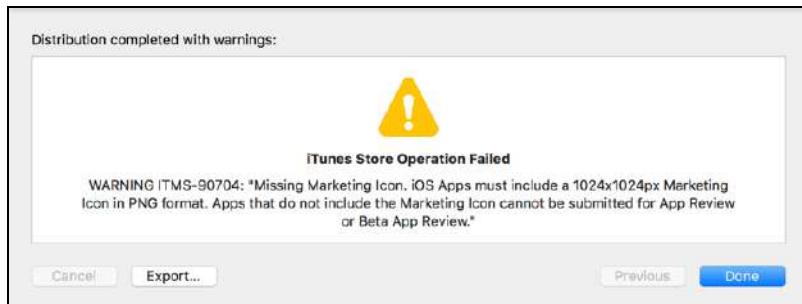
If the upload completes successfully, that's all you have to do at the Xcode end.

Sometimes though, you might get an error.

Some of these errors are basically incomprehensible since they might just be an App Store error code and a cryptic message. If you are unlucky enough to get one of those, you might have to Google for the error code and see if you find somebody else who has figured out the issue. Usually, it turns out to be an Xcode issue or an App Store issue that is resolved by Apple a few days later.

On the other hand, you might get specific error messages such as your app missing the app icon, or an app icon for a specific size that App Store Connect expected. In such cases, fixing the issue by adding the missing assets and then creating another archive — you can't re-use the previous one — and uploading that should resolve the issue.

Sometimes, you might also get warnings, like the following:



App Store Connect error about specific issue

The above indicates that the currently uploaded build is fine as far as passing the general validation goes, but that it's missing an icon. While this is just a warning, as the message indicates, you still cannot use this build for external beta testing or for submitting to the App Store for review. So you'll need to fix the issue and upload a new build.

One thing you have to watch out for when uploading several builds for the same app to the Apple servers is the build number ...

Build number

Each build you submit to Apple has to be uniquely identifiable. How this is generally done for Xcode projects is by combining the version number and build number for the project to get a unique value.

But where is this version number and build number, you ask? Easy enough.

In Xcode, go to your project root in the Project navigator, select your project target, go to the **General** tab, and then check the **Identity** section.

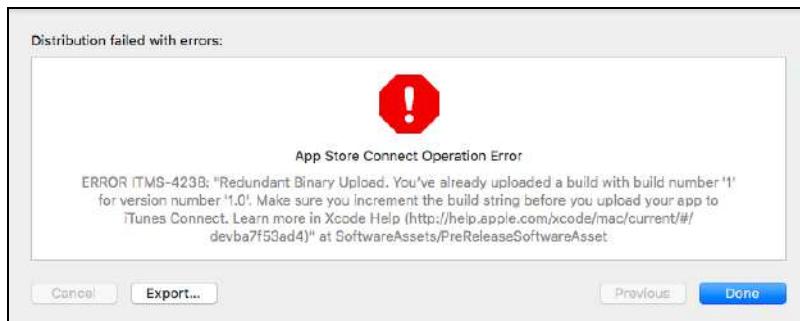


Xcode version and build numbers

For each new version of your app, you need to increment your version number. For each build you submit for a version, the build number has to change too, but, while you cannot repeat a version number, you can repeat a build number as long as the same build number is not used within a given version.

So, for example, you can have the builds for version 1.0 start from 1 and go up to however many builds as you like. And while you cannot use build number 1 for another build for version 1.0, if you start a new version, say 2.0, then you can start the build numbers for the new version again from 1 and go up incrementally.

So, if you've already uploaded a build to App Store Connect and you have to upload another build — either because of an error or because you made a code change — then you need to remember to change the build number. If you don't, you'd get an error during the upload process.



App Store Connect error about build number

Change the build number in your project settings, create a new archive, upload it and you should be good to go!

Check your upload

You can check on the status of your uploaded build by logging into App Store Connect.

- Select **My Apps** from the main dashboard to get a listing of your apps and then select the *StoreSearch* app from there.

- You'll be taken to the app detail screen on App Store Connect:

The screenshot shows the 'App Store Connect' interface with 'My Apps' selected. The main tab is 'App Store'. On the left, under 'APP STORE INFORMATION', 'App information' is selected. In the center, the 'App Information' section is displayed. It includes fields for 'Name' (StoreSearch), 'Privacy Policy URL' (http://example.com (optional)), and 'Subtitle' (Optional). A note states: 'This information is used for all platforms of this app. Any changes will be released with your next app version.' A 'Save' button is at the top right. A sidebar on the left lists 'Pricing and Availability', 'iOS APP', and a build entry: '1.0 Prepare for Submission...'. At the bottom, there's a '+ VERSION OR PLATFORM' button.

The app details on App Store Connect

This is the screen where you would manage everything to do with a given app. You can edit the app information, add screenshots, change the price information, submit builds for review, and check the status of a build.

- Click on the **Activity** tab at the top, on the second row of text from the top. This should take you to the latest activity for this particular app.

The screenshot shows the 'Activity' tab selected. On the left, under 'iOS HISTORY', 'All Builds' is selected. In the center, it says 'iOS Builds' with a note: 'All builds that have been submitted for iOS. Version numbers are the Xcode version numbers.' An orange arrow points to the 'Version 1.0' section. Below it, a table shows the build status: 'Build' (with a question mark icon and '1 (Processing)'), 'Upload Date' (Aug 25, 2018 at 10:47 AM), and 'App Store Status' (grayed out).

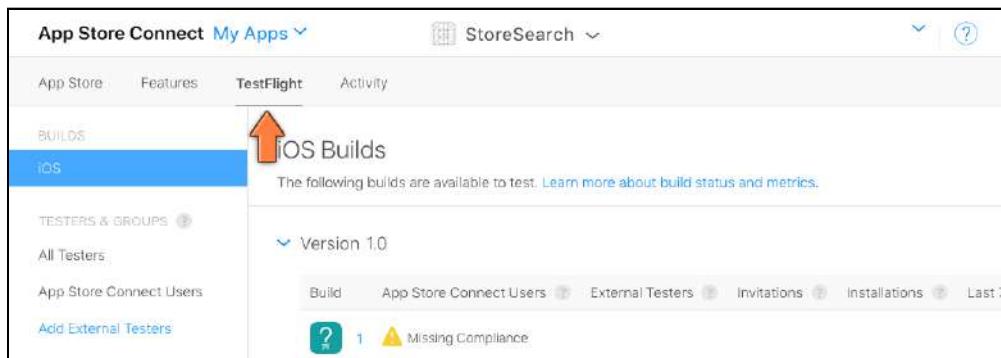
The activity page on App Store Connect

As the screen shows, your app is still processing. This can sometimes take a bit of time, though most of the time, the process is pretty quick. Once the processing is complete, you should receive a notification e-mail from Apple indicating that your app had completed processing and is now available for either testing or distribution.

Internal testing

As was mentioned before, there are two test modes for TestFlight — internal and external. Once your app upload completes processing, you can immediately start internal testing.

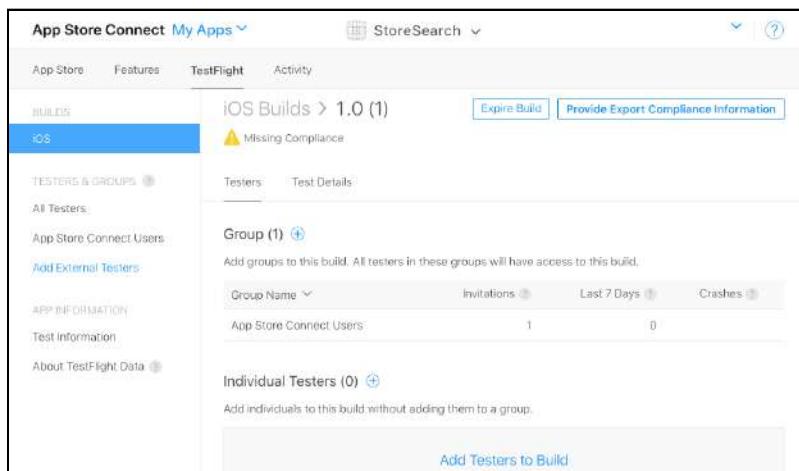
- Log in to App Store Connect and go to your app's detail screen. From there, select **TestFlight** from the list of tabs at the top.



The screenshot shows the App Store Connect interface for an iOS app. The top navigation bar has tabs for App Store, Features, TestFlight (which is selected), and Activity. On the left, a sidebar lists BUILD, TESTERS & GROUPS, and APP INFORMATION sections. Under BUILD, 'iOS' is selected, and under it, 'iOS Builds' is listed. An orange arrow points to the 'iOS Builds' link. Below this, it says 'The following builds are available to test.' A section for 'Version 1.0' is shown with tabs for Build, App Store Connect Users, External Testers, Invitations, Installations, and Last 7. A blue button labeled 'Missing Compliance' with a warning icon and the number '1' is visible.

The TestFlight page on App Store Connect

As you might notice, your app build has a warning — it's apparently missing compliance information. This is standard for all builds unless you provide the compliance information beforehand. The easiest way to fix this is to click on the blue "1" next to the app icon — that is a link which would take you to the detail information page for this particular build — build #1.



This screenshot shows the 'Builds' section of the TestFlight page for build #1. The build summary shows 'iOS Builds > 1.0 (1)' and a 'Missing Compliance' warning. Buttons for 'Expire Build' and 'Provide Export Compliance Information' are present. The 'Test Details' tab is selected, showing a 'Group (1)' section with a table for 'Group Name' (App Store Connect Users) and metrics for 'Invitations' (1), 'Last 7 Days' (0), and 'Crashes' (0). Below this is an 'Individual Testers (0)' section with a '+ Add' button. At the bottom is a large 'Add Testers to Build' button.

The build detail page on App Store Connect

There, at the top of the page, is a big **Provide Export Compliance Information** button! Click on the button, go through another screen where you specify whether your app uses encryption or not, and you should be done with the export compliance stuff till you upload another build.

The above screen also has a place to add testers for this build. So, you might think that is where you add testers for your app. Well ... yes, and no.

To add *internal* testers for your app, you actually have to go to the **App Store Connect Users** link on the left of that screen. That takes you to a new screen from where you can select up to 25 existing users in your team to be added as internal testers.

If you don't have any team members at the moment though, you would need to first go back to your App Store Connect dashboard, select the **Users and Roles** option, add some team members — and optionally, assign them the role of tester — and then come back and add the newly created users as internal testers for your app.

The selected team members will be notified via e-mail that a new build is ready for testing and they will be asked to install the TestFlight iOS app so that they can participate in the testing process.

Internal testing, as the name implies, is generally for testing within your team. So Apple does not require your beta build to go through any sort of review before you start testing. But internal testing is also limited to just 25 people at most.

If you want to do more extensive beta testing, then you have to opt for external testing.

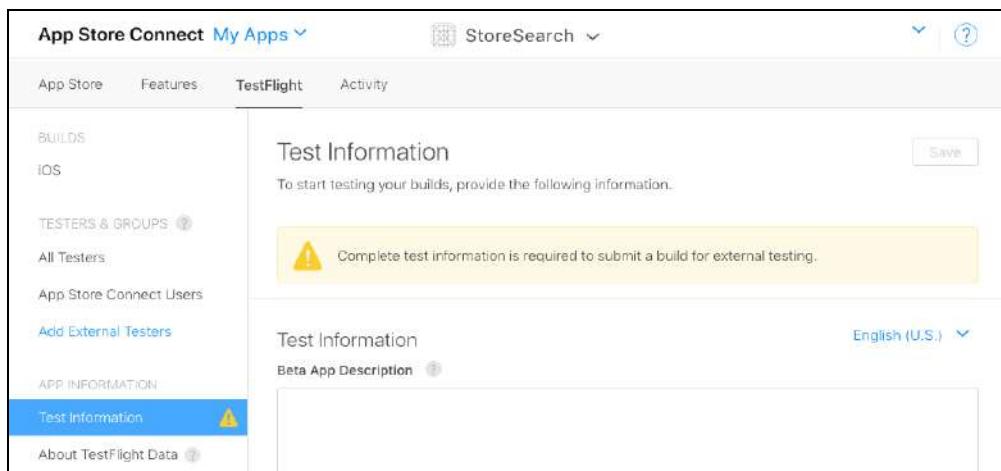
External testing

External testing allows you to distribute beta builds of your app to 10,000 testers. But before you can start inviting testers, you have to get your beta build approved by Apple.

Before you can submit your beta build to Apple for review, you have to fill in the relevant test information for the particular build you want tested.

► Go to App Store Connect, select your app, and on the app detail screen, go to the **TestFlight** tab. There should be an item named **Test information** on the left sidebar. Select it.

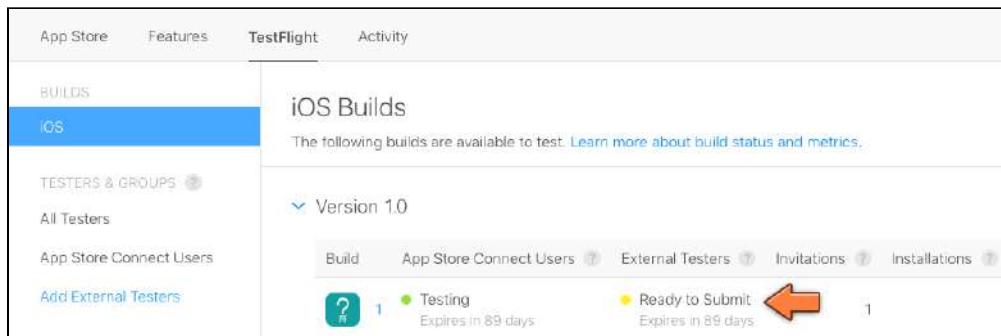




Enter test information on App Store Connect

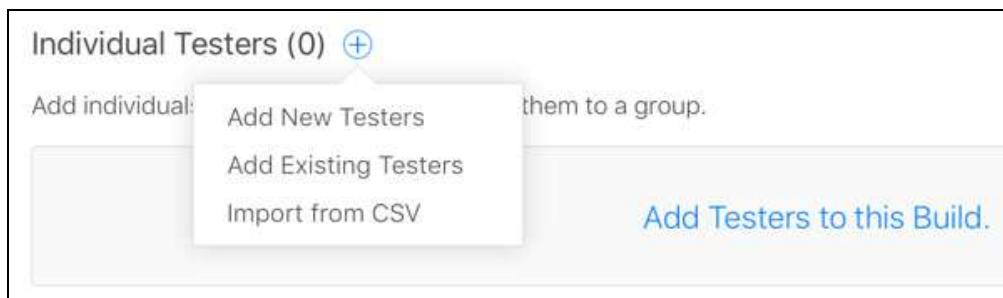
Fill in at least the **Feedback Email** and the whole **Beta App Review Information** section towards the end of the page before the warning at the top about completing test information goes away. However, it is possible that Apple may change these requirements from time to time. Once you are done, click **Save**.

- Go to the **Builds - iOS** item on the left sidebar. Your build should now appear as "Ready to Submit" for the *External Testers* column.



The build is ready for beta review

- Go into the build details screen by clicking the blue "1" next to the app icon.
- Before you can submit the app for beta review, you need to add at least one external beta tester. So, use the **Add Testers to this Build** link towards the bottom of the screen or the blue plus (+) icon above it — use the **Add New Testers** option — to start adding some external testers.



Add external testers for your app

- This shows a new dialog where you can add external testers by e-mail address. Add one or more testers here.

The screenshot shows a dialog titled "Add New Testers to Build 1.0 (1)". It displays a table with columns for Email, First Name, and Last Name. The first row contains data for a tester named Fahim Farook. There are three empty rows below for additional entries. At the bottom, there is a checked checkbox for "Automatically notify testers" and two buttons: "Cancel" and "Next".

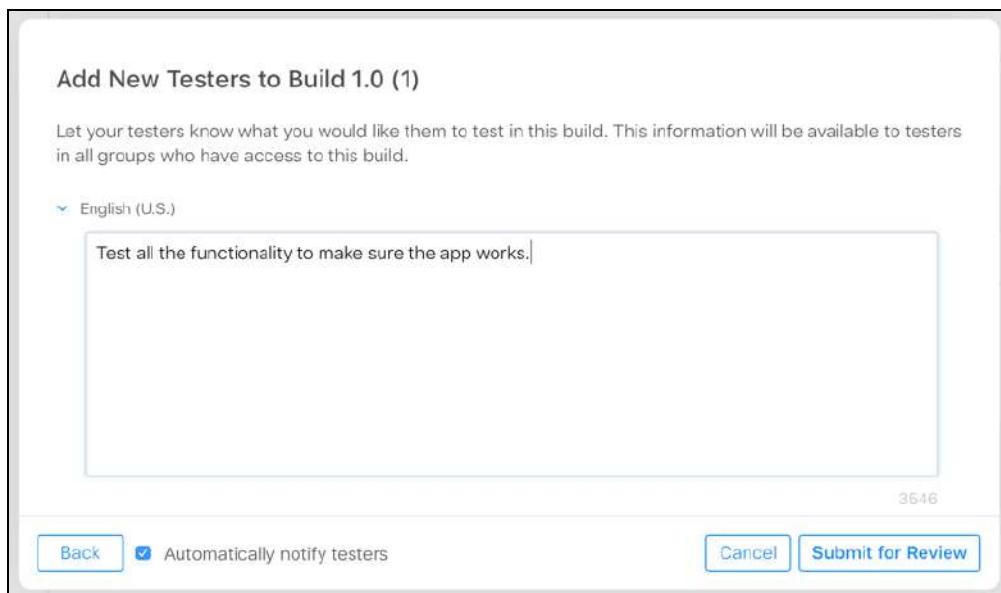
	Email	First Name	Last Name
1	fahimf@...	Fahim	Farook
2			
3			

The dialog for adding new testers

- When you are done adding testers, click the **Next** button.

This shows another screen which asks for sign in information in case your app requires a login.

This is so that the Apple personnel reviewing your beta build have all the necessary information in order to test your app. Since *StoreSearch* does not require any login information, you can indicate that no sign in is required and proceed to the next screen.



The submit for beta review dialog

- Enter some text here explaining what you need tested — and perhaps what changes were made since the last build if this is a subsequent beta build — and then click **Submit for Review** to start the beta review process.

Now, you wait!

Generally, you will hear back from Apple within a day or two. If the Apple review team finds any issues with your app, they will let you know and you would need to fix the issues and submit another build and go through the review process again till you succeed.

If the app passes review, then it will be in "Testing" state for external testing and your external testers will be notified via e-mail that a new build is ready for them to test. This will start off the beta cycle for your app. Congratulations!

At this point, all you have to do is wait for feedback from your beta testers, fix any issues they find or make changes based on beta feedback, release another beta build. You rinse and repeat till you are certain that your app is ready to be submitted to Apple for review in preparation for release.

Submit for review

When your beta testing is complete, you can submit the final build which passed beta testing for App Store review instead of uploading yet another build to App Store Connect. This way, you bypass the potential for accidental introduction of any new bugs when you create a new build.

Since the build was already uploaded to the Apple servers when you uploaded it for beta testing, you simply have to move on to the next stage in the process using the correct build.

At this point, if you only entered the bare minimum information to add an app to App Store Connect, you might need to provide some additional information for the app as well.

► Go to your app's detail screen and then select the **App Store** tab and then the **App Information** option from the left sidebar.

On this screen, make sure that the **Category** values are filled in correctly for your app. You can select any category from the dropdown here but do try to make sure that the categories you select are relevant for your app.

► Select the **Pricing and Availability** option from the left sidebar and set a price for your app. And if you want your app to be available in only specific App Stores around the world, you'd set that here via the Availability section.

You need to do the above two steps only once for every new app. All the other information changes you make below, might have to be done for every new release.

► Next, click on the **Prepare for Submission** link on the left sidebar with a yellow circle next to it. This is where you enter all the relevant information for each version that you submit to the App Store. Here, you'd need to complete at least the following information — you can fill in more values than the listed ones, but these are the mandatory ones:

- You can upload up to ten screenshots and three 30-second movies per device. At a minimum, you need to supply screenshots for the 5.5-inch iPhones, and the 12.9-inch iPad Pro. If you do not provide screenshots or videos for the smaller screen devices, the assets from the larger screen devices will be scaled down for the smaller screens.
- A list of keywords that customers can search for — limited to 100 characters.
- The URL for your support page.

- A description that will be visible on the store.
- The build to submit — this is the build that passed your beta testing. You can click on the plus icon (or use the link in the box) to get a list of uploaded builds and select the correct one from there.
- A 1024×1024 icon image. This image is automatically picked from the binary for builds created with Xcode 9 or later. If you happen to build your app with an older version of Xcode, you would need to upload the icon yourself.
- Copyright information.
- The version number.
- The app rating — this is to identify whether your app contains potentially offensive content. You have to select from a list of items to determine your final content rating.
- Your contact details. Apple will contact you at this address if there are any problems with your submission.
- Sign-in information. If your app requires a user login to test its functionality, provide the necessary demo user name and password here. If a demo login is not required, remember to uncheck the **Sign-in required** checkbox. Otherwise, you will get an error when you try to submit the app.
- Notes for the reviewer. These are optional, but it's good to provide some notes if the reviewer needs to do anything special in order to test your app.
- When your app should become available

If your app supports multiple languages, then you can also supply a translated description, screenshots and even an application name.

For more info and help, consult the guides available under Resources and Help on the home page.

Make a good first impression

People who are searching or browsing the App Store for cool new apps generally look at things in this order:

1. The name of the app. Does it sound interesting or like it does what they are looking for?

2. The icon. You need to have an attractive icon. If your icon sucks, your app probably does too. Or at least that's what people probably think and then they're gone.
3. The screenshots. You need to have good screenshots that are exciting — make it clear what your app is about. A lot of developers go further than just regular screenshots; they turn these images into small billboards for their app.
4. App preview videos. Create up to three 15 to 30-second videos that show off the best features of your app.
5. If you didn't lose the potential customer in the previous steps, they might finally read your description for more info.
6. The price. If you've convinced the customer they really can't live without your app, then the price usually doesn't matter that much anymore.

So, get your visuals to do most of the selling for you. Even if you can't afford to hire a good graphic designer to do your app's user interface, at least invest in a good icon. It will make a world of difference in sales.

After filling out all the fields, click the **Save** button at the top. When you're ready to submit the app, press **Submit for Review**.

If you missed any information, you will get an error message at the top of the screen indicating the errors. The error message will be accompanied by a link which takes you to the page with the missing (or invalid) information. Also, the fields with missing information will be highlighted in red or have a red circle with an exclamation point next to the title (or both). This will help you identify what information needs to be filled in, or corrected.

Once you fix the issues, save and try submitting again. Sometimes, you have to do this multiple times before you finally succeed!

Once you successfully submit your app, it enters the App Store approval process. If you're lucky, the app will go through in a few days, if you're unlucky it can take several weeks. These days the wait time is fairly short. See <http://appreviewtimes.com> for an indication of how long you might have to wait.

If you find a major bug in the mean time, you can reject the file you uploaded on App Store Connect and upload a new one, but this will put you back at square one and you'll have to start at the bottom of the app review queue once again.

If, after your app gets approved, you want to upload a new version of your app, the steps are largely the same. You change the version in Xcode (and change the build number), upload the new version to App Store Connect, update the information in the Prepare for Submission screen and re-submit.

Updates take about the same amount of time to get reviewed as new apps, so you'll always have to be patient for a few days.

The end

Awesome, you've done it! You made it all the way through *The iOS Apprentice*. It's been a long journey but I hope you have learned a lot about iOS programming, and software development in general. I had a lot of fun writing these chapters and I hope you had a lot of fun reading them!

Because this book is packed with tips and information, you may want to go through it again in a few weeks, just to make sure you've picked up on everything!

The world of mobile app development now lies at your fingertips. There is a lot more to be learned about iOS and I encourage you to read the official documentation — it's pretty easy to follow once you understand the basics. And play around with the myriad of APIs that the iOS SDK has to offer.

Most importantly, go write some apps of your own!

Credits for *StoreSearch*: The shopping cart from the app icon is based on a design from the Noun Project (thenounproject.com).



Want to learn more?

There are many great videos and books out there to learn more about iOS development. Here are some suggestions for you to start with:

- The iOS Developer Library has the full API reference, programming guides, and sample code: developer.apple.com/develop/
- Mobile Human Interface Guidelines (the “HIG”): developer.apple.com/design/human-interface-guidelines
- iOS App Programming Guide: developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html
- View Controller Programming Guide: https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/#//apple_ref/doc/uid/TP40007457-CH2-SW1
- The WWDC videos. WWDC is Apple’s yearly developer conference and the videos of the presentations can be watched online at developer.apple.com/videos/. They are really worth it!
- The team at raywenderlich.com and I also have several other books for sale, including more advanced tutorials on iOS development and game programming on iOS. If you’d like to check these out, visit our store here: store.raywenderlich.com

Stuck?

If you are stuck, ask for help. Sites such as Stack Overflow ([stack overflow .com](https://stackoverflow.com)), the Apple Developer Forums (forums.developer.apple.com), and iPhoneDevSDK (www.iphonedevsdk.com/forum/) are great — and let’s not forget our own forums (forums.raywenderlich.com).

I often go on Stack Overflow to figure out how to write some code. I usually more-or-less know what I need to do — for example, resize a `UIImage` — and I could spend a few hours figuring out how to do it on my own. However, the chances are someone else already wrote a blog post about it. Stack Overflow has tons of great tips on almost anything you can do with iOS development.



However, please don't post questions like this:

"i am having very small problem i just want to hide load more data option in tableview after finished loading problem is i am having 23 object in json and i am parsing 5 obj on each time at the end i just want to display three object without load more option."

This is an actual question that I copy-pasted from a forum. That guy isn't going to get any help because a) his question is unreadable; b) he isn't really making it easy for others to help him.

Here are some pointers on how to ask effective questions:

- Getting Answers http://www.mikeash.com/getting_answers.html
- What Have You Tried? <http://mattgemmell.com/what-have-you-tried/>
- How to Ask Questions the Smart Way <http://www.catb.org/~esr/faqs/smарт-questions.html>

And that's a wrap!

I hope you learned a lot through the *iOS Apprentice*, and that you take what you've learned to go forth and make some great apps of your own.

Above all, *have fun programming*, and let us know about your creations!

— Eli Ganim & Joey deVilla





Conclusion

We hope you're excited about the new world of iOS development that lies before you!

By completing this book, you've given yourself the knowledge and tools to create your own iOS applications, or even start working with other developers on a team. It's up to you now to couple your creativity with the things you've learned in this book and create some impressive apps of your own!

If you have any questions or comments about the projects in this book or in your own iOS apps, please stop by our forums at <http://forums.raywenderlich.com>.

Thank you again for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

— Matthijs, Joey, Eli and Adam

The *iOS Apprentice* team

