



iOS 11 & Swift 4 for Beginners

The Course Companion

By the raywenderlich.com Tutorial Team

Fahim Farook, Matt Galloway, Eli Ganim, Matthijs Hollemans
Ben Morrow, Cosmin Pupăză, and Steven Van Impe

iOS 11 & Swift 4 For Beginners

Fahim Farook, Matt Galloway, Eli Ganim, Matthijs Hollemans, Ben Morrow, Cosmin Pupăză and Steven Van Impe

Copyright ©2017 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.

License

By downloading *iOS 11 & Swift 4 For Beginners*, you have the following license:

- You are allowed to use and/or modify the source code in *iOS 11 & Swift 4 For Beginners* in as many apps as you want, with no attribution required.
- The source code included in *iOS 11 & Swift 4 For Beginners* is for your personal use only. You are NOT allowed to distribute or sell the source code in *iOS 11 & Swift 4 For Beginners* 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.

About the authors



Fahim Farook is a developer with over 25 years of experience in developing in over a dozen different languages. Fahim's current focus is on mobile development with over 80 iOS apps and a few Android apps under his belt. He has lived in Sri Lanka, USA, Saudi Arabia, New Zealand, Singapore, Malaysia, France, and the UAE and enjoys science fiction and fantasy novels, TV shows, and movies. You can follow Fahim on Twitter at @FahimFarook.



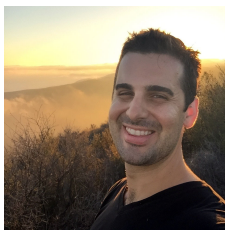
Matt Galloway is a software engineer with a passion for excellence. He stumbled into iOS programming when it first was a thing, and has never looked back. When not coding, he likes to brew his own beer.



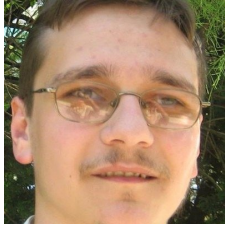
Eli Ganim is an engineer at Facebook. He is passionate about teaching, writing, and sharing his knowledge with others.



Matthijs Hollemans 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.



Ben Morrow delights in discovering the unspoken nature of the world. He'll tell you the surprising bits while on a walk. He produces beauty by drawing out the raw wisdom that exists within each of us.



Cosmin Pupăză is a software developer and tutorial writer from Romania and this is his first book. He is part of the Swift tutorial teams at AppCoda and raywenderlich.com. He has worked with more than a dozen programming languages, technologies, frameworks, tools and libraries over the years, but none of them has made such a great impact on himself as the advent of Swift. When not coding, he either plays the guitar or studies WWII history, and he hopes you enjoy reading the book just as much as he enjoyed writing it. Cosmin blogs about Swift at cosminpupaza.wordpress.com.

About the editors



Steven Van Impe is the technical editor of this book. Steven is a computer science lecturer at the University College of Ghent, Belgium. When he's not teaching, Steven can be found on his bike, rattling over cobblestones and sweating up hills, or relaxing around the table, enjoying board games with friends. You can find Steven on Twitter as [@svanimpe](https://twitter.com/svanimpe).



Chris Belanger is the editor of this book. Chris is the Book Team Lead and Lead Editor for raywenderlich.com. He was a developer for nearly 20 years in various fields from e-health to aerial surveillance to industrial controls. If there are words to wrangle or a paragraph to ponder, he's on the case. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach. Twitter: [@crispytwit](https://twitter.com/crispytwit).



Ray Fix is the final pass editor of this book. Ray Fix is a software developer from Southern California who is passionate about technology, communication, and helping people become their best. In addition to his role as Swift Team Lead for raywenderlich.com, he is an organizer for the Orange County iOS developer group and a board member for the non-profit, San Diego iOS group. Ray is semi-fluent in spoken and written Japanese and stays healthy by walking, jogging and playing ultimate frisbee. When he is not doing one of those things, he is writing and dreaming of code in Swift!

Table of Contents: Overview

Introduction	16
Section I: Your First Swift 4 and iOS 11 App	26
Chapter 1: The One-Button App	27
Chapter 2: Slider and Labels	55
Chapter 3: Outlets	77
Chapter 4: Rounds and Score	96
Chapter 5: Polish	112
Chapter 6: The New Look	126
Chapter 7: The Final App	157
Section II: Programming in Swift	184
Chapter 8: Expressions, Variables & Constants	185
Chapter 9: Types & Operations	211
Chapter 10: Basic Control Flow	228
Chapter 11: Advanced Control Flow	244
Chapter 12: Functions	259
Chapter 13: Optionals	273
Chapter 14: Arrays, Dictionaries, Sets	286
Chapter 15: Collection Iteration with Closures	310

Chapter 16: Strings	323
Chapter 17: Structures.....	338
Chapter 18: Properties.....	350
Chapter 19: Methods	363
Chapter 20: Classes	377
Chapter 21: Advanced Classes	392
Section III: Your Second Swift 4 and iOS 11	
App.....	413
Chapter 22: Table Views	414
Chapter 23: The Data Model.....	447
Chapter 24: Navigation Controllers.....	473
Chapter 25: Add Item Screen.....	496
Chapter 26: Delegates and Protocols.....	517
Chapter 27: Edit Items	532
Chapter 28: Saving and Loading	554
Chapter 29: Lists.....	572
Chapter 30: Improved Data Model.....	602
Chapter 31: User Defaults	623
Chapter 32: UI Improvements	640
Chapter 33: Local Notifications	673
Conclusion.....	704

Table of Contents: Extended

Introduction	16
How to use this book	17
Getting the most out of this course	17
iOS 11 and better only	19
What you need	20
Xcode	21
Get our free newsletter.....	22
License	22
Acknowledgments	23
About this book	23
About the cover	25
Section I: Your First Swift 4 and iOS 11 App	26
Chapter 1: The One-Button App	27
The one button app	29
The Bull's Eye game	30
The one-button app	32
The anatomy of an app.....	53
Chapter 2: Slider and Labels	55
Portrait vs. landscape.....	56
Objects, data and methods	60
Add the other controls	64
Chapter 3: Outlets	77
Improve the slider	77
Generate the random number	84
Add rounds to the game.....	86
Display the target value	91
Chapter 4: Rounds and Score	96
Get the difference	96

Other ways to calculate the difference	102
What's the score?	105
The total score.....	106
Display the score	108
One more round... ..	109
Chapter 5: Polish	112
Tweaks.....	112
The alert	119
Start over	122
Chapter 6: The New Look.....	126
Landscape orientation revisited	126
Spice up the graphics	128
The About Screen.....	144
Chapter 7: The Final App	157
Support different screen sizes	158
Crossfade.....	170
The icon.....	171
Display name.....	173
Run on device.....	174
The end... or the beginning?	182
Section II: Programming in Swift	184
Chapter 8: Expressions, Variables & Constants ..	185
How a computer works	185
Playgrounds	192
Getting started with Swift.....	196
Printing out	197
Arithmetic operations	198
Math functions	202
Naming data	203
Increment and decrement	207
Key points.....	208

Where to go from here?	209
Challenges.....	209
Chapter 9: Types & Operations	211
Type conversion	211
Strings.....	215
Strings in Swift.....	217
Tuples	221
A whole lot of number types	223
A peek behind the curtains: Protocols	224
Key points.....	225
Where to go from here?	226
Challenges.....	226
Chapter 10: Basic Control Flow.....	228
Comparison operators.....	228
The if statement	232
Loops	237
Key points	241
Where to go from here?	241
Challenges.....	242
Chapter 11: Advanced Control Flow	244
Countable Ranges	244
For loops	245
Switch statements	251
Key points.....	256
Where to go from here?.....	257
Challenges.....	257
Chapter 12: Functions	259
Function basics	259
Functions as variables.....	266
Key points.....	269
Where to go from here?.....	270

Challenges	270
Chapter 13: Optionals.....	273
Introducing nil.....	273
Introducing optionals.....	275
Unwrapping optionals.....	277
Introducing guard.....	280
Nil coalescing	282
Key points.....	283
Where to go from here?	283
Challenges	284
Chapter 14: Arrays, Dictionaries, Sets	286
Mutable versus immutable collections.....	286
Arrays	286
What is an array?.....	287
When are arrays useful?.....	287
Creating arrays	287
Accessing elements	288
Modifying arrays.....	292
Iterating through an array	295
Running time for array operations.....	296
Key points	297
Dictionaries	298
Creating dictionaries	298
Accessing values.....	299
Modifying dictionaries.....	300
Iterating through dictionaries.....	302
Running time for dictionary operations.....	303
Key points.....	304
Sets	304
Creating sets	304
Accessing elements	305
Adding and removing elements.....	305
Running time for set operations.....	305

Key points.....	306
Where to go from here?	306
Challenges	306
Chapter 15: Collection Iteration with Closures	310
Closure basics	310
Custom sorting with closures	315
Iterating over collections with closures	315
Key points.....	320
Where to go from here?	320
Challenges	321
Chapter 16: Strings.....	323
Strings as collections	323
Strings as bi-directional collections.....	328
Substrings	329
Encoding	330
Key points.....	335
Where to go from here?	336
Challenges	336
Chapter 17: Structures	338
Introducing structures.....	338
Accessing members	343
Introducing methods.....	344
Structures as values.....	345
Structures everywhere.....	346
Conforming to a protocol	347
Key points	348
Where to go from here?	348
Challenges	349
Chapter 18: Properties	350
Stored properties	350
Computed properties	352

Type properties	356
Property observers.....	357
Lazy properties	359
Key points	361
Where to go from here?	361
Challenges.....	362
Chapter 19: Methods	363
Method refresher	363
Introducing self	366
Introducing initializers	367
Introducing mutating methods	370
Type methods	371
Adding to an existing structure with extensions	372
Key points	374
Where to go from here?	375
Challenges.....	375
Chapter 20: Classes	377
Creating classes	377
Reference types	378
Understanding state and side effects.....	386
Extending a class using an extension.....	387
When to use a class versus a struct	388
Key points.....	389
Where to go from here?	390
Challenges	390
Chapter 21: Advanced Classes.....	392
Introducing inheritance	392
Inheritance and class initialization	400
When and why to subclass	405
Understanding the class lifecycle	407
Key points.....	411
Where to go from here?	411

Challenges.....	411
Section III: Your Second Swift 4 and iOS 11	
App.....	413
Chapter 22: Table Views.....	414
Table views and navigation controllers.....	415
The Checklists app design.....	417
Add a table view	418
The table view delegates	430
Chapter 23: The Data Model	447
Model-View-Controller.....	447
The data model	449
Clean up the code	470
Chapter 24: Navigation Controllers	473
Navigation controller	474
Delete rows	483
The Add Item screen.....	485
Chapter 25: Add Item Screen	496
Static table cells	497
Read from the text field.....	503
Polish it up.....	506
Chapter 26: Delegates and Protocols	517
Add new ChecklistItems	517
Chapter 27: Edit Items.....	532
Edit items.....	532
Refactor the code	546
One more thing	549
Chapter 28: Saving and Loading	554
The need for data persistence	554
The documents folder	555

Save checklist items.....	559
Load the file	566
What next?	571
Chapter 29: Lists	572
The All Lists view controller	573
The All Lists UI.....	581
View the checklists.....	587
Manage checklists	591
Are you still with me?.....	601
Chapter 30: Improved Data Model	602
The new data model.....	603
Fake it 'til you make it.....	606
Do saves differently	611
Improve the data model.....	618
Chapter 31: User Defaults	623
Remember the last open list	623
Defensive programming	630
The first-run experience	635
Chapter 32: UI Improvements.....	640
Show counts	641
Sort the lists.....	648
Add icons	650
Make the app look good.....	662
Support all iOS devices.....	665
Chapter 33: Local Notifications	673
Try it out	674
Set a due date	678
Due date UI	683
Schedule local notifications	696
That's a wrap!	702

Conclusion.....	704
-----------------	-----

Introduction

By Ray Wenderlich

Thank you for signing up for our iOS 11 and Swift 4 for Beginners course!

In this course, you'll learn everything you need to know to make your own apps. By the end of the course, you'll be experienced enough to turn your ideas into real apps that you can put on the App Store.

Even if you've never programmed before or if you're new to iOS, you should be able to follow along with the step-by-step tutorials and understand how each app is made. Our videos show and explain every step to prevent you from getting lost. Not everything will make sense right away, but hang in there and all will become clear in time.

This special book corresponds to the first 16 sections of our course, and is available exclusively available for students enrolled in the course. It will serve as a handy reference as you're working through the course, and has some additional details and exercises that you may find handy.

Let's take a quick tour of what's inside:

- This first section of this book corresponds to the **Your First Swift 4 and iOS 11 App** sections of the course. Here you'll create your first app: a simple but fun iPhone game called Bull's Eye.
- The second section of this book corresponds to the **Programming in Swift** sections of the course. Here you'll learn about fundamental Swift programming concepts, like variables, loops, collections, structures, classes, optionals, closures, and more.
- The third section of this book corresponds to the **Your Second Swift 4 and iOS 11 App** sections of the course. Here you'll create your second app, called checklists, where you'll learn about table views and navigation controllers.

That's where this book ends. But don't worry — the fun continues back in the iOS 11 and Swift 4 for Beginners course, where you'll dive into more advanced topics like saving data, Auto Layout, Collection Views, Scroll Views, networking, firebase, git, Xcode Tips & Tricks, and much more!

How to use this book

This book is meant to be a companion to the iOS 11 and Swift 4 for Beginners course. That means we recommend the following:

1. **Watch the videos** in the course first. Follow along with the hands-on tutorials, and try the challenges, and see if you feel comfortable with the material.
2. **If you feel confused or unsure** about any sections, then read the corresponding chapters in this book for additional details and practice. We're strong believers in learning via repetition, so the additional practice should help it sink in.

This book will act as a great reference after you finish the course and are making your own apps. It's sometimes easier to flip through a book to find something you forgot, than to flip through a bunch of videos.

Getting the most out of this course

This course 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 course.

Learn through repetition

You're going to make several apps in this course. Even though the apps will start out quite simple, you may find the instructions hard to follow at first, especially if you've never done any computer programming before, because we will be introducing 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 course, you'll go over many of these concepts again and again until they solidify in your mind.

Follow the instructions yourself

It is important that you don't just watch the videos (or read this book) but also actually **follow along with us**. Open Xcode, type in the source code fragments, and run the app in the Simulator. This helps you to see how the app gets built step by step.

Even better, play around with the code. Feel free to modify any part of the app and see what the results are. 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.

Don't panic – bugs happen!

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

There is nothing wrong with looking on the web for solutions — we do it all the time. But we want to give you the tools and knowledge to understand what you're doing and why. That way you'll learn faster and write better programs.

This course contains hands-on practical advice, not just a bunch of dry theory (although we can't avoid *some* theory). You are going to build real apps right from the start and we'll explain how everything works along the way.

We will do our best to make it clear how everything fits together, why we do things a certain way, and what the alternatives are.

Do the challenges

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

We 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 will build up your understanding of programming while making fun apps. By the end of this course, you'll have learned the essentials of Swift and the iOS development kit. More importantly, you should have a pretty good idea of how everything goes together and how to think like a programmer.

It is our aim that, by the time you reach the end of the course, you will have learned enough to stand on your own two feet as a developer. We are confident that eventually you'll be able to write any iOS app you want — as long as you get those basics down. You still may have a lot to learn, but when you're through with the *iOS 11 and Swift 4 for Beginners* course, you can do without the training wheels.

iOS 11 and better only

The code in this course is aimed exclusively at iOS version 11 and later. Each new release of iOS is such a big departure from the previous one that it just doesn't make sense anymore to keep developing for older devices and iOS versions. Things move fast in the world of mobile computing!

The majority of iPhone, iPod touch, and iPad users are pretty 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, such as the iPhone 4S, iPhone 5, or the first iPads, 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 always relentlessly looks towards the future – if you want to play in Apple's backyard, it's wise to follow their 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 will have to invest in the following:

iPhone, iPad, or iPod touch. We're assuming that you have at least one of these. iOS 11 runs on the following devices: iPhone 5s or newer, iPad 5th generation or newer, iPad mini 2 or newer, 6th generation iPod touch - basically any device which has a 64-bit processor. With iOS 11, Apple dropped support for 32-bit processors and apps. 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 the Simulator.

Note: Even though we mostly talk about the iPhone in this course, everything we say applies equally to the iPad and iPod touch. Aside from small hardware differences, they all use iOS and you program them in exactly the same way. You should also be able to run the apps from this course on your iPad or iPod touch without problems.

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 10.12.4 Sierra. Xcode, the development environment for iOS apps, is a memory-hungry tool. So, having 4 GB of RAM in your Mac is no luxury. You might be able to get by with less, but do yourself a favor and upgrade your Mac. The more RAM, the better. A smart developer invests in good tools!

With some workarounds it is possible to develop iOS apps on Windows or a Linux machine, or a regular PC that has macOS installed (a so-called "Hackintosh"), but you'll save yourself a lot of 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. Just make sure it meets the minimum requirements (Intel CPU, preferably more than 2 GB RAM). Should you happen to buy a machine that has an older version of OS X (10.11 El Capitan or earlier), you can upgrade to the latest version of macOS from the online Mac App Store for free.

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 will 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 in your source code and it has 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 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 an up-to-date version of macOS Sierra (10.12.4), so 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 about 5 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 are encouraged to always develop with the latest Xcode and the latest available SDK on the latest version of macOS.

We wrote this course with **Xcode version 9** and the **iOS 11** SDK on macOS Sierra (10.12.6). By the time you're following along with our course, the version numbers might have gone up again. We will do my best to keep the course 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 courses and books (anything before 2010) talk about Xcode 3, which is radically different from Xcode 9. More recent material may mention Xcode versions 4, 5, 6, 7, or 8, which at first glance are similar to Xcode 9 but differ in many of the details. So if you're reading an article and you see a picture of Xcode that looks different from yours, they might be talking 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.

Get our free newsletter

Want to get notified of all the latest news at raywenderlich.com?

We publish a free, weekly newsletter that includes a list of new tutorials on raywenderlich.com, important news like new books and new video courses, and a list of our favorite iOS development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

License

You have the following license for the materials in this book:

- You are allowed to use and/or modify the source code in *iOS 11 and Swift 4 for Beginners* 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 11 and Swift 4 for Beginners* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *iOS 11 and Swift 4 for Beginners*, available at www.raywenderlich.com”.
- The source code included in *iOS 11 and Swift 4 for Beginners* is for your personal use only. You are NOT allowed to distribute or sell the source code in *iOS 11 and Swift 4 for Beginners* 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.

Acknowledgments

We would like to thank many people for their assistance in making this book and course possible:

- **Our families:** For bearing with us in this crazy time as we worked all hours of the night to get this book ready for publication!
- **Everyone at Apple:** For producing the amazing hardware and software we know and love, and for creating an exciting new programming language that we can use to make apps for that hardware!
- **The Swift Community** For all the people, both inside and outside of Apple, who have worked very hard to make Swift the best computer language in the world.
- And most importantly, **the readers of raywenderlich.com — especially you!** Thank you so much for purchasing our course. Your continued readership and support is what makes all of this possible!

About this book

This book is based on portions of two of our best-selling books:

The iOS Apprentice

[The iOS Apprentice](#) covers how to build 4 complete iOS apps via step-by-step tutorials.



The full version of the iOS Apprentice builds on what you've learned to build even more exciting iOS apps:

- Learn how to build a GPS-based mapping application that saves a user's favorite locations, displays them on a map, uses the iPhone's camera to take pictures and save all this information to a database.
- Then, build an app that uses live data from the iTunes store to help the user browse music selections. You'll learn how to work with remote APIs, how to interpret the data sent back from network requests, and how to display that information back to the user.
- You'll also learn how to make your apps work in multiple languages around the world, how to adopt your apps for the iPad, and even how to submit your apps to the App Store for sale!

You can get the full version of the book here:

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

The Swift Apprentice

[The Swift Apprentice](#): Covers the Swift 4 language in detail through tutorials on fundamental programming concepts.



The full version of the Swift Apprentice goes even deeper with the language to help you become an expert at one of the most modern and fastest-growing programming languages today. You'll learn about advanced language features including the following:

- Enumerations, protocols and generics
- Pattern matching
- Advanced error handling

- Encoding and decoding information
- And much more!

You can get the full version of the Swift Apprentice here:

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

About the cover

Wondering why we have sea creatures on the cover of our books? Here's a little background about the animals featured on the cover:

- **Striped dolphins** (from the *iOS Apprentice*) 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?
- **Flying fish** (from the *Swift Apprentice*) have been known to soar 655 feet in a single flight, can reach heights of 20 ft above the water, and may fly as fast as 37 mph. If you ever feel like a fish out of water trying to learn Swift, just think about the animals on the cover of this book — if they can adapt to a completely new environment, so can you!

Section I: Your First Swift 4 and iOS 11 App

The first section of this book corresponds to the **Your First Swift 4 and iOS 11 App** sections of the course. In this section, you'll create your first app: a simple but fun iPhone game called Bull's Eye.

As you progress through building this app, you'll learn how to think like a programmer and how to plan your programming tasks. In addition, you'll also learn how to use Xcode, Interface Builder and even the basics of coding for iOS.

While some of the concepts in this section might seem a bit basic, please don't skip this section if you are new to iOS development! It contains a wealth of information about the fundamentals of iOS development which you'll rely heavily on in the future.

Chapter 1: The One-Button App

Chapter 2: Slider and Labels

Chapter 3: Outlets

Chapter 4: Rounds and Score

Chapter 5: Polish

Chapter 6: The New Look

Chapter 7: The Final App

Chapter 1: The One-Button App

By Fahim Farook and Matthijs Hollemans

The iPhone may pretend that it's a phone but it's really a pretty advanced computer that also 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 that the computer can understand.

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 is not precise enough to use for programming computers.

So, you will use an intermediary language, Swift, that is a little bit like English so it's reasonably straightforward for us humans to understand, while at the same time 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    $36, %esp
Ltmp97:
.cfi_offset %esi, -12
calll   L7$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)
calll   _objc_retain
movl    %eax, -16(%ebp)
.loc    1 161 2 prologue_end
```

Actually, what the computer sees is this:

```
000110010100111101001000110011111001010
001010001001111010110111001110101101001
0101000111100111110101110110000111000110
100100000111000101001101001111001100111
```

The `movl` and `calll` instructions are just there to make things 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 – that is what people used to do in the old days when computers cost a few million bucks apiece and took up a whole room – but I'd rather write programs that 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, Swift 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 is not the only option for making apps. Until recently, iOS apps were programmed in Objective-C, which is an object-oriented extension of the tried-and-true C language. Because of its heritage, Objective-C has some rough edges and is not really up to the demands of modern developers. That's why Apple created a new language.

Objective-C will still be around for a while but it's obvious that 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 is very powerful but as a beginning programmer you probably want to stay away from it. I only mention it because C++ can also be used to write iOS apps, and there is an unholy marriage of C++ and Objective-C named Objective-C++ that you may come across from time to time.

I could have started *The iOS Apprentice* with an in-depth exploration of the features of Swift, but you'd probably fall asleep halfway. So instead, I will explain the language as we go along, very briefly at first but more in-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?

The one button app

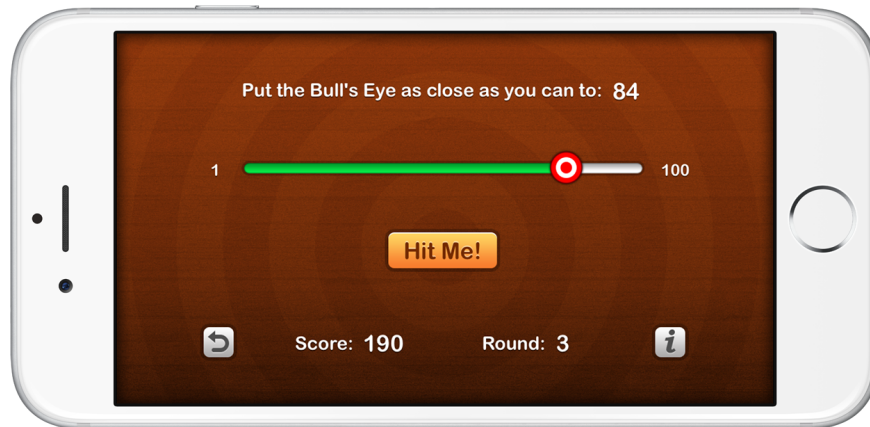
There's an old Chinese proverb that "A journey of a thousand miles begins with a single step." You are about to take that first step on your journey to iOS developer mastery. And you will take that first step by creating the *Bull's Eye* game.

This chapter covers the following:

- **The Bull's Eye game:** An introduction to the first app you'll make.
- **The one-button app:** Creating a simple one-button app where 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 Bull's Eye game

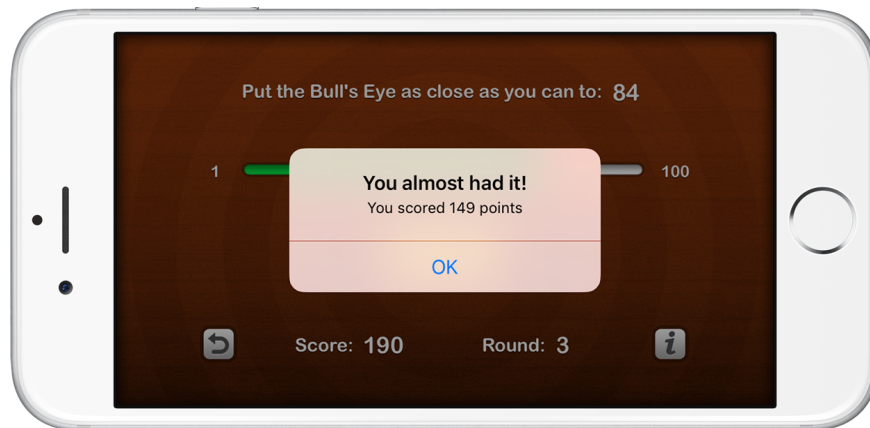
This is what the *Bull's Eye* game will look like when you're finished:



The finished Bull's Eye game

The objective of the game is to put the bull's eye, 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 bull's eye 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 popup, also known as an alert, will tell you what your score is:



An alert popup shows the score

The closer to the target value you are, the more points you score. After you dismiss the alert popup 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 curly arrow in the bottom-left corner), which resets the score to 0.

This game probably won't make you an instant millionaire on the App Store, but even future millionaires have to start somewhere!

Make a programming to-do list

Exercise: Now that you've seen what the game will 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 OK if you draw a blank, but give it a shot anyway.

I'll give you an example:

The app needs to put the "Hit Me!" button on the screen and show an alert popup 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 is not 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 I start working on a new app, I first make a list of all the different pieces of functionality I think the app will need. This becomes my 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.

You may have a cool idea for an app but when you sit down to write the program the whole thing can seem overwhelming. There is so much to do... and where to begin? By cutting up the workload into small steps you make the project less daunting – you can always find a step that is simple and small enough to make a good starting point and take it from there.

It's no big deal if this exercise is giving you difficulty. You're new to all of this! As your understanding grows of how software and the development process works, it will become easier to identify the different parts that make up a design, and to split it into manageable pieces.

This is what I came up with. I simply took the gameplay description and cut it into very small chunks:

- Put a button on the screen and label it "Hit Me!"

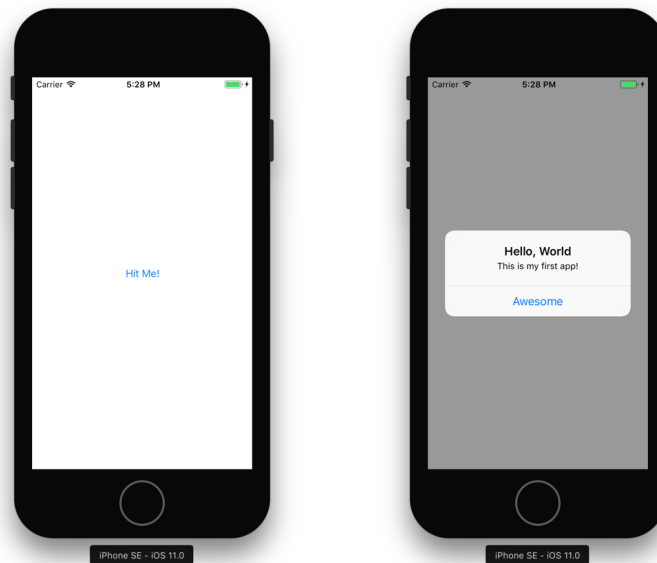
- When the player presses the Hit Me button the app has to show an alert popup to inform the player how well they did. Somehow you have to calculate the score and put that into this alert.
- Put text on the screen, such as the “Score:” and “Round:” labels. Some of this text changes over time, for example the score, which increases when the player scores points.
- Put a slider on the screen and make it go 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 popup.
- Put the Start Over button on the screen. Make it reset the score and put the player back into the first round.
- Put the app in landscape orientation.
- Make it look pretty. :]

I might have missed a thing or two, but this looks like a decent list to start with. Even for a game as basic as this, there are quite a few things you need to do. Making apps is fun, but it’s definitely a lot of work too!

The one-button app

Let’s 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)

Time to start coding! I'm assuming you have downloaded and installed the latest version of the SDK and the development tools at this point.

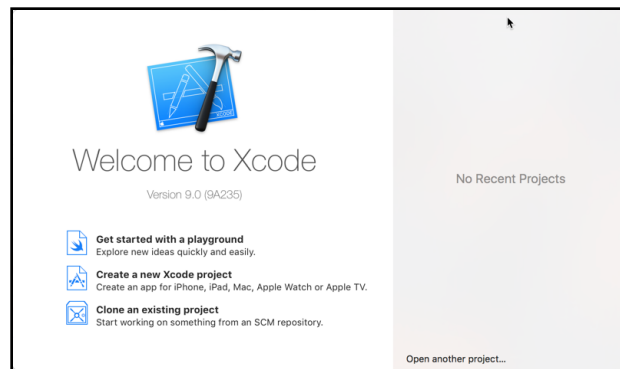
In this book, you'll be working with **Xcode 9.0** or better. Newer versions of Xcode may also work but anything older than version 9.0 probably would be a no-go.

Because Swift is a very new language, it tends to change between versions of Xcode. If your Xcode is too old – or too new! – then not all of the code in this book may work properly. (For this same reason you're advised not to use beta versions of Xcode, only the official one from the Mac App Store.)

Create a new project

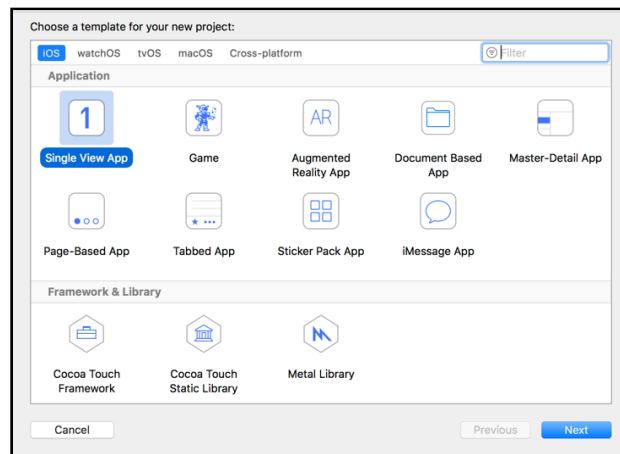
► Launch Xcode. If you have trouble locating the Xcode application, you can find it in the folder **/Applications/Xcode** or in your Launchpad. Because I use Xcode all the time, I placed its icon in my dock for easy access.

Xcode shows 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:

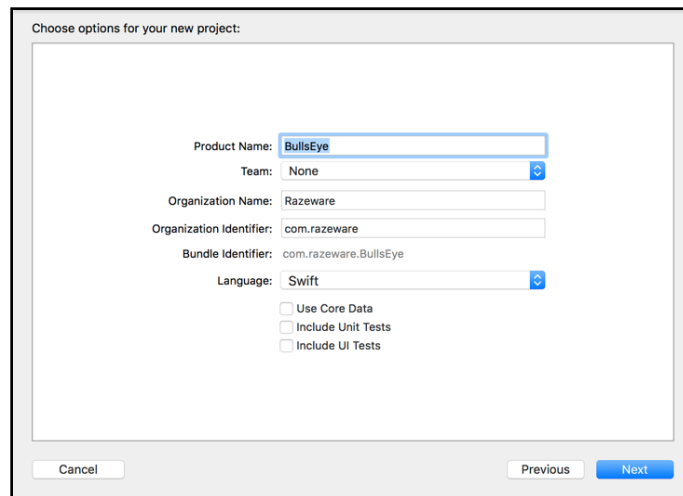


Choosing the template for the new project

Xcode has bundled templates for a variety of application styles. Xcode will make a pre-configured project for you based on the template you choose. The new project will already include some of the source files you need. These templates are handy because they can save you a lot of typing. They are ready-made starting points.

► Select **Single View Application** and press **Next**.

This opens a screen where you can enter options for the new app:



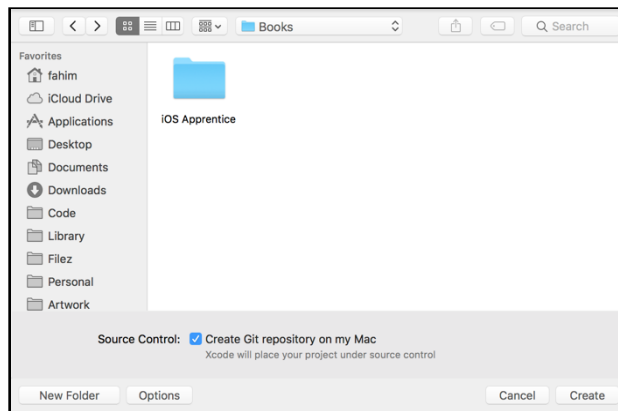
Configuring the new project

► Fill out these options as follows:

- **Product Name: BullsEye.** If you want to use proper English, you can name the project Bull’s Eye instead of BullsEye, but it’s best to avoid spaces and other special characters in project names.
- **Team:** If you already are a member of the Apple Developer Program, this will show your team name. For now, it’s best to leave this setting alone; we’ll get back to this later on.
- **Organization Name:** Fill in your own name here or the name of your company.
- **Organization Identifier:** Mine says “com.razeware”. That is the identifier I use for my apps. As is customary, it is my domain name written in reverse. You should use your own identifier here. Pick something that is unique to you, either the domain name of your website (but backwards) or simply your own name. You can always change this later.
- **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 be using those 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.

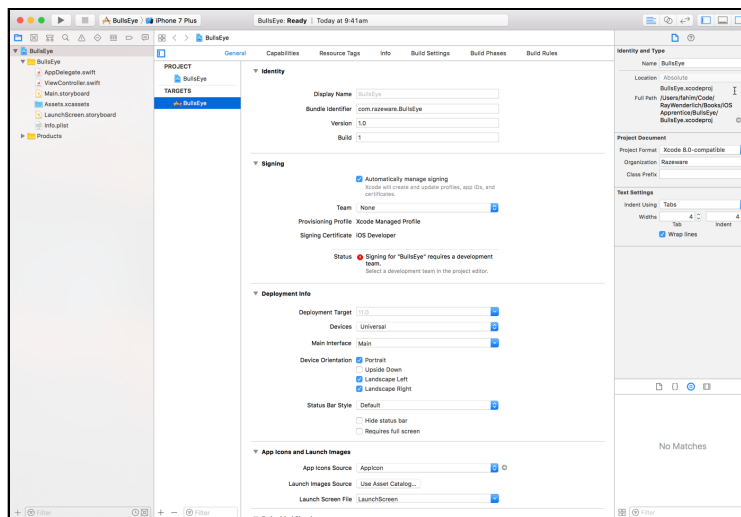
Xcode will automatically make a new folder for the project using the Product Name that you entered in the previous step (in your case BullsEye), so you don't need to make a new folder yourself.

At the bottom there is a checkbox that says, "Create Git repository on My Mac". You can ignore this for now. You'll learn about the Git version control system later on.

► Press **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:



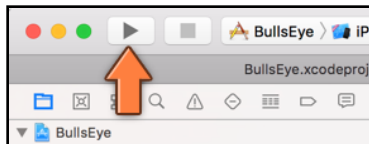
The main Xcode window at the start of your project

There may be small differences with what you're seeing on your own computer if you're using a version of Xcode newer than my version. Rest assured, any differences will only be superficial.

Note: If you don't see a file named `ViewController.swift` in the list on the left but instead have `ViewController.h` and `ViewController.m`, then you picked the wrong language (Objective-C) when you created the project. Start over and be sure to choose Swift as the programming language.

Run your project

► Press the **Run** button in the top-left corner:

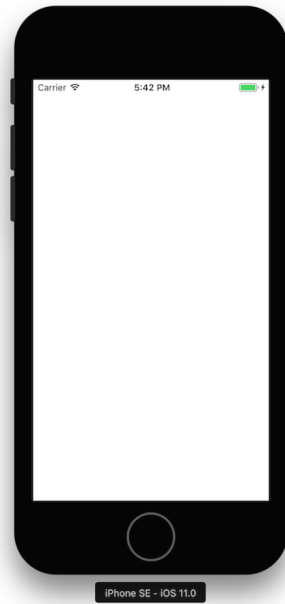


Press Run to launch the app

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.

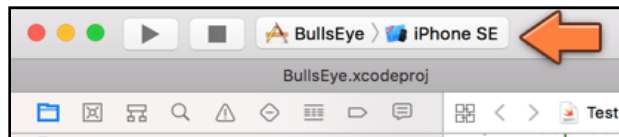
Also, make sure that you do not have your iPhone or iPad plugged in at this point to your computer, for example, for charging. If you do, it might switch to the actual device instead of the Simulator for running the app and since you are not yet set up for running on device, this could result in errors that might leave you scratching your head :]

Xcode will labor for a bit and then launch your brand new app in the iOS Simulator. The app may not look like much yet – and there is not anything you can do with it either – but this is an important first milestone in your journey!



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

If Xcode says “Build Failed” or “Xcode cannot run using the selected device” when you press the Run button, then make sure the picker at the top of the window says **BullsEye > iPhone SE** (or any other model number) and not **Generic iOS Device**:



Making Xcode run the app on the Simulator

If your iPhone is currently connected to your Mac via USB cable, Xcode may have attempted to run the app on your iPhone and that may not work without some additional setting up. I’ll show you how to get the app to run on your iPhone so you can show it off to your friends soon, but for now just stick with the Simulator.

► Next to the Run button is the **Stop** button (the square thingy). Press that to exit the app.

On your phone (or even 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 $\text{⌘} + \text{⌘} + \text{H}$ 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.

Until you press Stop, Xcode’s activity viewer at the top says “Running BullsEye on iPhone SE”:



The Xcode activity viewer

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

What happens when you press Run?

Xcode will first *compile* your source code – that is: translate it – from Swift into machine code that the iPhone (or the Simulator) can understand. Even though the programming language for writing iOS apps is Swift or Objective-C, the iPhone itself doesn’t speak those languages. A translation step is necessary.

The compiler is the part of Xcode that converts your Swift source code into executable binary code. It also gathers all the different components that make up the app – source files, images, storyboard files, and so on – and puts them into the “application bundle”.

This entire process is also known as *building* the app. If there are any errors (such as spelling mistakes for method names), the build will fail. If everything goes according to plan, Xcode copies the application bundle to the Simulator or the iPhone and launches the app. All that from a single press of the Run button.

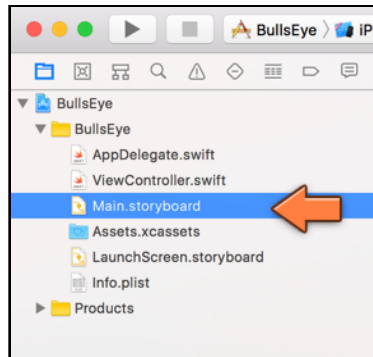
Add a button

I’m sure you’re as unimpressed as I am with an app that just displays a dull white screen :] So, let’s make it a bit more interesting by adding a button to it.

The left pane of the Xcode window is named the **Navigator area**. The row of icons along the top lets you select a specific navigator. The default navigator is the **Project navigator**, which shows the files in your project.

The organization of these files roughly corresponds to the project folder on your hard disk, but that isn’t necessarily always so. You can move files around and put them into new groups and organize away to your heart’s content. We’ll talk more about the different files in your project later.

► 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/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. (OK, bad analogy, but Interface Builder is a super tool in my opinion.)

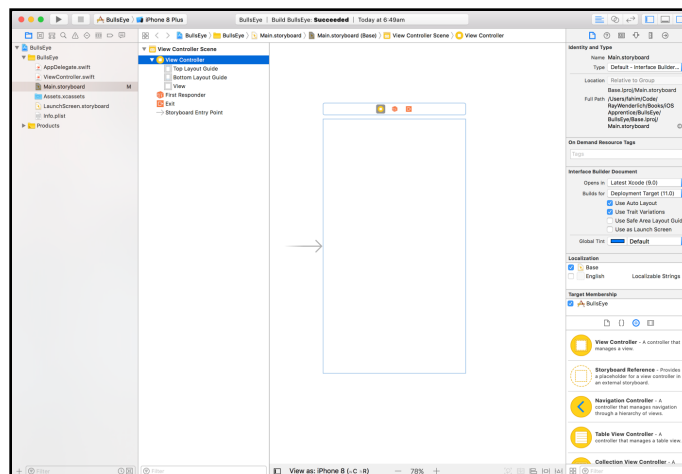
► If it's not already blue, click the **Hide or show utilities** button in Xcode's toolbar:



Click this button to show the Utilities pane

These toolbar buttons 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

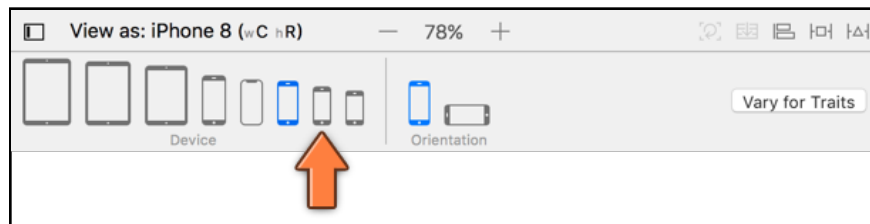
This is the *storyboard* for your app. The storyboard contains the designs for all of your app's screens, and shows the navigation flow in your app from one screen to another.

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 white 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 has the size of an iPhone 8. To keep things simple, you will first design the app for the iPhone SE, which has a slightly smaller screen. Later you'll also make the app fit on the larger iPhone models.

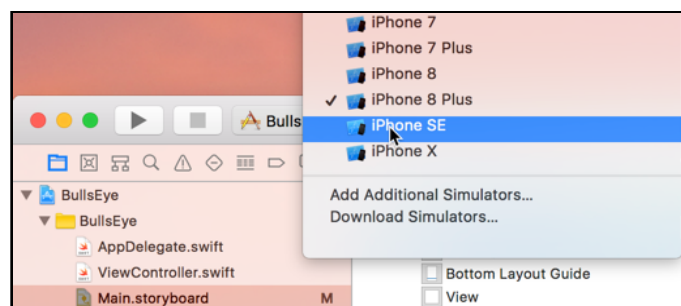
► At the bottom of the Interface Builder window, click **View as: iPhone 8** to open up the following panel:



Choosing the device type

Select the **iPhone SE**, the second smallest iPhone, thus resizing the preview UI you see in Interface Builder to be set to that of an iPhone SE. You'll notice that the scene's rectangle now becomes a bit smaller, corresponding to the screen size of the iPhone 5, iPhone 5s, and iPhone SE models.

► In the Xcode toolbar, make sure it says **Bullseye > iPhone SE** (next to the Stop button). If it doesn't then click it and pick iPhone SE from the list:

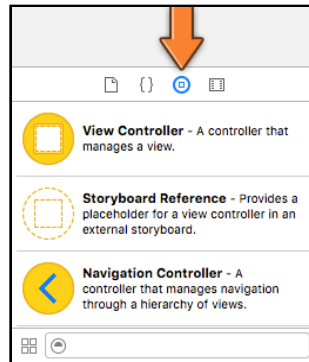


Switching the Simulator to iPhone SE

Now when you run the app, it will run on the iPhone SE Simulator (try it out!).

Back to the storyboard:

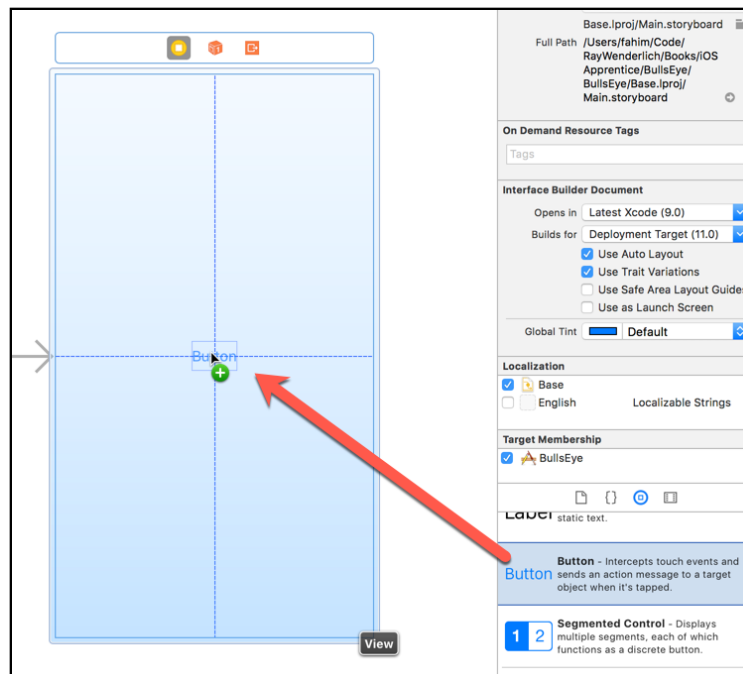
- At the bottom of the Utilities pane you will find the **Object Library** (make sure the third button, the one that looks like a circle, is selected):



The Object Library

Scroll through the items in the Object Library's list until you see **Button**. (Alternatively, you can type the word "button" in to the search/filter box at the bottom of the Object Library.)

- Click on **Button** and drag it into the working area, on top of the scene's rectangle.



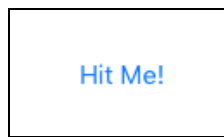
Dragging the button on top of the scene

That's how easy it is to add new buttons, just drag & drop. That goes for all other user interface elements too. You'll be doing 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.

This should give you some idea of the UI controls that are available in iOS. Notice that the Interface Builder helps you to layout 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 has 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 rectangles on or off using the **Editor** → **Canvas** → **Show Bounds Rectangles** menu option.

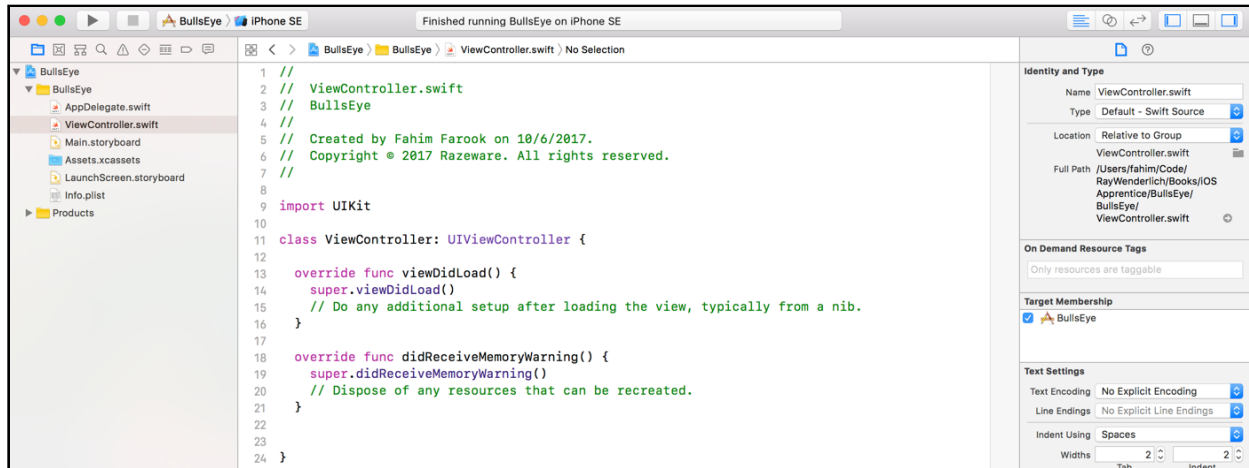
When you're done playing with Interface Builder, press the Run button from Xcode's toolbar. The app should now appear in the Simulator, complete with your "Hit Me!" button. However, when you tap the button it doesn't do anything yet. For that you'll have to write some Swift code!

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 popup. In the finished game the alert will display the player's score, but for now we shall limit ourselves 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:



The source code editor

Note: If your Xcode editor window does not show the line numbers as in the screenshot above, and you'd actually like to see the line numbers, from the menu bar choose **Xcode** → **Preferences...** → **Text Editing** and go to the **Editing** tab. There, you should see a **Line numbers** checkbox under **Show** - check it.

► 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:

```
//
//  ViewController.swift
//  BullsEye
//
//  Created by <you> on <date>.
//  Copyright © <year> <your organization>. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a
        nib.
    }

    override func didReceiveMemoryWarning() {
```

```
super.didReceiveMemoryWarning()  
// Dispose of any resources that can be recreated.  
}  
  
@IBAction func showAlert() {  
}  
}
```

How do you like your first taste of Swift? Before I can tell you what this all means, I have to introduce the concept of a view controller.

Xcode will autosave

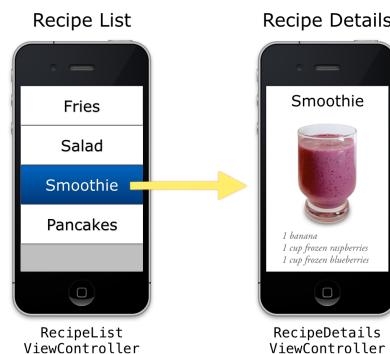
You don't have to save your files after you make changes to them because Xcode will automatically save any modified files when you press the Run button. Nevertheless, Xcode isn't the most stable piece of software out there and occasionally it may crash on you before it has had a chance to save your changes, so I still like to press **⌘+S** on a regular basis to save my files.

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 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 *Bull's Eye* 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.

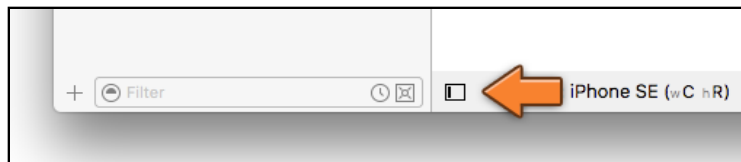
Because you used the Single View Application template, Xcode automatically created the view controller for you. Later you will add a second screen to the game and you will create your own view controller for that.

Make connections

The line of source code you have just added to ViewController.swift lets Interface Builder know that the controller has a “showAlert” action, which presumably will show an alert popup. 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, 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.

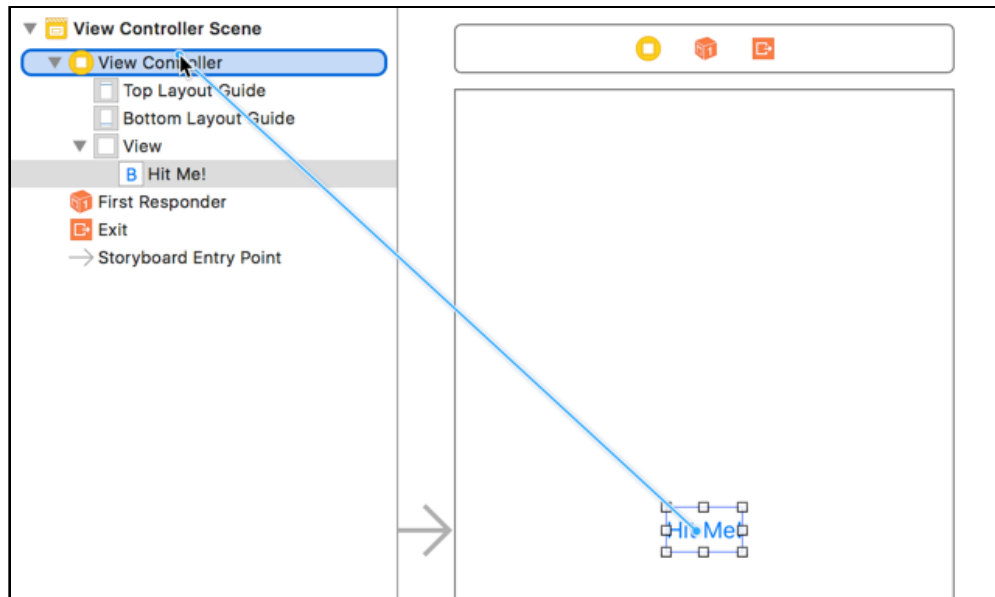


The button that shows the Document Outline pane

► Click the **Hit Me button** once to select it.

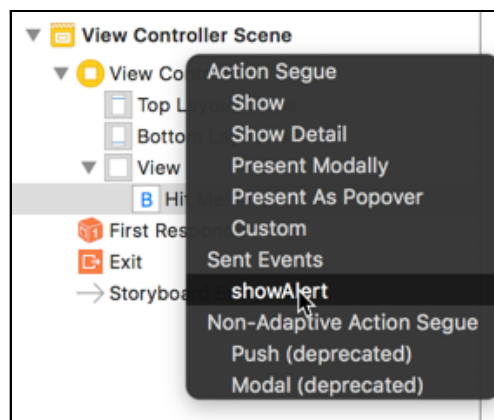
With the Hit Me button selected, hold down the **Control** key, click on the button and drag up to the **View Controller** item in the Document Outline. You should see a blue line going from the button up to View Controller.

(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 two sections, "Action Segue" and "Sent Events", 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 and **showAlert** is the name of the action that you added earlier in the source code of **ViewController.swift**.



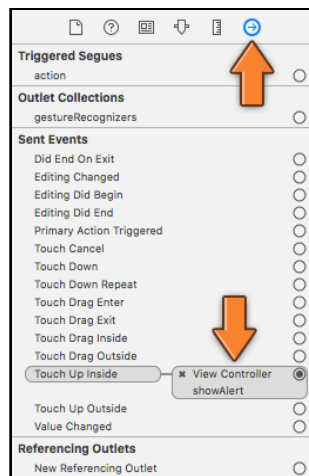
The popup menu with the showAlert action

► 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.

► Click the small arrow-shaped button at the top of the pane to switch to the Connections inspector:

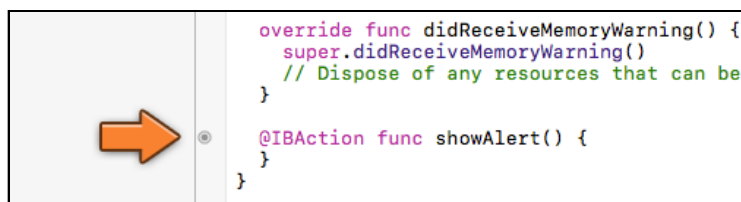


The inspector shows the connections from the button to any other objects

In the Sent Events section, the “Touch Up Inside” event is now connected to the showAlert action. You can 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.



A solid circle means the action is connected to something

Act 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)
}
```

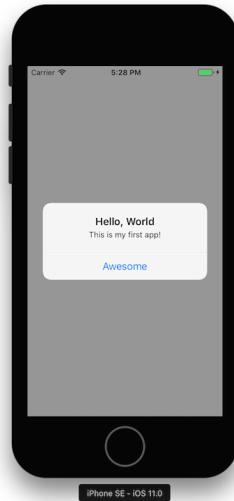
The new lines of code implement the actual alert display functionality.

The commands between the `{ }` brackets tell the iPhone what to do, and they are performed from top to bottom.

The code in `showAlert` creates an alert with a title “Hello, World”, a message “This is my first app!” and a single button labeled “Awesome”.

If you’re not sure about the distinction between the title and the message: both show text, but the title is slightly bigger and in a bold typeface.

► Click the **Run** button from Xcode’s toolbar. If you didn’t make any typos, your app should launch in the Simulator and you should see the alert box when you tap the button.



The alert popup in action

Congratulations, you’ve just written your first iOS app! What you just did may have been mostly gibberish to you, but that shouldn’t matter. We 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, I have provided the complete Xcode projects which are snapshots of the project as at the beginning and end of each chapter. That way you can compare your version of the app to mine, or – if you really make a mess of things – continue from a version that is known to work.

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

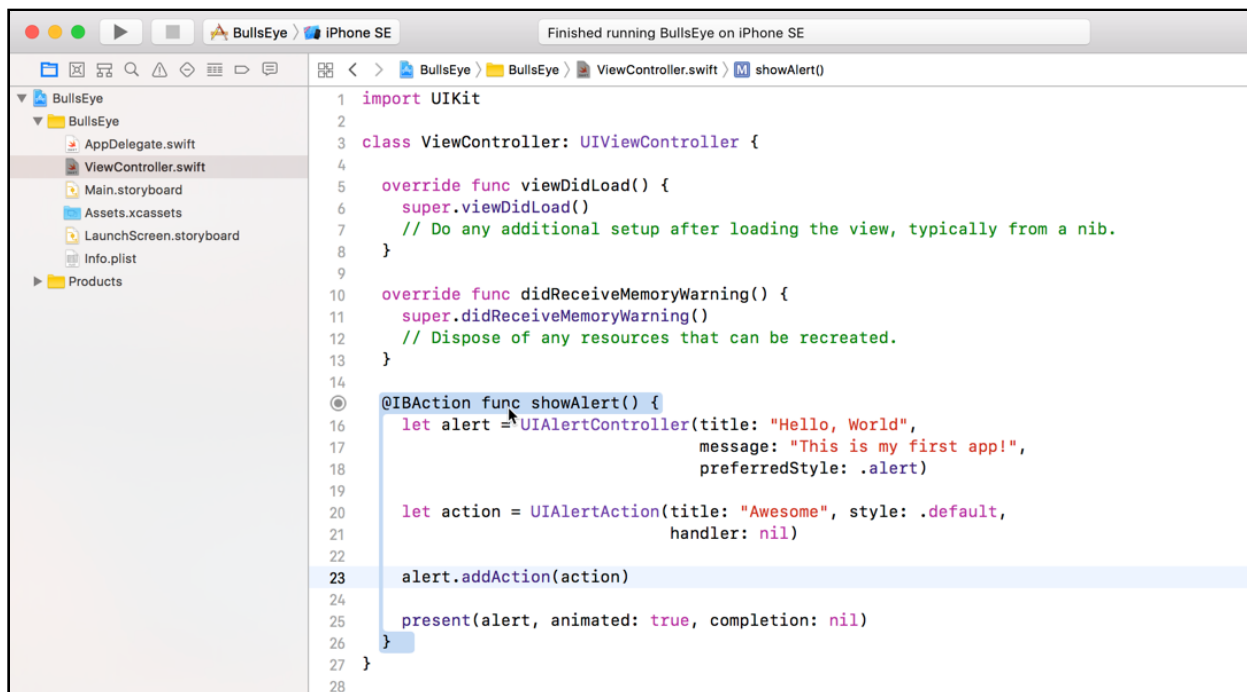
Problems?

If Xcode gives you a “Build Failed” error message after you press Run, then make sure you typed in everything correctly. 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 produce several errors elsewhere in that file.

Typical mistakes are 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 curly bracket `}` or parenthesis `)` somewhere. Not matching up opening and closing brackets is a common error.

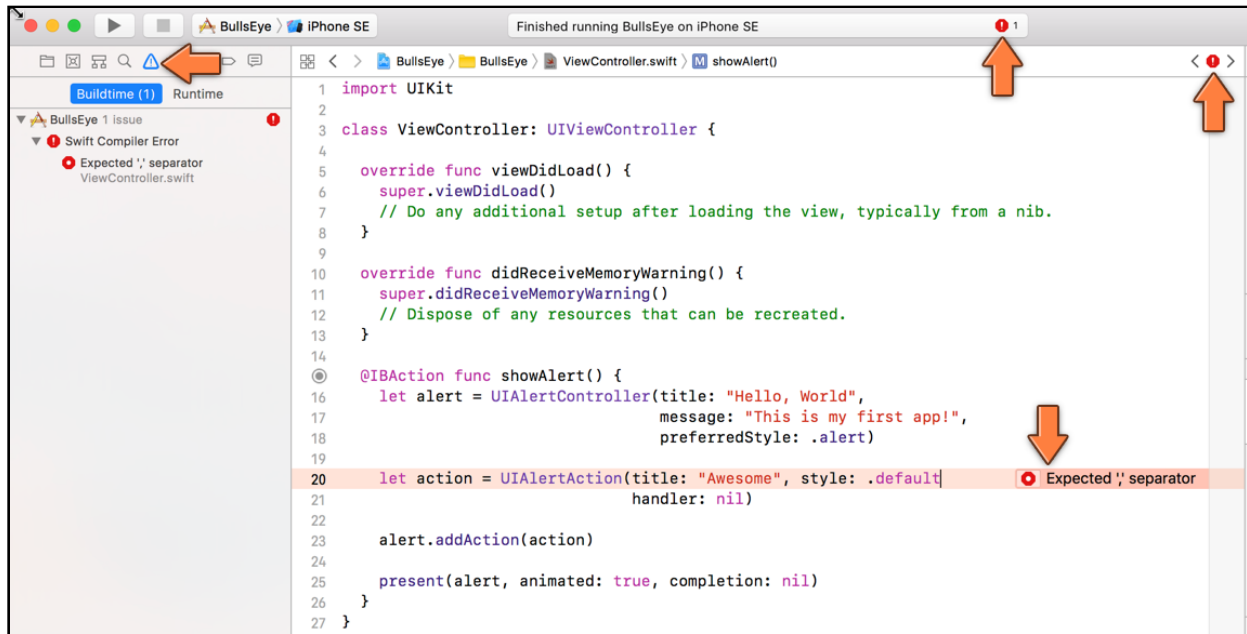
Tip: In Xcode, there are multiple ways to find matching brackets to see if they line up. If you move the text cursor over a closing bracket, Xcode will highlight the corresponding opening bracket, or vice versa. You could also hold down the `⌘` key and move your mouse cursor over a line with a curly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



Xcode shows you the complete block for curly brackets

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 files list using the small buttons at the top.)

In the above screenshot, apparently, I forgot a comma somewhere.

Click on the error message in the Issue navigator and Xcode takes you to the line in the source code with the error. Sometimes, it even suggests a fix to resolve it:



Fix-it suggests a solution to the problem

Sometimes it's a bit of a puzzle to figure out what exactly you did wrong when your build fails - fortunately, Xcode lends a helping hand.

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 till the error is fixed. Warnings are informative. Xcode just says, "You probably didn't mean to do this, but go ahead anyway."

In my opinion, 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 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 – a `UIButton` object – and the alert popup – 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. When the user taps the Hit Me button in the app, for example, 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.

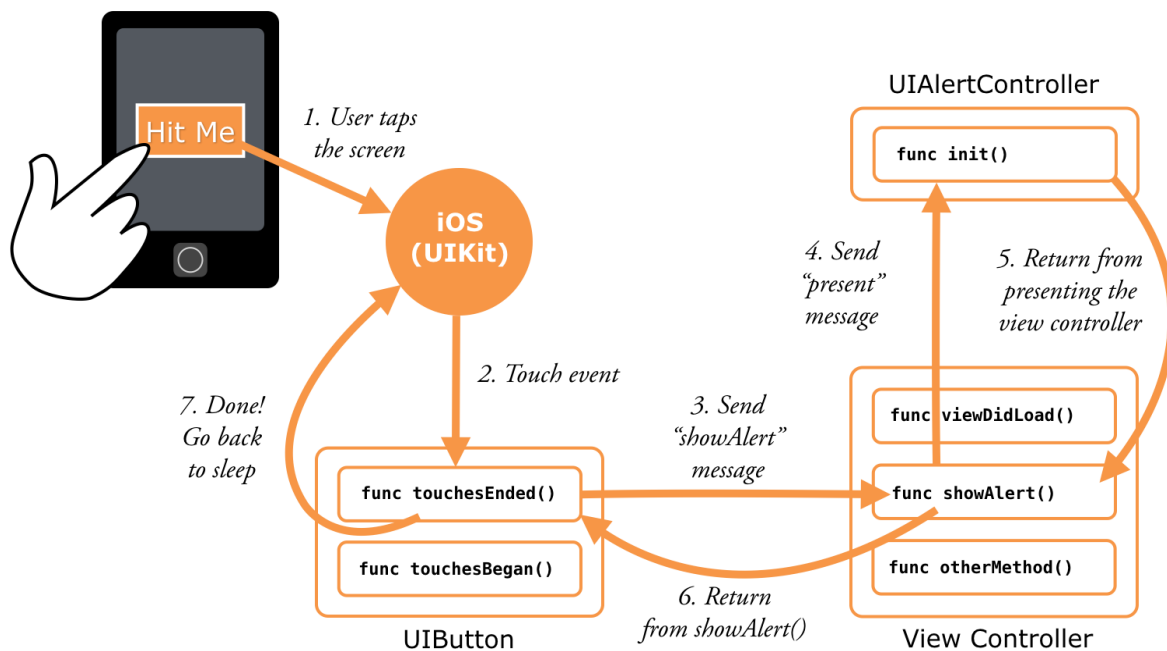
As strange as it may sound, an app spends most of its time doing... absolutely nothing. It just sits there waiting for something to happen. When the user taps the screen, the app springs to action for a few milliseconds and then it goes back to sleep again until the next event arrives.

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.



The general flow of events in an app

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, and many more.

Everything your app does is triggered by some event.

You can find the project files for the app up to this point under **01 - The One-Button App** in the Source Code folder.

Chapter 2: Slider and Labels

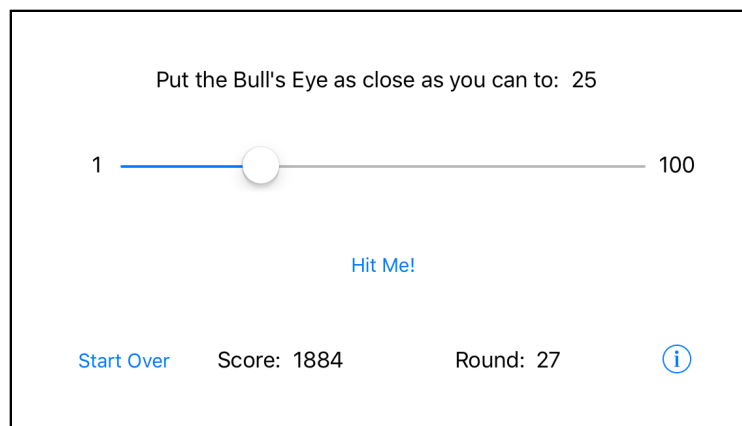
By Fahim Farook and Matthijs Hollemans

Now that you have 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.

So let's 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

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 UIKit controls. This is what they look like straight out of the box.

You’ve probably seen this look before because it is perfectly suitable for regular apps. But because the default look is a little boring for a game, 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.
- **Objects, data and methods:** A quick primer on the basics of object oriented programming.
- **Add the other controls:** Add the rest of the controls necessary to complete the user interface of your app.

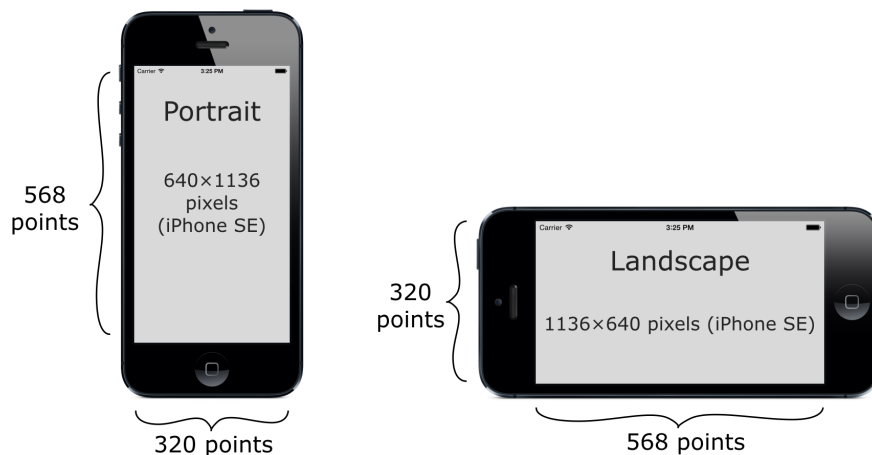
Portrait vs. landscape

Notice that in the previous screenshot, the dimensions of the app have changed: the iPhone is tilted on its side and the screen is wider but less tall. This is called *landscape* orientation.

You’ve no doubt seen landscape apps before on the iPhone. It’s a common display orientation for games but many other types of apps work in landscape mode too, usually in addition to the regular “upright” *portrait* orientation.

For instance, many people prefer to write emails with their device flipped over because the wider screen allows for a bigger keyboard and easier typing.

In portrait orientation, the iPhone SE screen consists of 320 points horizontally and 568 points vertically. For landscape these dimensions are switched.



Screen dimensions for portrait and landscape orientation

So what is a *point*?

On older devices – up to the iPhone 3GS and corresponding iPod touch models, as well as the first iPads – one point corresponds to one pixel. As a result, these low-resolution devices don't look very sharp because of their big, chunky pixels.

I'm sure you know what a pixel is? In case you don't, it's the smallest element that a screen is made up of. (That's how the word originated, a shortened form of pictures, PICS or PIX + ELeMENT = PIXEL.) The display of your iPhone is a big matrix of pixels that each can have their own color, just like a TV screen. Changing the color values of these pixels produces a visible image on the display. The more pixels, the better the image looks.

On the high-resolution Retina display of the iPhone 4 and later models, one point actually corresponds to two pixels horizontally and vertically, so four pixels in total. It packs a lot of pixels in a very small space, making for a much sharper display, which accounts for the popularity of Retina devices.

On the Plus devices it's even crazier: they have a 3x resolution with *nine* pixels for every point. Insane! You need to be eagle-eyed to make out the individual pixels on these fancy Retina HD displays. It becomes almost impossible to make out where one pixel ends and the next one begins, that's how miniscule they are!

It's not only the number of pixels that differs between the various iPhone and iPad models. Over the years they have received different form factors, from the small 3.5-inch screen in the beginning all the way up to 12.9-inches on the iPad Pro model.

The form factor of the device determines the width and height of the screen in points:

Device	Form factor	Screen dimension in points
iPhone 4s and older	3.5"	320 x 480
iPhone 5, 5c, 5s, SE	4"	320 x 568
iPhone 6, 6s, 7, 8	4.7"	375 x 667
iPhone 6, 6s, 7, 8 Plus	5.5"	414 x 736
iPhone X	5.8"	375 x 812
iPad, iPad mini	9.7" and 7.9"	768 x 1024
iPad Pro	10.5"	834 x 1112
iPad Pro	12.9"	1024 x 1366

In the early days of iOS, there was only one screen size. But those days of “one size fits all” are long gone. Now we have a variety of screen sizes to deal with.

UIKit and other frameworks

iOS offers a lot of building blocks in the form of frameworks or “kits”. The UIKit framework provides the user interface controls such as buttons, labels and navigation bars. It manages the view controllers and generally takes care of anything else that deals with your app’s user interface. (That is what UI stands for: User Interface.)

If you had to write all that stuff from scratch, you’d be busy for a long while. Instead, you can build your app on top of the system-provided frameworks and take advantage of all the work the Apple engineers have already put in.

Any object you see whose name starts with UI, such as UIButton, comes from UIKit. When you’re writing iOS apps, UIKit is the framework you’ll spend most of your time with, but there are others as well.

Examples of other frameworks are Foundation, which provides many of the basic building blocks for building apps; Core Graphics for drawing basic shapes such as lines, gradients and images on the screen; AVFoundation for playing sound and video; and many others.

The complete set of frameworks for iOS is known collectively as Cocoa Touch.

Remember that UIKit works with points instead of pixels, so you only have to worry about the differences between the screen sizes measured in points. The actual number of pixels is only important for graphic designers because images are still measured in pixels.

Developers work in points, designers work in pixels.

The difference between points and pixels can be a little confusing, but if that is the only thing you’re confused about right now then I’m doing a pretty good job. ;-)

For the time being, you’ll work with just the iPhone SE screen size of 320×568 points – just to keep things simple. Later on you’ll also make the game fit on the other iPhone screens.

Convert 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.

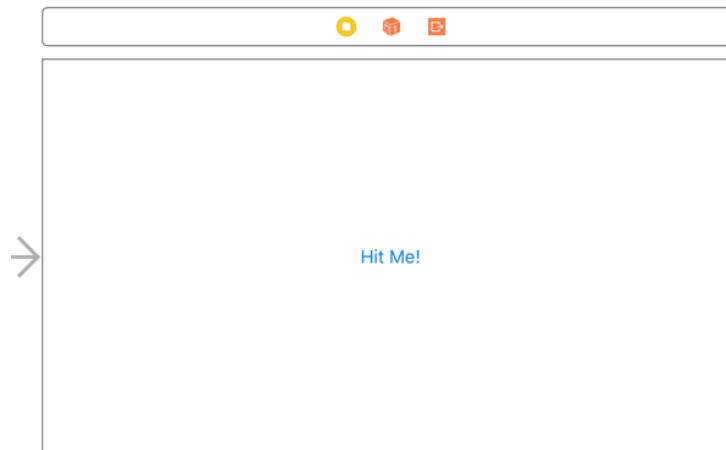
► Open **Main.storyboard**. In Interface Builder, in the **View as: iPhone SE** panel, change **Orientation** to landscape:



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 the iPhone SE 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 the 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.

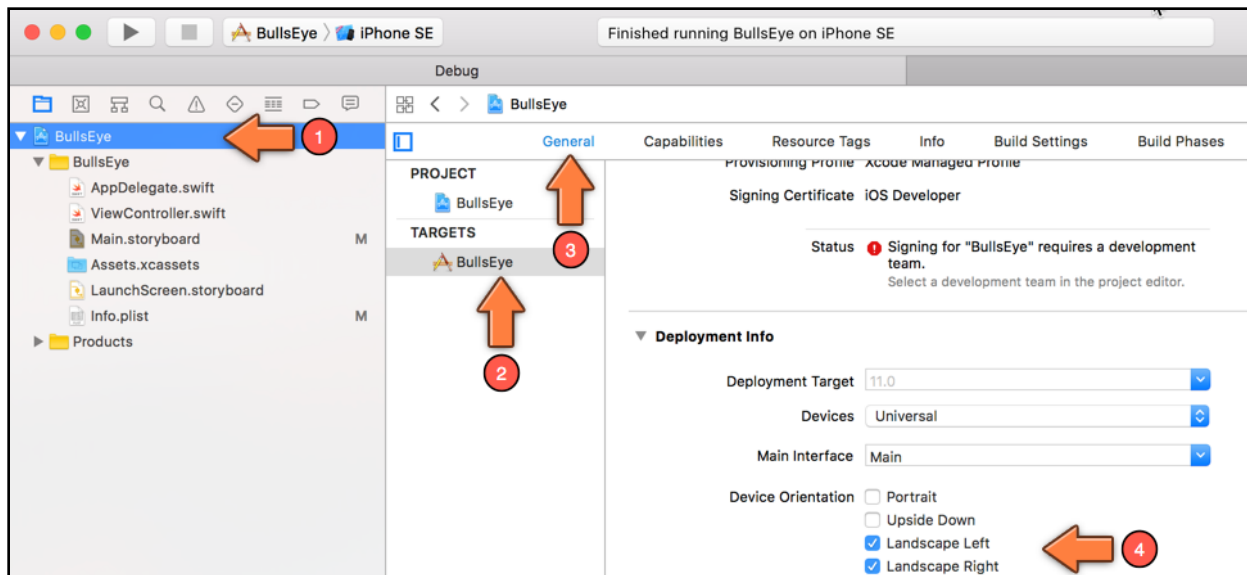
Notice that in landscape orientation the app no longer shows the iPhone's status bar.

This gives apps more room for their user interfaces.

To finalize the orientation switch, you should do one more thing. There is a configuration option that tells iOS what orientations your app supports. New apps that you make from a template always support both portrait and landscape orientations.

► 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.

Objects, data and methods

Time for some programming theory. Yes, you cannot 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 popup. 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 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() {
```

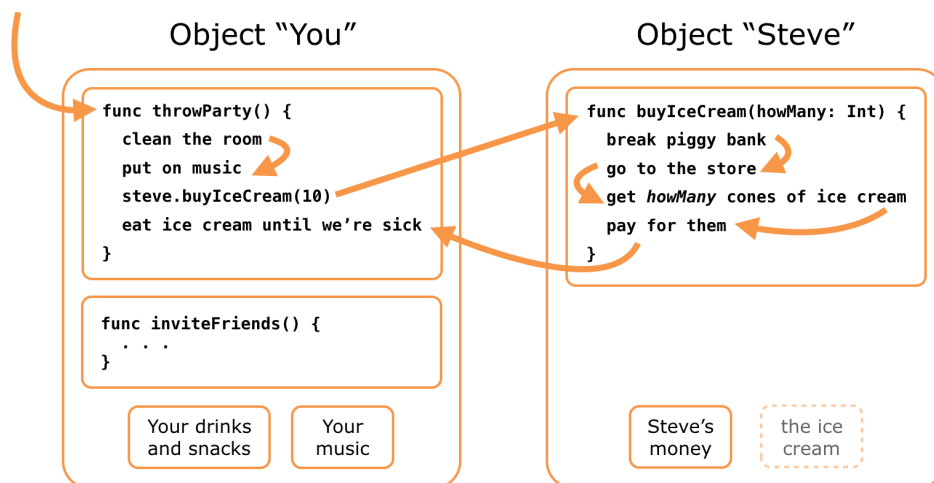


All method definitions start with the word func and have parentheses

If you look through the rest of **ViewController.swift** you’ll see several other methods, such as viewDidLoad() and didReceiveMemoryWarning().

These currently don’t do much; the Xcode template placed them there for your convenience. These specific methods are often used by view controllers, so it’s likely that you will need to fill them in at some point.

The concept of methods may still feel a little weird, so here’s an example:



Every party needs ice cream!

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.

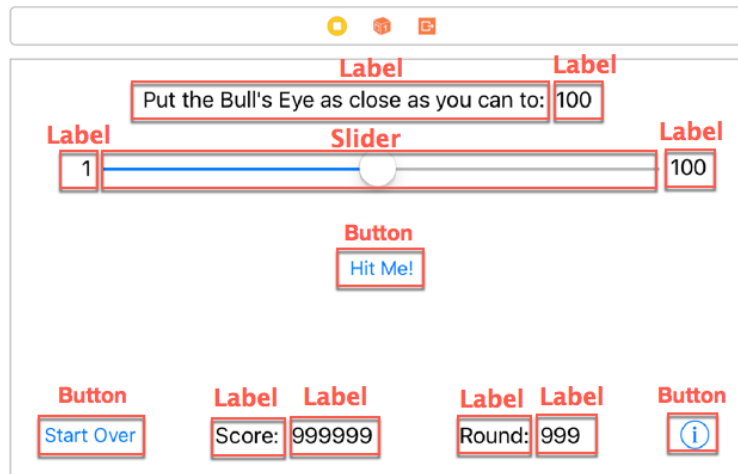
Often the terms “calling a method” or “invoking a method” are used instead. That means the exact same thing as sending a message: the computer jumps to the method you’re calling and returns to where it left off when that method is done.

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.)

Add 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:

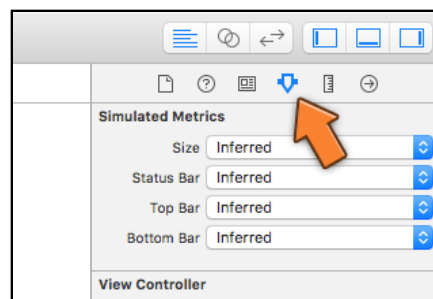


The different views in the game screen

As you can see, I put placeholder values into 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 on to 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.

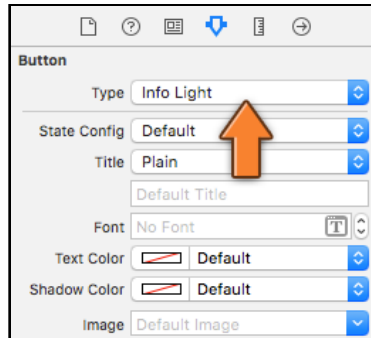
To tweak the settings of these views, you use the **Attributes inspector**. You can find this inspector in the right-hand pane of the Xcode window:



The Attributes inspector

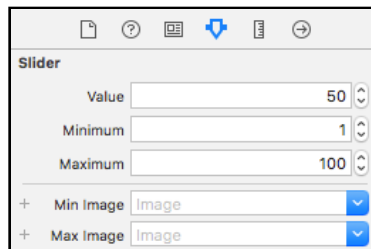
The inspector area shows various aspects of the item that is currently selected. The Attributes inspector, for example, lets you change the background color of a label or the size of the text on a button. You've already seen the Connections inspector that showed the button's actions. As you become more 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 any programming! That is going to change really soon, because you will have to write Swift code to actually make the controls do anything.

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 popup. (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.

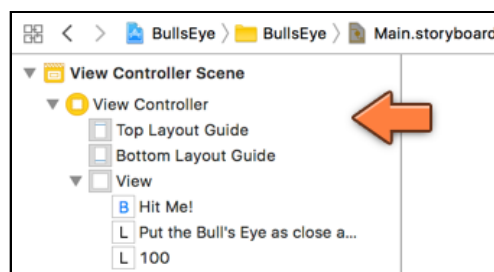
The steps for adding this action are largely the same as before.

► 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)")  
}
```

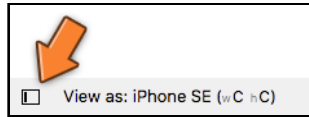
► Second, go to the storyboard and Control-drag from the slider to View Controller in the Document Outline. Let go of the mouse button and select **sliderMoved:** from the popup. Done!

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:

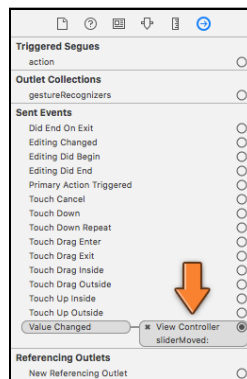


This button shows or hides the Document Outline

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.

You can verify that the connection was made by selecting the slider and looking at the **Connections inspector**:

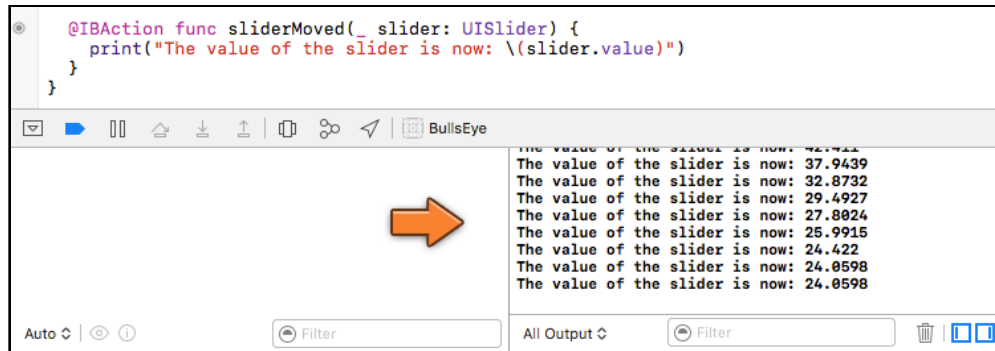


The slider is now hooked up to the view controller

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 opens a new pane at the bottom, the **Debug area**, showing a list of messages:



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 help 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.

I often use `print()` to make sure my apps are doing the right thing before I 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 the iOS Simulator. You can safely ignore these messages.

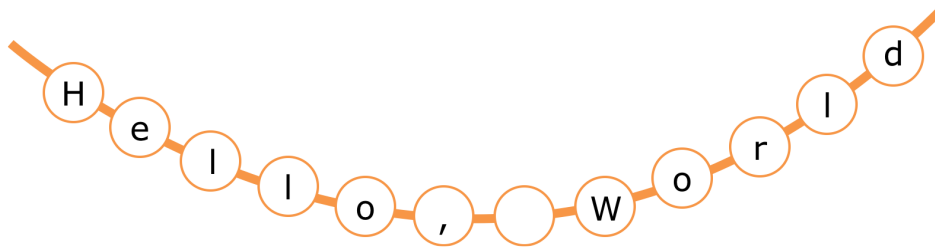
Strings

To put text in your app, you use something called a “string”. The strings you have used so far are:

```
"Hello, World"  
"This is my first app!"  
"Awesome"  
"The value of the slider is now: \(slider.value)"
```

The first three were used to make the `UIAlertController`; the last one you used with `print()`.

Such a chunk of text is called a string because you can visualize the text as a sequence of characters, as if they were pearls in a necklace:



A string of characters

Working with strings is something you need to do all the time when you’re writing apps, so over the course of this book you’ll get quite experienced in using strings.

In Swift, to create a string, simply put the text in between double quotes. In other languages you can often use single quotes as well, but in Swift they must be 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"  
  
// These are wrong:  
'I should have double quotes'  
"Two single quotes do not make a double quote"  
"My quotes are too fancy"  
@"I am an Objective-C string"
```

Anything between the characters `\(` and `)` inside a string is special. The `print()` statement used the string, "The value of the slider is now: \(slider.value)". Think of the `\(...)` as a placeholder: "The value of the slider is now: X", where X will be replaced by the value of the slider.

Filling in the blanks this way is a very common way to build strings in Swift.

Variables

Printing information with `print()` to the Console is very useful during development of the app, but it's absolutely useless to the user because they can't see the Console when the app is running on a device.

Let's improve this to show the value of the slider in the alert popup. So how do you get the slider's value into `showAlert()`?

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.

Fortunately, Swift has a building block for exactly this purpose: the *variable*.

► Open **ViewController.swift** and add the following at the top, directly below the line that says `class ViewController`:

```
var currentValue: Int = 0
```

You have now added a variable named `currentValue` to the view controller object.

The code should look like this (I left out the method code, also known as the method implementations):

```
import UIKit

class ViewController: UIViewController {
    var currentValue: Int = 0

    override func viewDidLoad() {
        . . .
    }

    override func didReceiveMemoryWarning() {
        . . .
    }

    @IBAction func showAlert() {
        . . .
    }

    @IBAction func sliderMoved(_ slider: UISlider) {
        . . .
    }
}
```

It is customary to add the variables above the methods, and to indent everything with a tab, or two to four spaces. Which one you use is largely a matter of personal preference. I like to use two spaces. (You can configure this in Xcode's preferences panel. From the menu bar choose **Xcode** → **Preferences...** → **Text Editing** and go to the **Indentation** tab.)

Remember when I said that a view controller, or any object really, could have both data and functionality? The `showAlert()` and `sliderMoved()` actions are examples of functionality, while the `currentValue` variable is part of the view controller's data.

A variable allows the app to remember things. Think of a variable as a temporary storage container for a single piece of data. Similar to how there are containers of all sorts and sizes, data comes in all kinds of shapes and sizes.

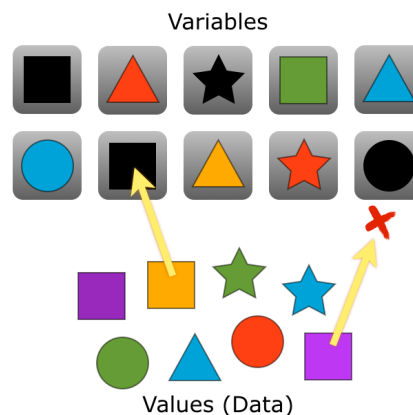
You don't just put stuff in the container and then forget about it. You will often replace its contents with a new value. When the thing that your app needs to remember changes, you take the old value out of the box and put in the new value.

That's the whole point behind variables: they can *vary*. For example, you will update `currentValue` with the new position of the slider every time the slider is moved.

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 type `Int` for the `currentValue` variable, which means this container can hold whole numbers (also known as “integers”) between at least minus two billion and plus two billion. `Int` is one of the most common data types. There are many others though, and you can even make your own.

Variables are like children's toy blocks:



Variables are containers that hold values

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 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.

I said a variable is a *temporary* storage container. How long will it 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 a new value into that variable or until you destroy the container altogether.

Each variable has a certain lifetime (also known as its *scope*) that depends on exactly where in your program you defined that variable. In this case, `currentValue` sticks around for just as long as its owner, `ViewController`, does. Their fates are intertwined.

The view controller, and thus `currentValue`, is there for the duration of the app. They don’t get destroyed until the app quits. Soon you’ll also see variables that are short-lived (also known as “local” variables).

Enough theory, let’s make this variable work for us.

► 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)
```

What is going on here?

You’ve seen `slider.value` before, which is the slider’s position at a given moment. This is a value between 1 and 100, possibly with digits behind the decimal point. And `currentValue` is the name of the variable you have just created.

To put a new value into a variable, you simply do this:

```
variable = the new value
```

This is known as “assignment”. You *assign* the new value to the variable. It puts the shape into the box. Here, you put the value that represents the slider’s position into the `currentValue` variable.

Functions

But what is the `lroundf` thing? Recall that the slider’s value can be a non-whole number. You’ve seen this with the `print()` output in the Console as you moved the slider.

However, this game would be really hard if you made the player guess the position of the slider with an accuracy that goes beyond whole numbers. That will be nearly impossible to get right!

To give the player a fighting chance, you use whole numbers only. That is why `currentValue` has a data type of `Int`, because it stores *integers*, a fancy term for whole numbers.

You use the function `lroundf()` to round the decimal number to the nearest whole number and you then store that rounded-off number in `currentValue`.

Functions and methods

You’ve already seen that methods provide functionality, but *functions* are another way to put functionality into your apps (the name sort of gives it away, right?). Functions and methods are how Swift programs combine multiple lines of code into single, cohesive units.

The difference between the two is that a function doesn’t belong to an object while a method does. In other words, a method is exactly like a function – that’s why you use the `func` keyword to define them – except that you need to have an object to use the method. But regular functions, or *free functions* as they are sometimes called, can be used anywhere.

Swift provides your programs with a large library of useful functions. The function `lroundf()` is one of them and you’ll be using quite a few others as you progress. `print()` is also a function, by the way. You can tell because the function name is always followed by parentheses that possibly contain one or more parameters.

► 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)
}
```

The line with `let message =` is new. Also note the other two small changes marked by comments.

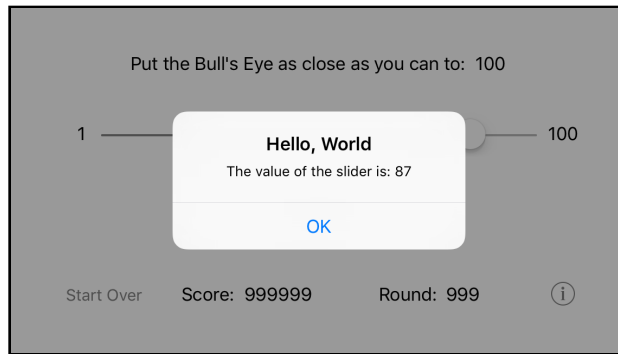
Note: Anything appearing after two slashes `//` (and up to the end of that particular line) in Swift source code is treated as a comment - a note by the developer to themselves, or to other developers. The Swift compiler generally ignores comments - they are there for the convenience of humans.

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 popup. 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's knob 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 **02 - Slider and Labels** in the Source Code folder.

Chapter 3: Outlets

By Fahim Farook and Matthijs Hollemans

You've built the user interface for *Bull's Eye* and you know how to find the current position of the slider. That already knocks quite a few items off the to-do list. This chapter takes care of a few other items from the to-do list and covers the following items:

- **Improve the slider:** Set the initial slider value (in code) to be whatever value 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.

Improve 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.

Get 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,  
    // typically from a nib.  
}
```

When you created this project based on Xcode's template, Xcode inserted the `viewDidLoad()` method into the source code. You will now add some code to it.

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)  
39     }  
}
```

Use of unresolved identifier 'slider'

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 I first introduced variables, I 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 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.

Set up outlets

So, with this newly-gained knowledge of variables and their scope, 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`. I usually put outlets with the other instance

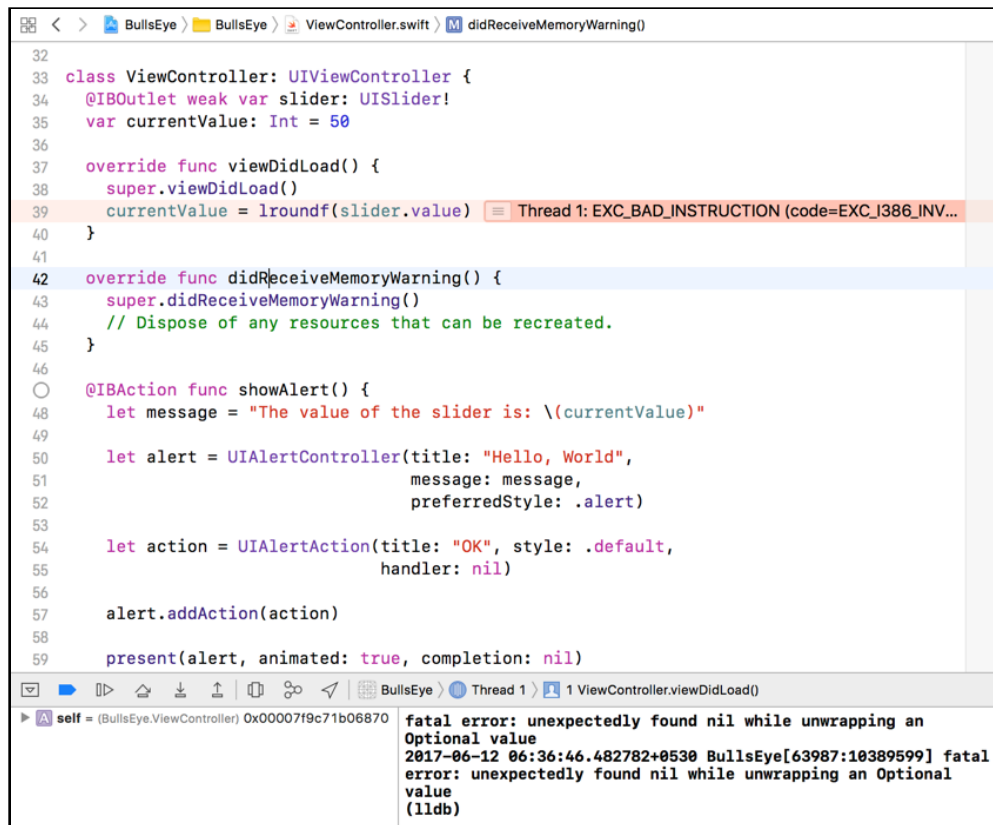
variables - at the top of the class implementation.

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:



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

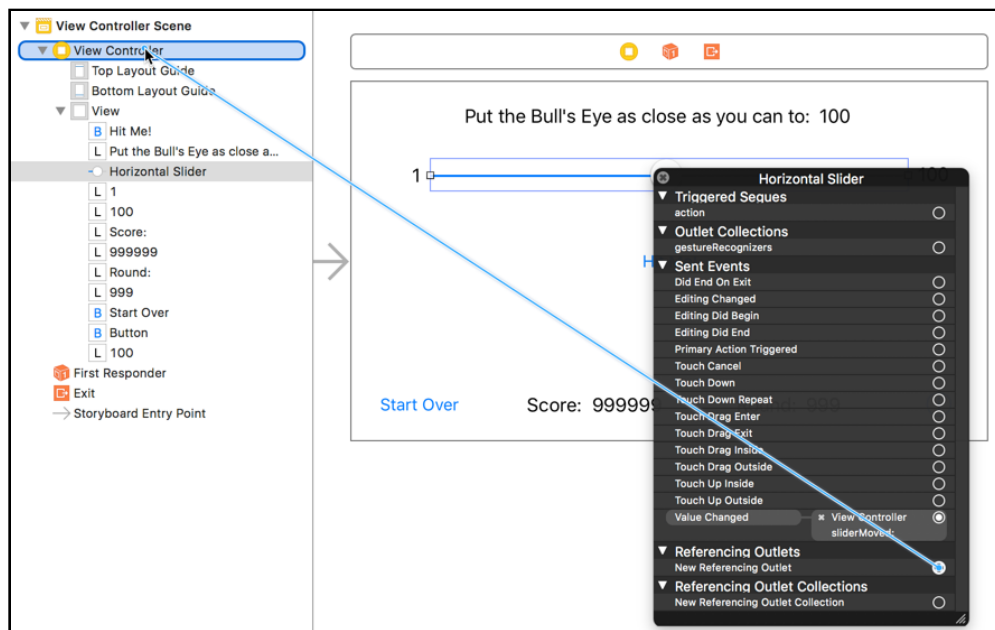
ran and `viewDidLoad()` was called, it tried to find the matching connection in the storyboard and could not - and crashed.

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 popup menu works exactly the same as the Connections inspector. I just wanted to show you this 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 popup 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.

Comments

You’ve seen green text that begin with `//` a few times now. As I explained earlier briefly, these are comments. You can write any text you want after the `//` symbol as the compiler will ignore such lines from the `//` to the end of the line completely.

```
// I am a comment! You can type anything here.
```

Anything between the `/*` and `*/` markers is considered a comment as well. The difference between `//` and `/* */` is that the former only works on a single line, while the latter can span multiple lines.

```
/*  
  I am a comment as well!  
  I can span multiple lines.  
*/
```

The `/* */` comments are often used to temporarily disable whole sections of source code, usually when you’re trying to hunt down a pesky bug, a practice known as “commenting out”. (You can use the **Cmd-`/`** keyboard shortcut to comment/uncomment the currently selected lines, or if you have nothing selected, the current line.)

The best use for comment lines is to explain how your code works. Well-written source code is self-explanatory but sometimes additional clarification is useful. Explain to whom? To yourself, mostly.

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. Use comments to jog your memory.

Generate the random number

You still have quite a ways to go before the game is playable. So, let's get on with the next item on the list: generating a random number and displaying it on the screen.

Random numbers come up a lot when you're making games because games often need to have some element of unpredictability. You can't really get a computer to generate numbers that are truly random and unpredictable, but you can employ a *pseudo-random generator* to spit out numbers that at least appear to be random. You'll use my favorite, `arc4random_uniform()`.

Before you generate the random value though, you need a place to store it.

► Add a new variable at the top of **ViewController.swift**, with the other variables:

```
var targetValue: Int = 0
```

If you don't tell the compiler what kind of variable `targetValue` is, then it doesn't know how much storage space to allocate for it, nor can it check if you're using the variable properly everywhere.

Variables in Swift must always have a value, so here you give it the initial value 0. That 0 is never used in the game; it will always be overwritten by the random value you'll generate at the start of the game.

I hope the reason is 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 what the user selected.

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 = 1 + Int(arc4random_uniform(100))
```

The complete `viewDidLoad()` should now look like this:

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

```
currentValue = lroundf(slider.value)
targetValue = 1 + Int(arc4random_uniform(100))
}
```

What did you do here? You call the `arc4random_uniform()` function to get an arbitrary integer (whole number) between 0 and 99.

Why is the highest value 99 when the code says 100, you ask? That is because `arc4random_uniform()` treats the upper limit as exclusive. It only goes up-to 100, not up-to-and-including. To get a number that is truly in the range 1 - 100, you add 1 to the result of `arc4random_uniform()`.

Display 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)"

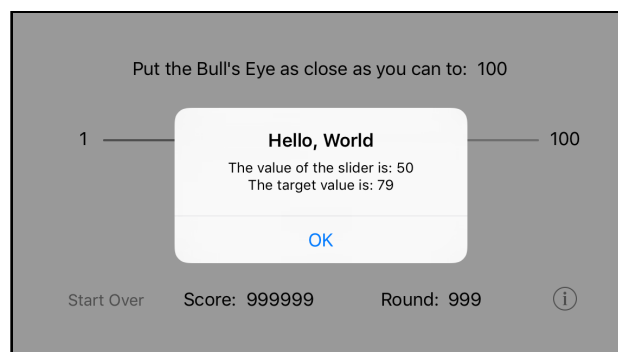
    let alert = . . .
}
```

Tip: Whenever you see `. . .` in a source code listing I mean that as shorthand for: this part didn't change. Don't go replacing the existing code with an 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.

► Run the app and try it out!



The alert shows the target value on a new line

Note: Earlier you’ve 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 :]

Add 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.

Start 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 = 1 + Int(arc4random_uniform(100))  
    currentValue = 50  
}
```

```
    slider.value = Float(currentValue)
}
```

It doesn't really matter where you put it, 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.

Use 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()
}
```

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. The iPhone will then go 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.

Different ways to call methods

Sometimes you may see method calls written like this:

```
self.startNewRound()
```

That does the exact same thing as just `startNewRound()` without `self.` in front. Recall how I just said 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 most cases.

To be fair, 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 programs, 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

I hope you can see the advantage of putting 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 = 1 + Int(arc4random_uniform(100))  
    currentValue = 50  
    slider.value = Float(currentValue)
```

```
}  
  
@IBAction func showAlert() {  
    . . .  
  
    targetValue = 1 + Int(arc4random_uniform(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 would have to duplicate will 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 this change in two places as well.

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.

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 = 1 + Int(arc4random_uniform(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 and look inside.

Well-written source code speaks for itself. I hope I have convinced you of the value of making new methods!

► 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 I thought 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(...)` and `Int(...)` do in these lines:

```
targetValue = 1 + Int(arc4random_uniform(100))
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 puts it into a new `Float` value that it can pass on to the `UISlider`.

Something similar happens with `arc4random_uniform()`, where the random number gets converted to an `Int` first before it can be stored in `targetValue`.

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, as you've done here.

Display 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...

Set up the storyboard

When you made the storyboard, you already 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

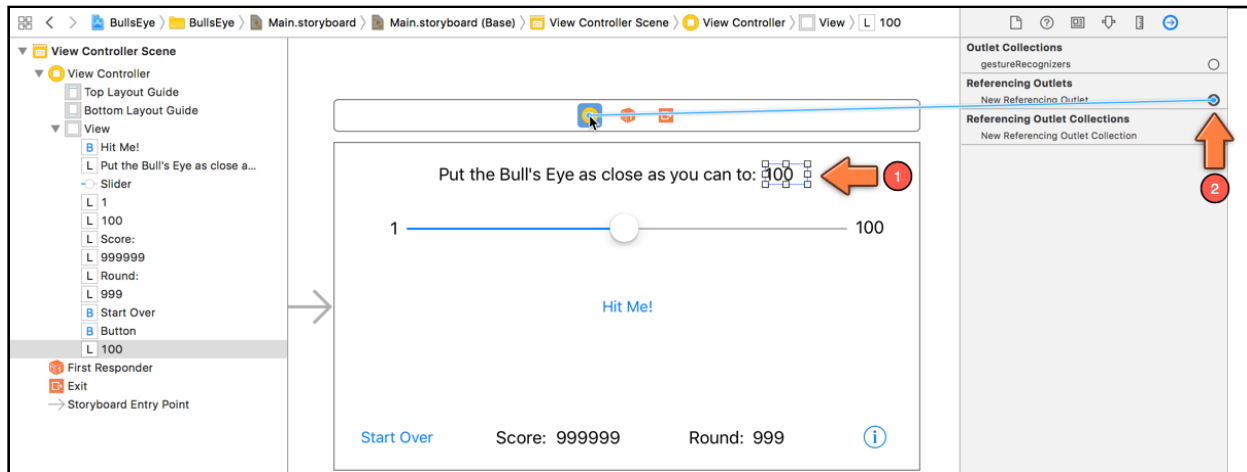
This will be very similar to what you did with the slider. Recall that you added an `@IBOutlet` variable so you could reference the slider anywhere from within the view controller. Using this outlet variable you could ask the slider for its value, through `slider.value`. You'll do the same thing for the label.

► In **ViewController.swift**, add the following line below the other outlet:

```
@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.)



Connecting the target value label to its outlet

- Select **targetLabel** from the popup, and the connection is made.

Display the target value via code

- Now on to the good stuff. Add the following method below `startNewRound()` in **ViewController.swift**:

```
func updateLabels() {
    targetLabel.text = String(targetValue)
}
```

You're putting this logic into its own 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 kinds of things.

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(...)` and `Int(...)`.

Just in case you were wondering, you could also convert `targetValue` to a `String` by using it as a string with a placeholder 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` anywhere.)

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` specifier. This 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.

Call the method

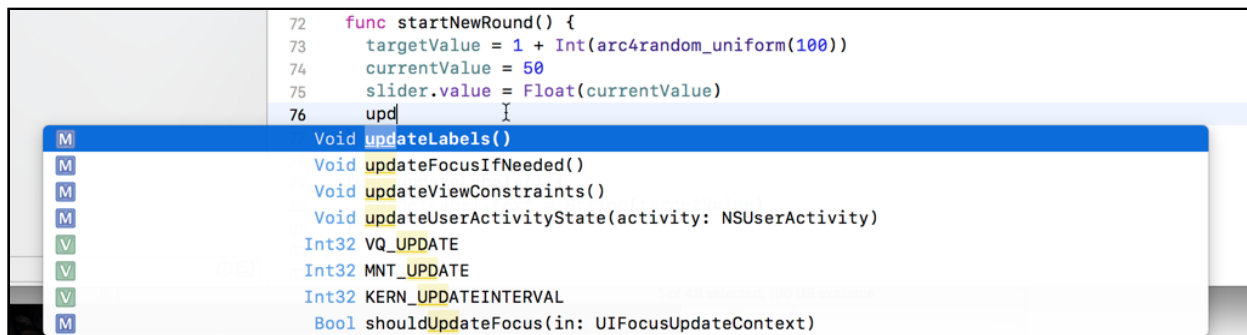
The logical place to call `updateLabels()` would be after each call to `startNewRound()`, because that is where you calculate the new target value. So, you could always add a call to `updateLabels()` in `viewDidLoad()` and `showAlert()`, but there's another way too!

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 = 1 + Int(arc4random_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
    updateLabels() // Add this line
}
```

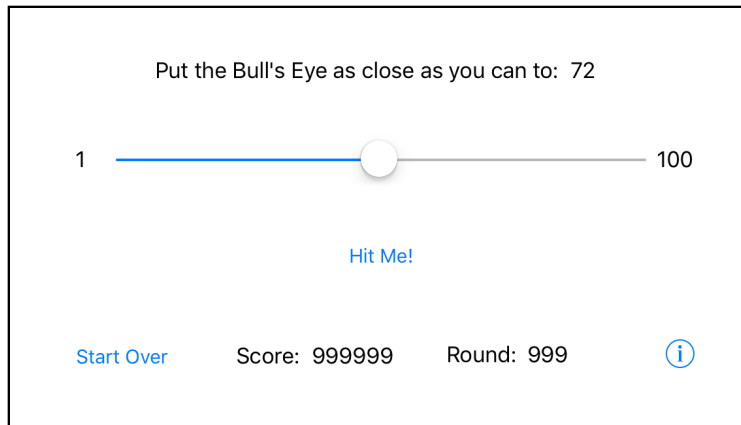
You should be able to type just the first few letters of the method name, like **upd**, and Xcode will show you a list of suggestions matching what you typed. Press **Enter** (or **Tab**) to accept the suggestion (if you are on the right item - or scroll the list to find the right item and then press 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 for from the beginning - Xcode uses fuzzy search and typing "date" 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

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

Chapter 4: Rounds and Score

By Fahim Farook and Matthijs Hollemans

OK, so you have made quite a bit of progress on the game and the to-do list is getting ever shorter :) So what's next on the list now that you can generate a random number and display it on screen?

A quick look at the task list shows that you now have to "compare the value of the slider to that random number and calculate a score based on how far off the player is". Let's get to it!

This chapter covers the following:

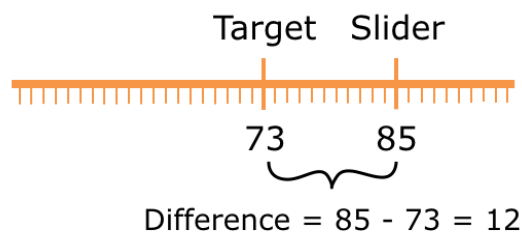
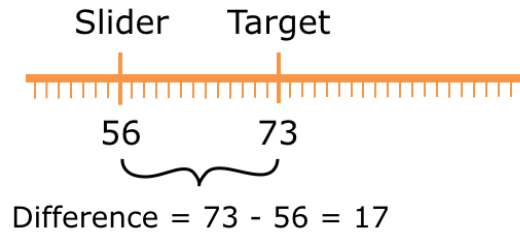
- **Get the difference:** Calculate the difference between the target value and the value that the user selected.
- **Other ways to calculate the difference:** Other approaches to calculating the difference.
- **What's the score?:** Calculate the user's score based on the difference value.
- **The total score:** Calculate the player's total score over multiple rounds.
- **Display the score:** Display the player score on screen.
- **One more round...:** Implement updating the round count and displaying the current round on screen.

Get the difference

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.

To calculate the score for each round, you look at how far off 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 `currentValue` from `targetValue`.

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

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 – `currentValue` minus `targetValue` – 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 we're in trouble here...

Exercise: How would you frame the solution to this problem if I asked you to solve it in natural language? Don't worry about how to express it in computer language for now, just think it through in plain English.

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.*

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:

If the slider is at position 60 and the target value is 40, then onscreen the slider is to the right of the target value, and 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 here is $30 - 10 =$ also 20.

Algorithms

What you've just done is come up with an *algorithm*, which is a fancy term for a series of steps for solving a computational problem. This is only a very simple algorithm, but it is one nonetheless.

There are many famous algorithms, such as *quicksort* for sorting a list of items and *binary search* for quickly searching through such a sorted list. Other people have already invented many algorithms that you can use in your own programs - that'll save you a lot of thinking!

However, in the programs that you write, you'll probably have to come up with a few algorithms of your own at some time or other. Some are simple such as the one above; others can be pretty hard and might cause you to throw up your hands in despair. But that's part of the fun of programming :]

The academic field of Computer Science concerns itself largely with studying algorithms and finding better ones.

You can describe any algorithm in plain English. It's just a series of steps that you perform to calculate something. Often, you can perform that calculation in your head or on paper, the way you did above. But for more complicated algorithms doing that might take you forever, so at some point you'll have to convert the algorithm to computer code.

The point I'm trying to make is this: if you ever get stuck and you don't know how to make your program calculate something, take a piece of paper and try to write out the steps in English. Set aside the computer for a moment and think the steps through. How you would perform this calculation by hand?

Once you know how to do that, converting the algorithm to code should be a piece of cake.

The difference algorithm

Getting back to your ccode, it is possible you came up with a different way to solve this little problem, and I'll show you two alternatives in a minute, but let's convert this one to computer code first:

```
var difference: Int
if currentValue > targetValue {
    difference = currentValue - targetValue
} else if targetValue > currentValue {
    difference = targetValue - currentValue
} else {
    difference = 0
}
```

The if construct is new. It allows your code to make decisions and it works much like you would expect from English. Generally, it works like this:

```
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, for example currentValue is greater than targetValue, then the code in the block between the { } brackets is executed.

However, if the condition is not true, then the computer looks at the else if condition and evaluates that. There may be more than one else if, and it tries them one by one from top to bottom until one proves to be true.

If none of the conditions are found to be valid, then the code in the else block is executed.

In the implementation of this little algorithm, you first create a local variable named difference to hold the result. This will either be a positive whole number or zero, so an Int will do:

```
var difference: Int
```

Then you compare the `currentValue` against the `targetValue`. First, you determine if `currentValue` is greater than `targetValue`:

```
if currentValue > targetValue {
```

The `>` is the *greater-than* operator. The condition `currentValue > targetValue` is considered true if the value stored in the `currentValue` variable is at least one higher than the value stored in the `targetValue` variable. In that case, the following line of code is executed:

```
difference = currentValue - targetValue
```

Here you subtract `targetValue` (the smaller one) from `currentValue` (the larger one) and store the difference in the `difference` variable.

Notice how I chose variable names that clearly describe what kind of data the variables contain. Often you will see code such as this:

```
a = b - c
```

It is not immediately clear what this is supposed to mean, other than that some arithmetic is taking place. The variable names “a”, “b” and “c” don’t give any clues as to their intended purpose or what kind of data they might contain.

Back to the `if` statement. If `currentValue` is equal to or less than `targetValue`, the condition is untrue (or *false* in computer-speak) and the program will skip the code block and move on to the next condition:

```
} else if targetValue > currentValue {
```

The same thing happens here as before, except that now the roles of `targetValue` and `currentValue` are reversed. The computer will only execute the following line when `targetValue` is the greater of the two values:

```
difference = targetValue - currentValue
```

This time you subtract `currentValue` from `targetValue` and store the result in the `difference` variable.

There is only one situation you haven’t handled yet, and that is when `currentValue` and `targetValue` 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  
}
```

Since at this point 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.

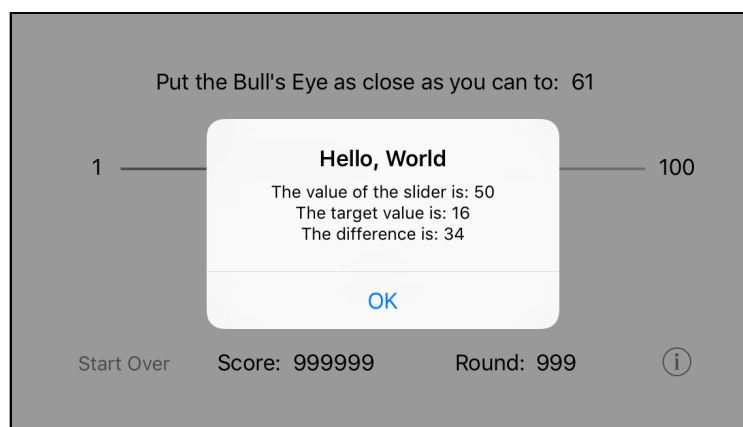
Display the difference

► Let's put this algorithm into action. Add it to the top of `showAlert()`:

```
@IBAction func showAlert() {  
    var difference: Int  
    if currentValue > targetValue {  
        difference = currentValue - targetValue  
    } else if targetValue > currentValue {  
        difference = targetValue - currentValue  
    } else {  
        difference = 0  
    }  
  
    let message = "The value of the slider is: \(currentValue)" +  
        "\nThe target value is: \(targetValue)" +  
        "\nThe difference is: \(difference)"  
    . . .  
}
```

Just so you can see that it works, you add the difference value to the alert message as well.

► Run it and see for yourself.



The alert shows the difference between the target and the slider

Other ways to calculate the difference

I mentioned earlier that there are other ways to calculate the difference between `currentValue` and `targetValue` as a positive number. The above algorithm works well but it is eight lines of code. I think we can come up with a simpler approach that takes up fewer lines.

The new algorithm goes like this:

1. *Subtract the target value from the slider's value.*
2. *If the result is a negative number, then multiply it by -1 to make it a positive number.*

Here you no longer avoid the negative number since computers can work just fine with negative numbers. You simply turn it into a positive number.

Exercise: Convert the above algorithm into source code. Hint: the English description of the algorithm contains the words “if” and “then”, which is a pretty good indication you’ll have to use an `if` statement.

You should have arrived at something like this:

```
var difference = currentValue - targetValue
if difference < 0 {
    difference = difference * -1
}
```

This is a pretty straightforward translation of the new algorithm.

You first subtract the two variables and put the result into the `difference` variable.

Notice that you can create the new variable and assign the result of a calculation to it, all in one line. You don’t need to put it onto two different lines, like so:

```
var difference: Int
difference = currentValue - targetValue
```

Also, in the one-liner version you didn’t have to tell the compiler that `difference` takes `Int` values. Because both `currentValue` and `targetValue` are `Int`s, Swift is smart enough to figure out that `difference` should also be an `Int`.

This feature is called *type inference* and it’s one of the big selling points of Swift.

Once you have the subtraction result, you use an `if` statement to determine whether difference is negative, i.e. less than zero. If it is, you multiply by `-1` and put the new result – now a positive number – back into the `difference` variable.

When you write,

```
difference = difference * -1
```

the computer first multiplies `difference`'s value by `-1`. Then it puts the result of that calculation back into `difference`. In effect, this overwrites `difference`'s old contents (the negative number) with the positive number.

Because this is a common thing to do, there is a handy shortcut:

```
difference *= -1
```

The `*=` operator combines `*` and `=` into a single operation. The end result is the same: the variable's old value is gone and it now contains the result of the multiplication.

You could also have written this algorithm as follows:

```
var difference = currentValue - targetValue
if difference < 0 {
    difference = -difference
}
```

Instead of multiplying by `-1`, you now use the negation operator to ensure `difference`'s value is always positive. This works because negating a negative number makes it positive again. (Ask a math professor if you don't believe me.)

Use the new algorithm

► Give these new algorithms a try. You should replace the old stuff at the top of `showAlert()` as follows:

```
@IBAction func showAlert() {
    var difference = currentValue - targetValue
    if difference < 0 {
        difference = difference * -1
    }

    let message = . . .
}
```

When you run this new version of the app (try it!), it should work exactly the same as before. The result of the computation does not change, only the technique you used changed.

Another variation

The final alternative algorithm I want to show you uses a function.

You’ve already seen functions a few times before: you used `arc4random_uniform()` when you made random numbers and `lroundf()` for rounding off the slider’s decimals.

To make sure a number is always positive, you can use the `abs()` function.

If you took math in school you might remember the term “absolute value”, which is the value of a number without regard to its sign.

That’s exactly what you need here, and the standard library contains a convenient function for it, which allows you to reduce this entire algorithm down to a single line of code:

```
let difference = abs(targetValue - currentValue)
```

It really doesn’t matter whether you subtract `currentValue` from `targetValue` or the other way around. If the number is negative, `abs()` turns it positive. It’s a handy function to remember.

► Make the change to `showAlert()` and try it out:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
  
    let message = . . .  
}
```

It doesn’t get much simpler than that!

Exercise: Something else has changed... can you spot it?

Answer: You wrote **let** `difference` instead of **var** `difference`.

Variables and constants

Swift makes a distinction between variables and *constants*. Unlike a variable, the value of a constant, as the name implies, cannot change.

You can only put something into the box of a constant once and cannot replace it with something else afterwards.

The keyword `var` creates a variable while `let` creates a constant. That means `difference` is now a constant, not a variable.

In the previous algorithms, the value of `difference` could possibly change. If it was negative, you turned it positive. That required `difference` to be a variable, because only variables can have their value change.

Now that you can calculate the whole thing in a single line, `difference` will never have to change once you've given it a value. In that case, it's better to make it a constant with `let`. (Why is that better? It makes your intent clear, which in turn helps the Swift compiler understand your program better.)

By the same token, `message`, `alert`, and `action` are also constants (and have been all along!). Now you know why you declared these objects with `let` instead of `var`. Once they've been given a value, they never need to change.

Constants are very common in Swift. Often, you only need to hold onto a value for a very short time. If in that time the value never has to change, it's best to make it a constant (`let`) and not a variable (`var`).

What's the score?

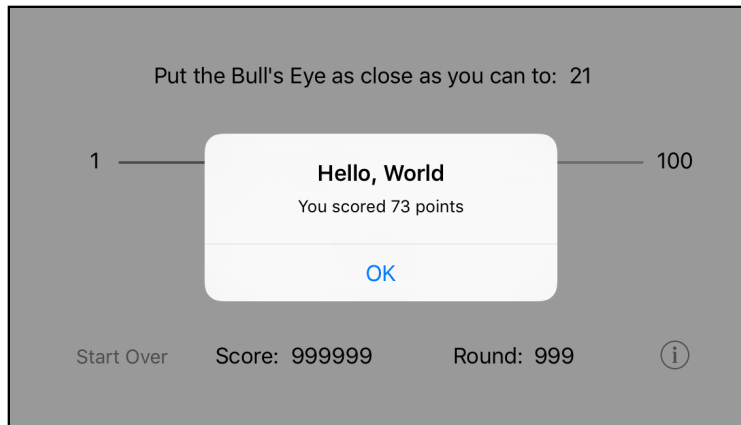
Now that you know how far off the slider is from the target, calculating the player's score for each round is easy.

► Change `showAlert()` to:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    let points = 100 - difference  
  
    let message = "You scored \(points) points"  
    . . .  
}
```

The maximum score you can get is 100 points if you put the slider right on the target and the difference is zero. The further away from the target you are, the fewer points you earn.

► Run the app and score some points!



The alert with the player's score for the current round

Exercise: Because the maximum slider position is 100 and the minimum is 1, the biggest difference is $100 - 1 = 99$. That means the absolute worst score you can have in a round is 1 point. Explain why this is so. (Eek! It requires math!)

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.

Store the total score

Because the game needs to keep the total score around for a long time, you will need an instance variable.

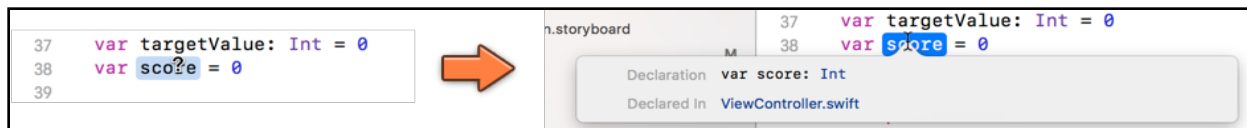
► Add a new score instance variable to **ViewController.swift**:

```
class ViewController: UIViewController {  
    var currentValue: Int = 0  
    var targetValue: Int = 0  
    var score = 0           // add this line  
}
```

Did you notice that? Unlike the other two instance variables, you did not state that score is an Int!

If you don't specify a data type, Swift uses *type inference* to figure out what type you meant. Because 0 is a whole number, Swift assumes that score should be an integer, and therefore automatically gives it the type Int. Handy!

Note: If you are not sure about the inferred type of a variable, there is an easy way to find out. Simply hold down the **Alt** key and hover your cursor over the variable in question. The variable will be highlighted in blue and your cursor will turn into a question mark. Now, click on the variable and you will get a handy pop up which tells you the type of the variable as well as the source file in which the variable was declared.



Discover the inferred type for a variable

In fact, now that you know about type inference, you don't need to specify `Int` for the other instance variables either:

```
var currentValue = 0
var targetValue = 0
```

► Make the above changes.

Thanks to type inference, you only have to list the name of the data type when you're not giving the variable an initial value. But most of the time, you can safely make Swift guess at the type.

I think type inference is pretty sweet! It will definitely save you some, uh, typing (in more ways than one!).

Update 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
```

Personally, I prefer the shorthand `+=` version, but either one is okay. Both accomplish exactly the same thing.

Display the score

In order to show your current score, you're going to do exactly 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 without my help. You've already done these things before for the target value label, so you should be able to repeat these steps by yourself for the score label.

Done? You should have done the following. You add 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 popup 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 the view controller, won't work.

There is more than one way to skin a cat, or, connect outlets :]

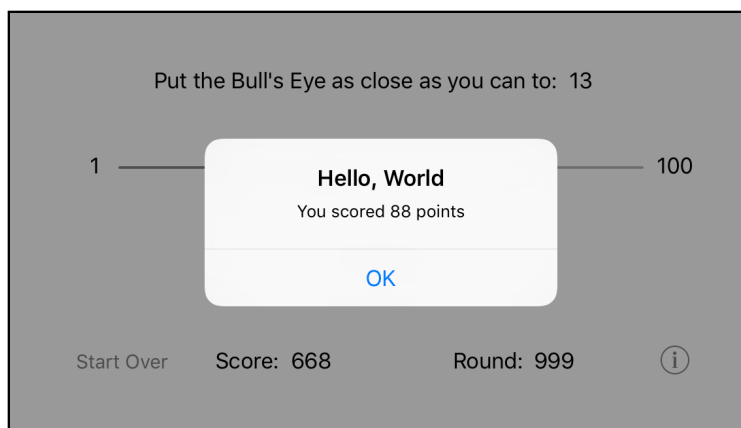
Great, that gives you a `scoreLabel` outlet that you can use to display the score. Now where in the code can you do that? In `updateLabels()`, of course.

► 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 were right. You should have added 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.

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?

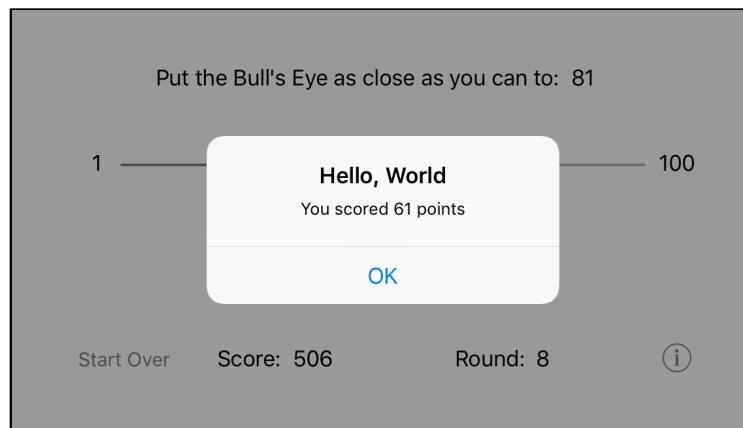
I'd say the `startNewRound()` method is a pretty good place. After all, you call this method whenever you start a new round. It makes sense to increment the round counter there.

► 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 **04 - 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.

Chapter 5: Polish

By Fahim Farook and Matthijs Hollemans

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. As far as I can tell, there are no bugs either. 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 and 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, *Bull's Eye*, but I have 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.

Exercise: Think of a way to accomplish this. Where would you put this logic and how would you program it? Hint: there are an awful lot of “if’s” in the preceding sentences.

The right place for this logic is `showAlert()`, because that is where you create the `UIAlertController`. You already do some calculations to create the message text and now you will do something similar for the title text.

► Here is the changed method in its entirety - replace the existing method with it:

```
@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 local string 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 a fixed text.

Constant initialization

In the above code, did you notice that `title` was declared explicitly as being a `String` value? And did you ask yourself why type inference wasn’t used there instead? Also, you might have noticed that `title` is actually a constant and yet the code appears to set its value in multiple places. How does that work?

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 in your own project! 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 fine, and the compiler error would go away and everything would work fine. But you’ve got to ask yourself, do you really need a variable there? Or, would a constant do? I personally prefer 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. But you can decide to carve out your own path 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 for 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".

```

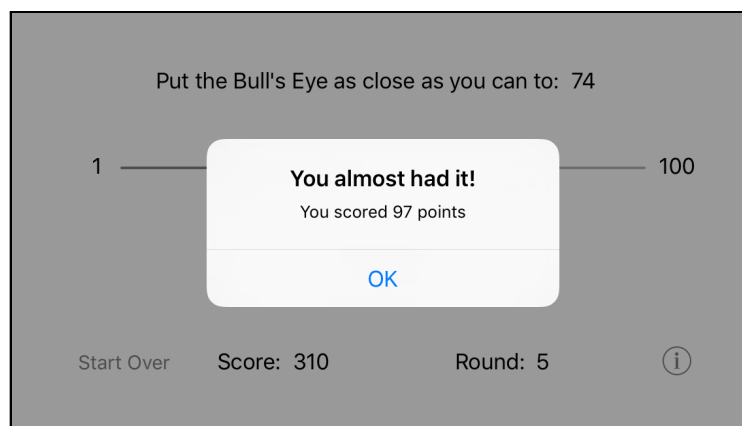
59     let title: String
60     if difference == 0 {
61         title = "Perfect!"
62     } else if difference < 5 {
63         title = "You almost had it!"
64     } else if difference < 10 {
65         title = "Pretty good!"
66     // } else {
67     //     title = "Not even close..."
68     }
69
70     let message = "You scored \(points) points"
71
72     let alert = UIAlertController(title: title,
73                                  message: message,
74                                  preferredStyle: .alert)

```

Constant 'title' used before being initialized

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 bull's eye 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 is how I would have made 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
        }
    } else if difference < 10 {
        title = "Pretty good!"
    } else {
        title = "Not even close..."
    }
    score += points                // move this line here
    . . .
}
```

You should notice a few things:

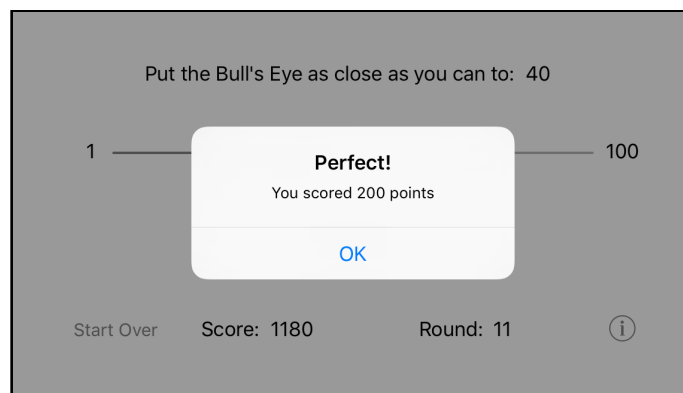
- In the first `if` you'll see a new statement between the curly brackets. When the difference is equal to zero, you now not only set `title` to "Perfect!" but also award an extra 100 points.
- The second `if` has changed too. There is now an `if` inside another `if`. Nothing wrong with that! You want 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 was 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 changed 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...

Local variables recap

I would like to point out once more the difference between local variables and instance variables. As you should know by now, a local variable only exists for the duration of the method that it is defined in, while an instance variable exists as long as the view controller (or any object that owns it) exists. The same thing is true for constants.

In `showAlert()`, there are six locals and you use three instance variables:

```
let difference = abs(targetValue - currentValue)
var points = 100 - difference
let title = . . .
score += points
let message = . . .
let alert = . . .
let action = . . .
```

Exercise: Point out which are the locals and which are the instance variables in the `showAlert()` method. Of the locals, which are variables and which are constants?

Locals are easy to recognize, because the first time they are used inside a method their name is preceded with `let` or `var`:

```
let difference = . . .  
var points = . . .  
let title = . . .  
let message = . . .  
let alert = . . .  
let action = . . .
```

This syntax creates a new variable (`var`) or constant (`let`). Because these variables and constants are created inside the method, they are locals.

Those six items – `difference`, `points`, `title`, `message`, `alert`, and `action` – are restricted to the `showAlert()` method and do not exist outside of it. As soon as the method is done, the locals cease to exist.

You may be wondering how `difference`, for example, can have a different value every time the player taps the Hit Me button, even though it is a constant – after all, aren't constants given a value just once, never to change afterwards?

Here's why: each time a method is invoked, its local variables and constants are created anew. The old values have long been discarded and you get brand new ones.

When `showAlert()` is called, it creates a completely new instance of `difference` that is unrelated to the previous one. That particular constant value is only used until the end of `showAlert()` and then it is discarded.

The next time `showAlert()` is called after that, it creates yet another new instance of `difference` (as well as new instances of the other locals `points`, `title`, `message`, `alert`, and `action`). And so on... There's some serious recycling going on here!

But inside a single invocation of `showAlert()`, `difference` can never change once it has a value assigned. The only local in `showAlert()` that can change is `points`, because it's a `var`.

The instance variables, on the other hand, are defined outside of any method. It is common to put them at the top of the file:

```
class ViewController: UIViewController {  
    var currentValue = 0  
    var targetValue = 0  
    var score = 0  
    var round = 0
```

As a result, you can use these variables inside any method, without the need to declare them again, and they will keep their values till the object holding them (the view controller in this case) ceases to exist.

If you were to do this:

```
@IBAction func showAlert() {  
    let difference = abs(targetValue - currentValue)  
    var points = 100 - difference  
  
    var score = score + points    // doesn't work!  
    . . .  
}
```

Then things wouldn't work as you'd expect them to. Because you now put `var` in front of `score`, you have made it a new local variable that is only valid inside this method.

In other words, this won't add points to the *instance variable* `score` but to a new *local variable* that also happens to be named `score`. The instance variable `score` never gets changed, even though it has the same name.

Obviously that is not what you want to happen here. Fortunately, the above won't even compile. Swift knows there's something fishy about that line.

Note: To make a distinction between the two types of variables, so that it's always clear at a glance how long they will live, some programmers prefix the names of instance variables with an underscore.

They would name the variable `_score` instead of just `score`. Now there is less confusion because names beginning with an underscore won't be mistaken for being locals. This is only a convention. Swift doesn't care one way or the other how you spell your instance variables.

Other programmers use different prefixes, such as "m" (for member) or "f" (for field) for the same purpose. Some even put the underscore *behind* the variable name. Madness!

The alert

There is something that bothers me about the game. You may have noticed it too...

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 already gets started 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 popup. 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 popup:

```
@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 popup 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 already starts before the alert popup 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 popup 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. To change this, you'll need to give the `UIAlertAction` some code to execute when the 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 take it from there.

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 something new: a *closure*.

► Change the bottom bit of `showAlert()` to:

```
@IBAction func showAlert() {  
    let alert = UIAlertController(. . .)  
    let action = UIAlertAction(title: "OK", style: .default,  
                              handler: { action 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 it and see for yourself. I think the game feels a lot better this way.

Self

You may be wondering why in the handler block you did `self.startNewRound()` instead of just writing `startNewRound()` like before.

The `self` keyword allows the view controller to refer to itself. That shouldn't be too strange a concept. When you say, "I want ice cream," you use the word "I" to refer to yourself. Similarly, objects can talk about (or to) themselves as well.

Normally you don't need to use `self` to send messages to the view controller, even though it is allowed. The exception: inside closures you *do* have to use `self` to refer to the view controller.

This is a rule in Swift. If you forget `self` in a closure, Xcode doesn't want to build your app (try it out). This rule exists because closures can "capture" variables, which comes with surprising side effects. You'll learn more about that in later chapters.

Start over

No, you're not going to throw away the source code and start this project all over! I'm talking 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. I suggest you 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 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 valid. Personally, I like to have less code since that 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 if 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.

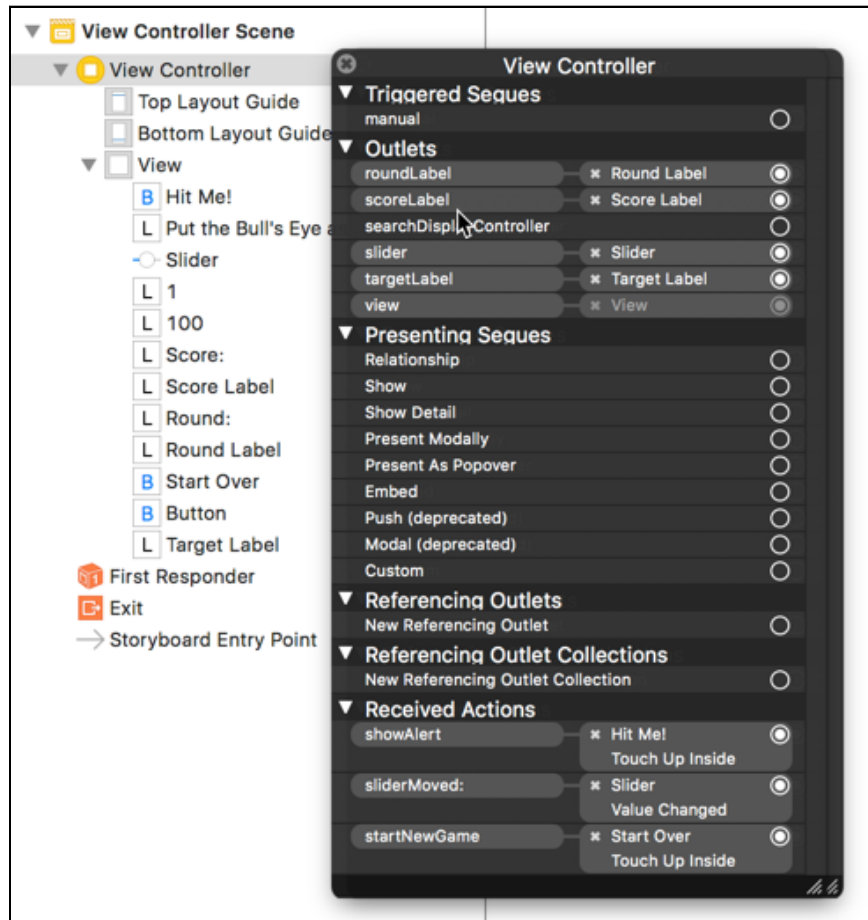
► Open the storyboard and Control-drag from the **Start Over** button to **View Controller**. Let go of the mouse button and pick **startNewGame** from the popup if you opted to have `startNewGame()` as the action method. Otherwise, pick the name of your action method.

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 popup, or simply view the connections in the **Connections inspector**. This shows all the connections for the view controller.



All the connections from View Controller to the other objects

Now your game is pretty polished and your task list is getting ever shorter :]

You can find the project files for the current version of the app under **05 - Polish** in the Source Code folder.

Chapter 6: The New Look

By Fahim Farook and Matthijs Hollemans

Bull's Eye 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, I'm not sure many people would be excited to download it. Fortunately, iOS makes it easy for you to create good-looking apps, so let's give *Bull's Eye* 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

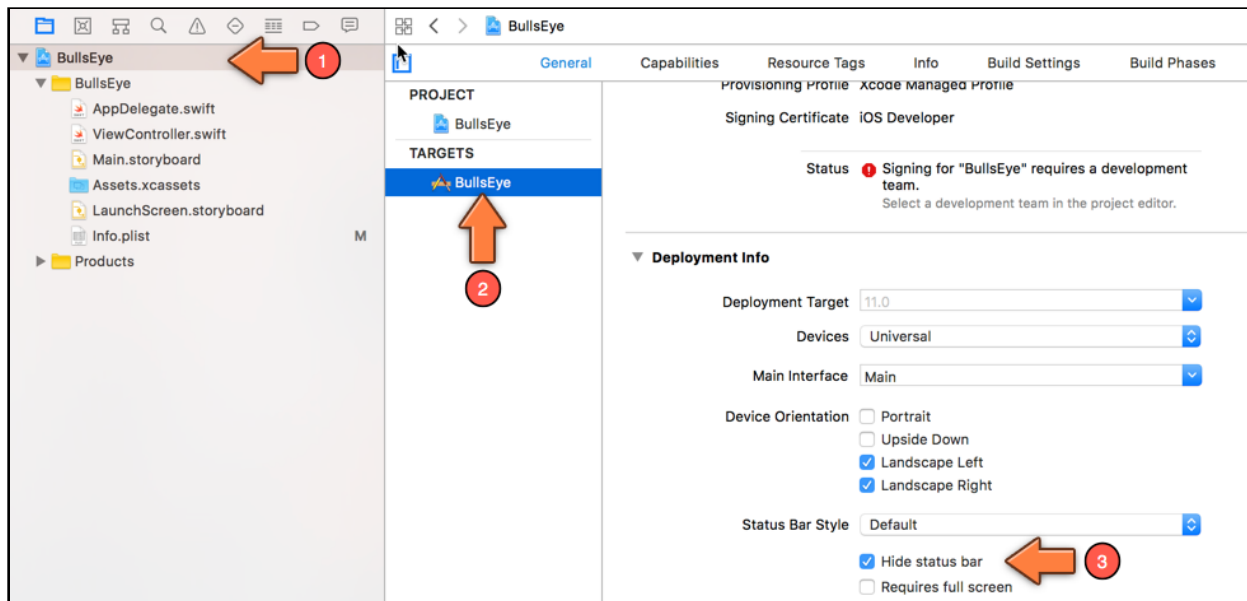
First, let's quickly revisit another item in the to-do list - "Put the app in landscape orientation." You already did this, right? But there's a little bit of clean up to be done with regards to that item.

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.

Even though the system automatically handles not showing the status bar for your game, there is still one thing you can do to improve the way *Bull's Eye* handles the status bar.

► Go to the **Project Settings** screen and scroll down to **Deployment Info**. Under **Status Bar Style**, check the option **Hide status bar**.

This will ensure that the status bar is hidden during application launch.



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, and 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 app is that great apps get all the small details right.

► That's it. Run the app and you'll see that the status bar is history.

Info.plist

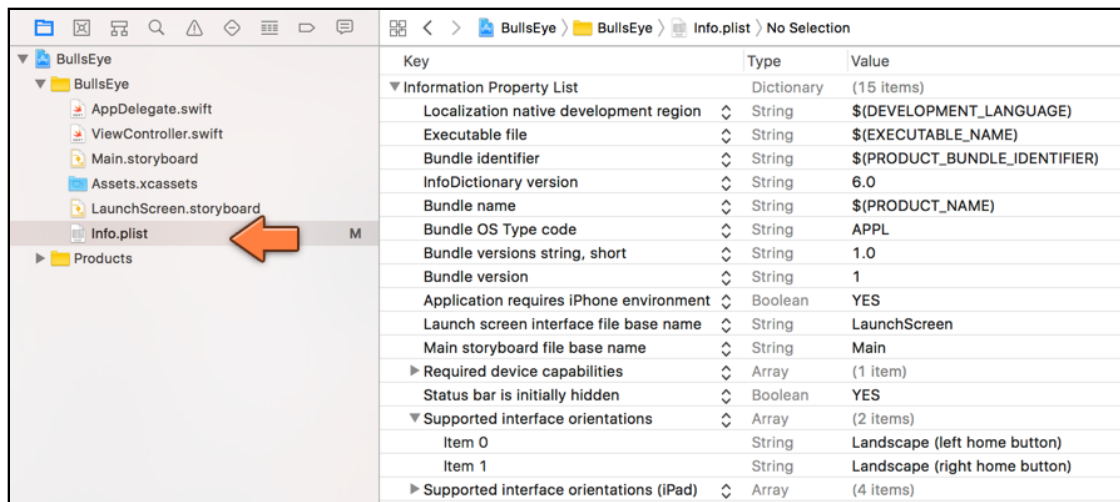
Most of the options from the Project Settings screen, such as the supported device orientations and whether the status bar is visible during launch, get stored in your app's Info.plist file.

Info.plist is a configuration file inside the application bundle that tells iOS how the app will behave. It also describes certain characteristics of the app, such as the version number, that don't really fit anywhere else.

With some earlier versions of Xcode, you often had to edit Info.plist by hand, but with the latest Xcode versions this is hardly necessary anymore. You can make most of the changes directly from the Project Settings screen.

However, it's good to know that Info.plist exists and what it looks like.

► Go to the **Project navigator** and select the file named **Info.plist** to take a peek at its contents.

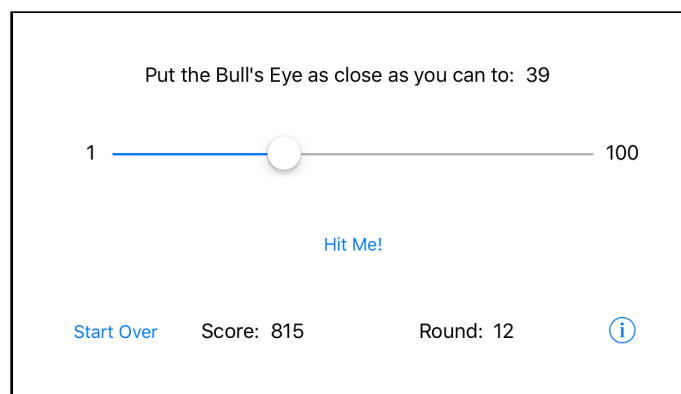


The Info.plist file is just a list of configuration options and their values. Most of these may not make sense to you, but that's OK – they don't always make sense to me either.

Notice the option **Status bar is initially hidden**. It has the value YES. This is the option that you just changed.

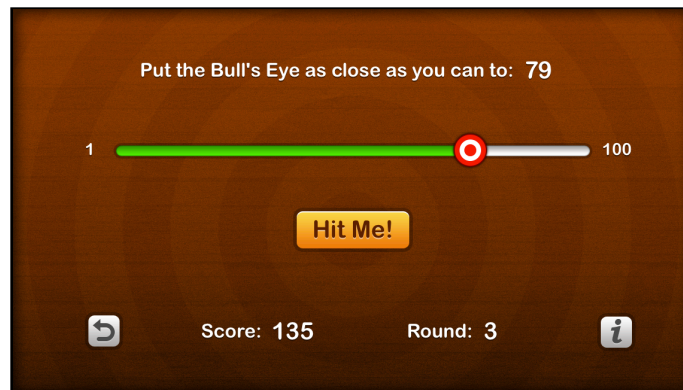
Spice up the graphics

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:



Cool :-)

The actual controls don't change. You'll simply use 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.

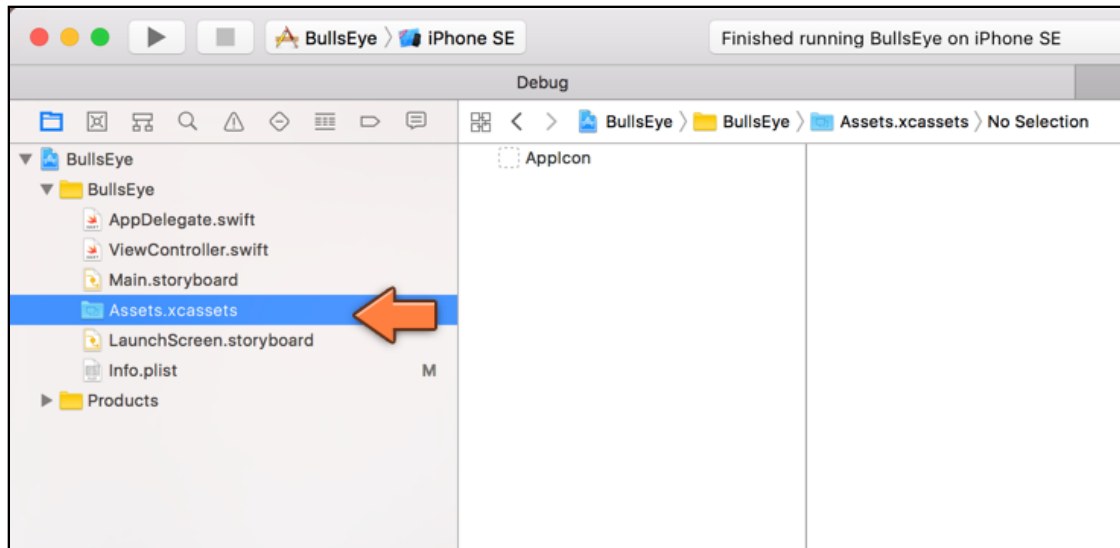
Add the image assets

If you are artistically challenged, then don't worry, I have 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 will first import these images into the Xcode project.

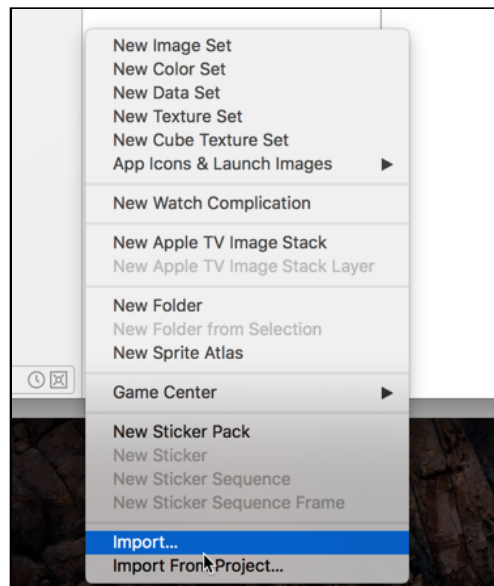
► In the **Project navigator**, find **Assets.xcassets** and click on it.

This is known as the asset catalog for the app and it contains all the app's images. Right now, it is empty and contains just a placeholder for the app icon, which you'll add soon.



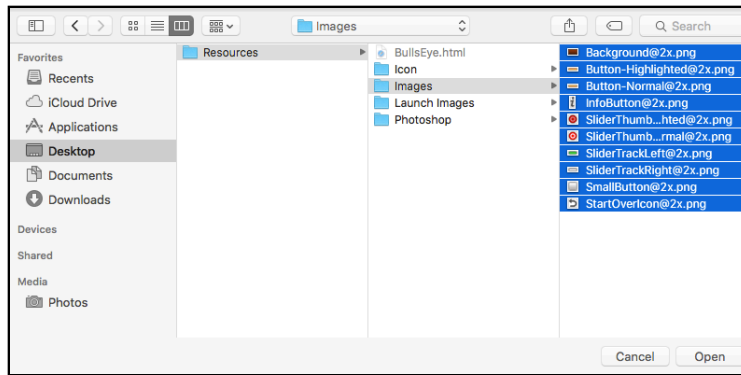
The asset catalog is initially empty

► At the bottom of the secondary pane, the one with AppIcon, there is a + button. Click it and then select the **Import...** option:



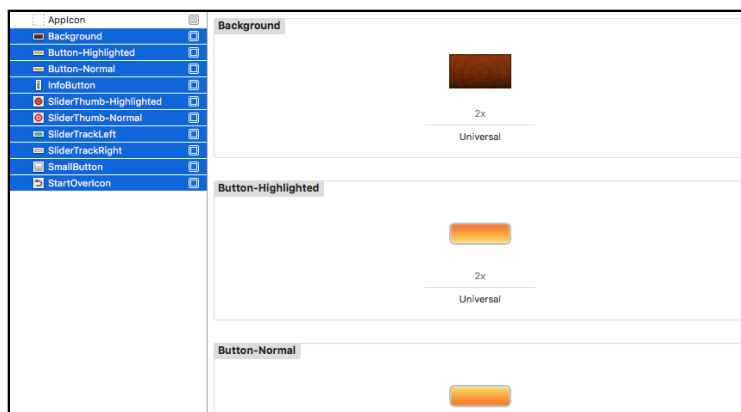
Choose Import to put existing images into the asset catalog

Xcode shows a file picker. Select the **Images** folder from the resources and press **⌘+A** to select all the files inside this folder.



Choosing the images to import

Click **Open** and Xcode copies all the image files from that folder into the asset catalog:



The images are now inside the asset catalog

If Xcode added a folder named “Images” instead of the individual image files, then try again and this time make sure that you select the files inside the Images folder rather than the folder itself before you click Open.

Note: Instead of using the **Import...** menu option as above, you could also simply drag the necessary files from Finder on to the Xcode asset catalog view. As ever, there's more than one way to do the same thing in Xcode.

1x, 2x, and 3x displays

Currently, each image set in the asset catalog has a slot for a “2x” image, but you can also specify 1x and 3x images. Having multiple versions of the same image in varying sizes allows your apps to support the wide variety of iPhone and iPad displays in existence.

1x is for low-resolution screens, the ones with the big, chunky pixels. There are no low-resolution devices in existence that can actually run iOS 11 – they are too old to bother

with – so you’re not likely to come across many 1x images anymore. 1x is only a concern if you’re working on an app that still needs to support iOS 9 or older.

2x is for high-resolution Retina screens. This covers most modern iPhones, iPod touches, and iPads. Retina images are twice as big as the low-res images, hence the 2x. The images you imported just now are 2x images.

3x is for the super high-resolution Retina HD screen of the iPhone Plus devices. If you want your app to have extra sharp images on these top-of-the-line iPhone models, then you can drop them into the “3x” slot in the asset catalog.

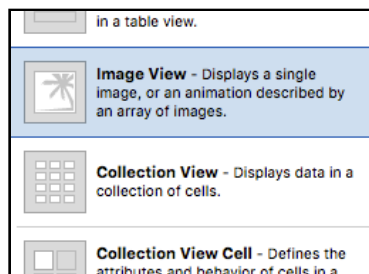
There is a special naming convention for image files. If the filename ends in **@2x** or **@3x** then that’s considered the Retina or Retina HD version. Low-resolution 1x images have no special name (you don’t have to write **@1x**).



Put up the wallpaper

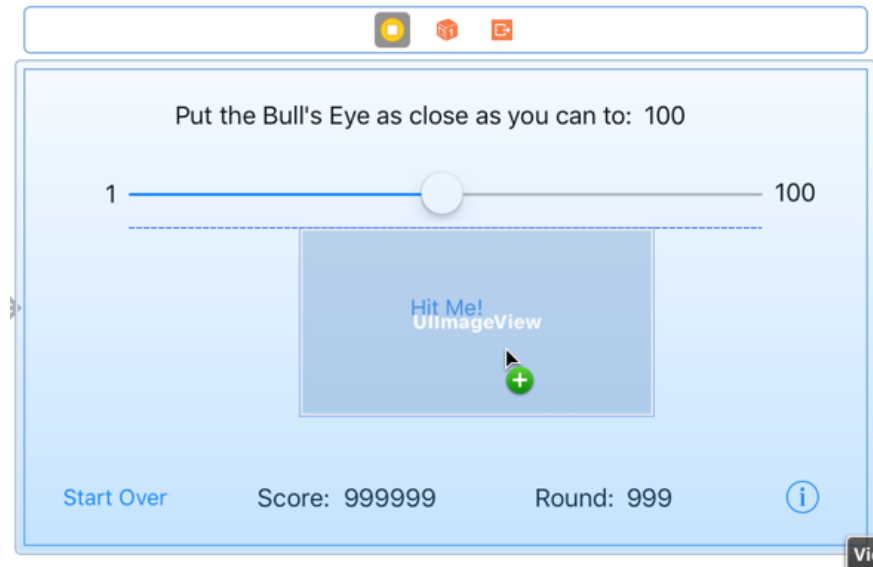
Let’s begin by changing the drab white background in *Bull’s Eye* to something more fancy.

► Open **Main.storyboard**. Go into the **Object Library** and locate an **Image View**. (Tip: if you type “image” into the search box at the bottom of the Object Library, it will quickly filter out all the other views.)



The Image View control in the Object Library

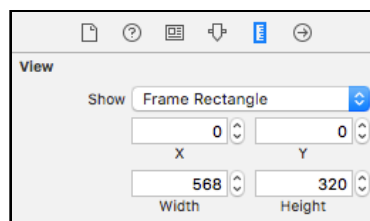
► Drag the image view on top of the existing user interface. It doesn’t really matter where you put it, as long as it’s inside the Bull’s Eye View Controller.



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 568 and Height to 320.

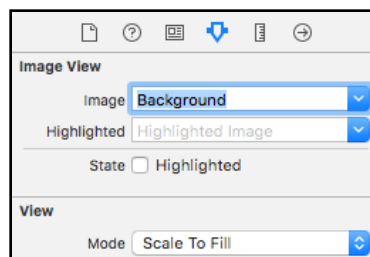
This will make the image view cover the entire screen.



The Size inspector settings for the Image View

► 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.

This will put the image named “Background” from the asset catalog into the image view.



Setting the background image on the Image View

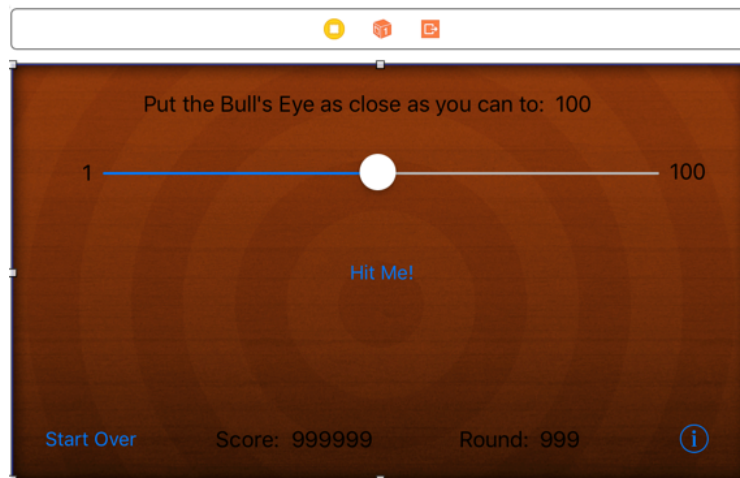
There is only one problem: the image now covers all the other controls. There is an easy fix for that; you have to move the image view behind the other views.

► 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 (it still has a few bugs) 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 View, to accomplish the same thing. (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:



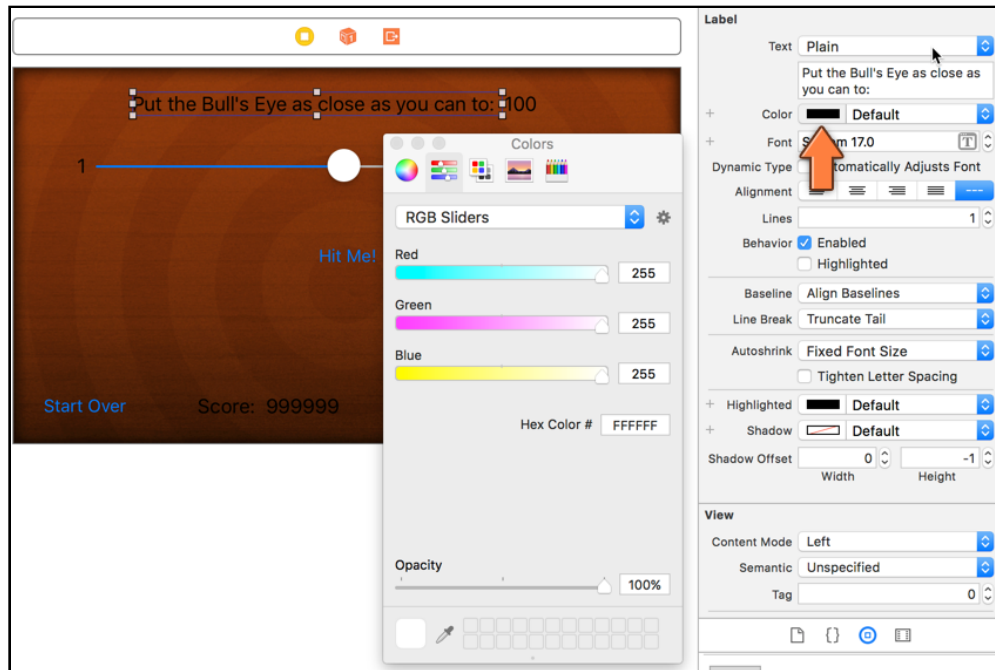
The game with the new background image

That takes care of the background. Run the app and marvel at the new graphics.

Change the labels

Because the background image is quite dark, the black text labels have become hard to read. Fortunately, Interface Builder lets you change their 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 - there are two parts to the item, so you need to click on the actual color and not the text part.



Setting the text color on the label

This opens the Color Picker, which has several ways to select colors. I prefer the sliders (second tab). If all you see is a gray scale slider, then select RGB Sliders from the select box at the top.

- Pick a pure white color, Red: 255, Green: 255, Blue: 255, Opacity: 100%.
- 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%.

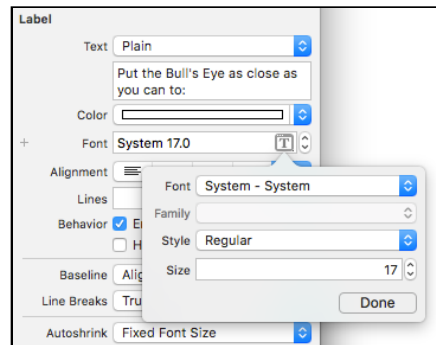
Note: Sometimes when you change the Color or Shadow attributes, the background color of the view also changes. This is a bug in Xcode. Put it back to Clear Color if that happens.

- 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.

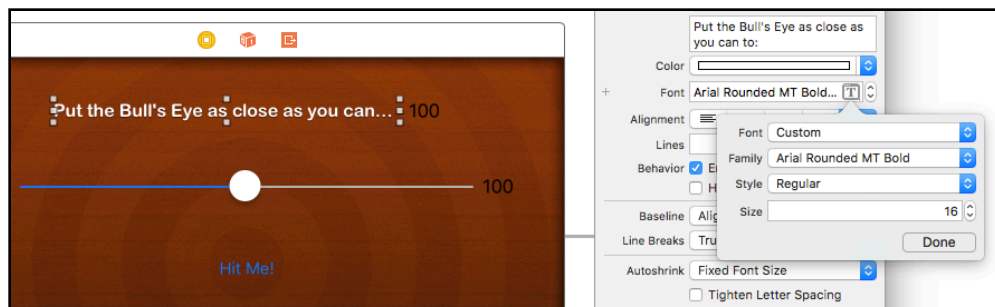
- Click on the **[T]** icon of the **Font** attribute. This opens the Font Picker.

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.)

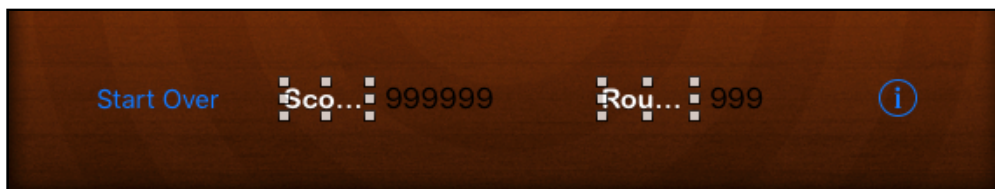
The label will now become slightly larger or smaller so that it fits snugly around the text. If the text got cut off when you changed the font, now all the text will show again.

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.

As you can see, in my storyboard the text no longer fits into the Score and Round labels:

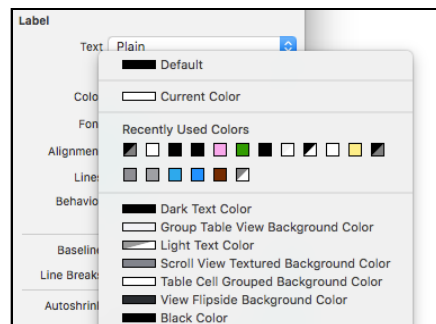


The font is too large to fit all the text in the Score and Round labels

You can either make the labels larger by dragging their handles to resize them manually, or you can use the **Size to Fit Content** option (**⌘=**). I prefer the latter because it's less work.

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.

Click the tiny arrows at the end of the color field (or, if there is a text name for the color, click on the text part) and the menu will pop up:

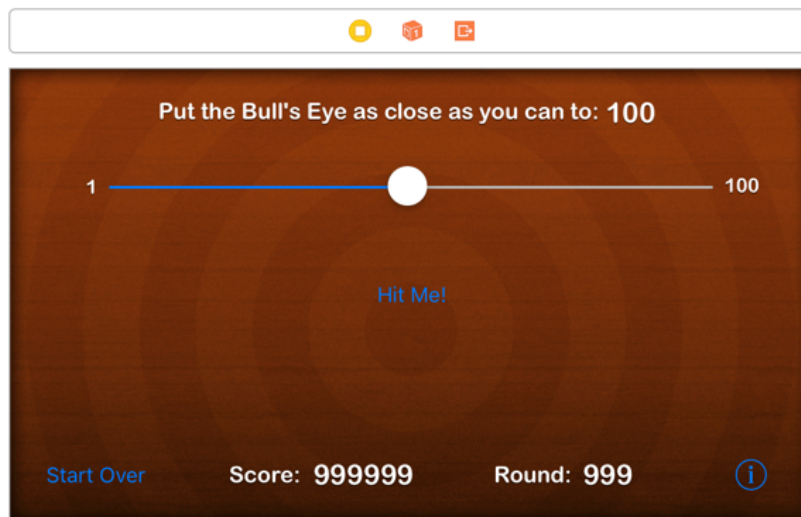


Quick access to recently used colors and several handy presets

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.

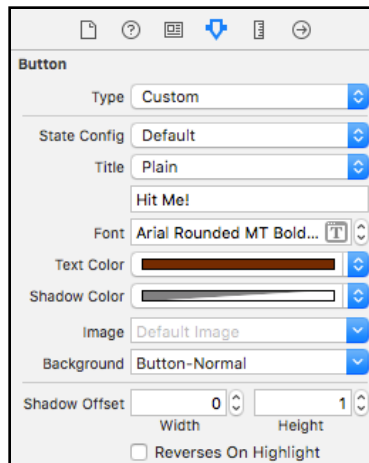
- Still in the **Attributes inspector**, press the arrow on the **Background** field and choose **Button-Normal** from the list.
- Set the **Font** to **Arial Rounded MT Bold**, size 20.
- Set the **Text Color** to red: 96, green: 30, blue: 0, opacity: 100%. This is a dark brown color.
- Set the **Shadow 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

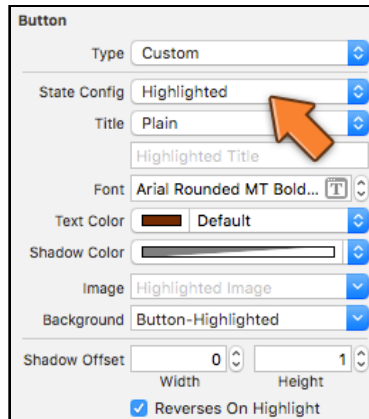
Buttons can have more than one state. When you tap a button and hold it down, it should appear “pressed down” to let you know that the button will be activated when you lift your finger. This is known as the *highlighted* state and is an important visual cue to the user.

- 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.

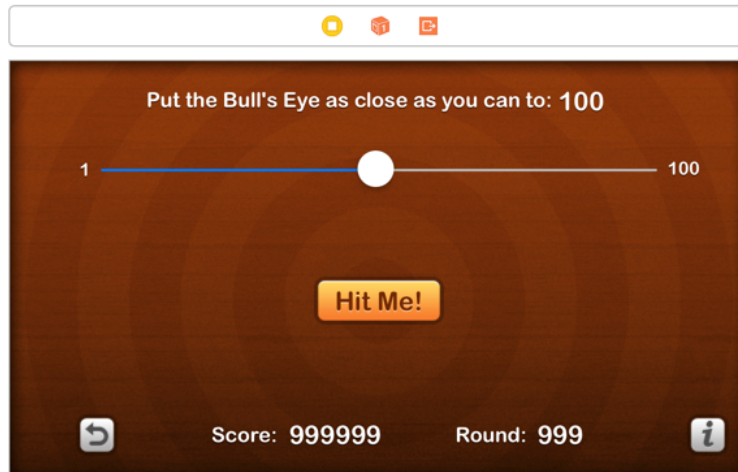
► Select the **Start Over** button and change the following attributes:

- 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 source code.

Everything you have 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)
```

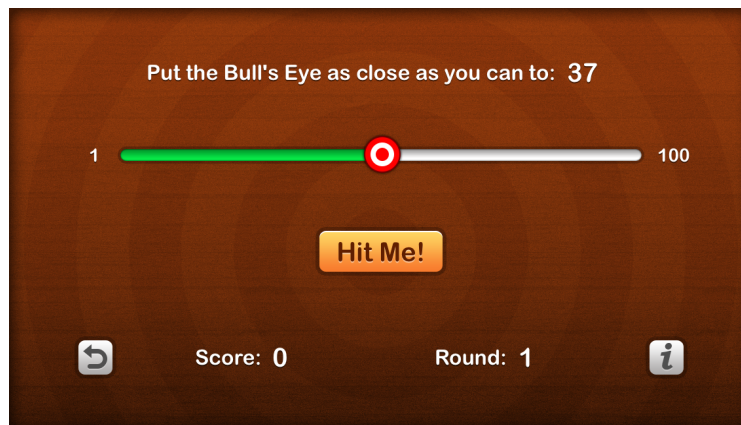
```
let trackRightImage = UIImage(named: "SliderTrackRight")!
let trackRightResizable =
    trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

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.

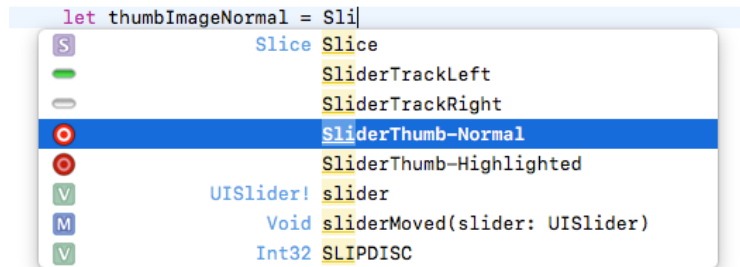
Tip: Xcode now has a handy new feature that makes it really easy to add images in your code. Instead of writing:

```
let thumbImageNormal = UIImage(named: "SliderThumb-Normal")
```

You can now type:


```
let thumbImageNormal = Sli
```


And Xcode's autocomplete will kick in and show a list of suggestions to complete the text `Sli`, including any images whose names start with those letters.




Xcode autocomplete also shows images


Pick **SliderThumb-Normal** from the list and it will add a tiny icon of the image into the code! This tiny icon is known as an *image literal*. If you do the same for the other images, your code will look like this:

```
// Customize slider
let thumbImageNormal = 
slider.setThumbImage(thumbImageNormal, for: .normal)

let thumbImageHighlighted = 
slider.setThumbImage(thumbImageHighlighted, for: .highlighted)

let insets = UIEdgeInsets(top: 0, left: 14, bottom: 0, right: 14)

let trackLeftImage = 
let trackLeftResizable =
    trackLeftImage.resizableImage(withCapInsets: insets)
slider.setMinimumTrackImage(trackLeftResizable, for: .normal)

let trackRightImage = 
let trackRightResizable =
    trackRightImage.resizableImage(withCapInsets: insets)
slider.setMaximumTrackImage(trackRightResizable, for: .normal)
```

The images are now part of your source code

Give it a try! I really like how it shows a tiny thumbnail of the image right in the code.

Run your app once again to verify that adding the image literals did not change the functionality of the game in any way. It shouldn't, but it's always good to be sure, right?

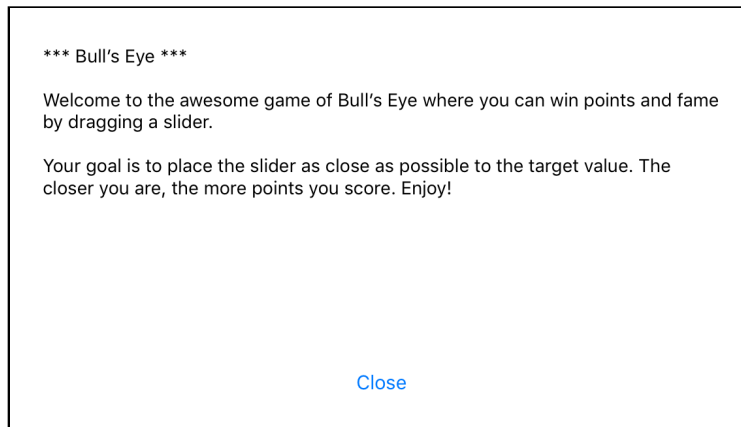
The About Screen

Your game looks awesome and your to-do list is done. So, does this mean that you are done with *Bull's Eye*?

Not so fast :] Remember the ⓘ button on the game screen? Try tapping it. Does it do anything? No?

Ooops! 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 which shows some information about the game and have it display when the user taps on the ⓘ button.

Initially, the screen will look something like this (but we'll prettify it soon enough):



The new About screen

This new screen contains a *text view* with the gameplay rules and a button to close the screen.

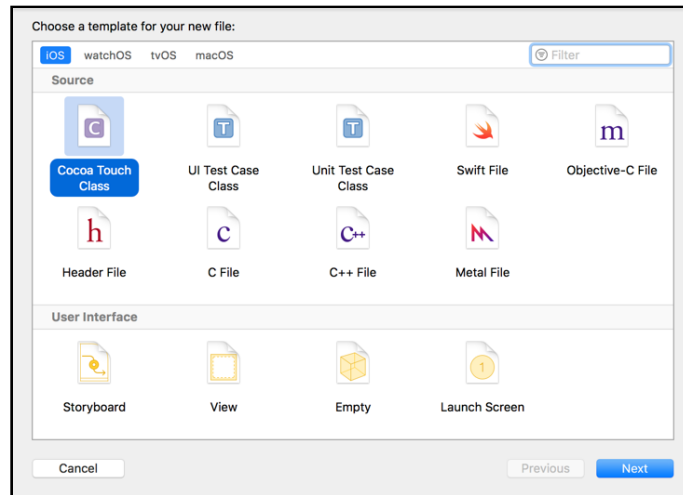
Most apps have more than one screen, even very simple games. So, this is as good a time as any to learn how to add additional screens to your apps.

I have pointed it out a few times already: 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.

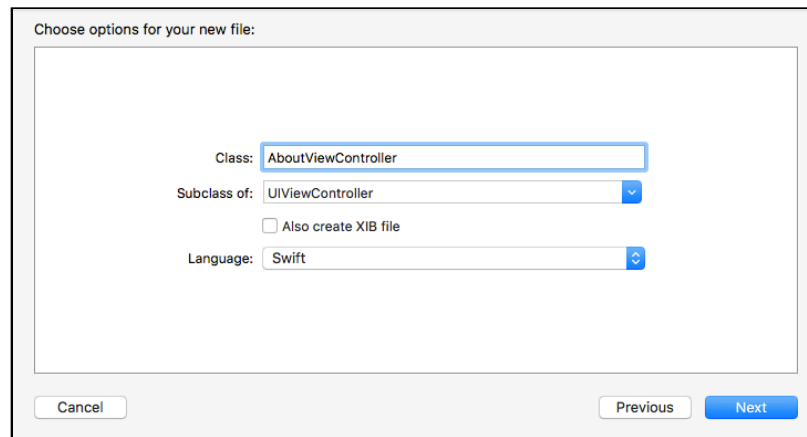
Add a new view controller

➤ 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).



Choosing the file template for Cocoa Touch Class

Click **Next**. Xcode gives you some options to fill out:

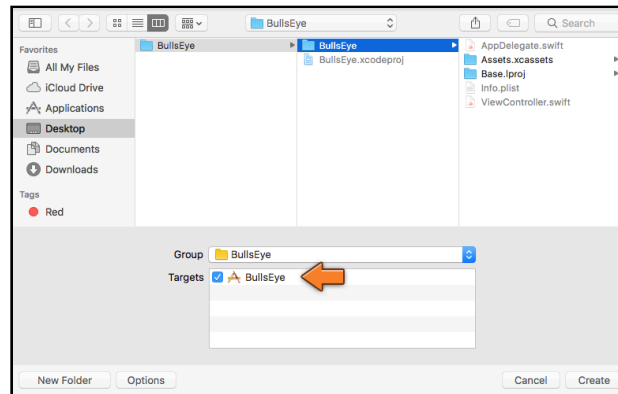


The options for the new file

Choose the following:

- Class: **AboutViewController**
- Subclass of: **UIViewController**
- Also create XIB file: Leave this box unchecked.
- Language: **Swift**

Click **Next**. Xcode will ask you where to save this new view controller.



Saving the new file

- 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**. (If you don't see this panel, click the Options button at the bottom of the dialog.)

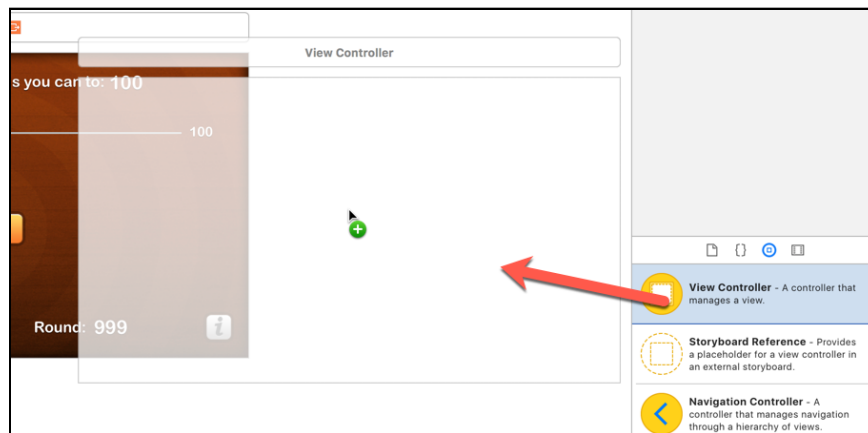
- Click **Create**.

Xcode will create a new file and add it to your project. As you might have guessed, the new file is **AboutViewController.swift**.

Design 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 **Object Library**, choose **View Controller** and drag it on to the canvas, to the right of the main View Controller.

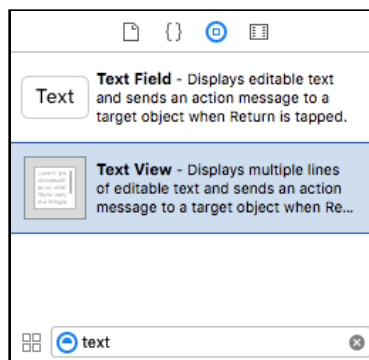


Dragging a new View Controller from the Object Library

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 in 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.

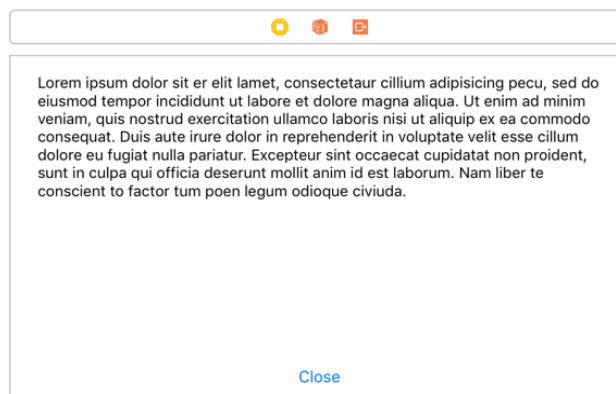
You can find these components in the Object Library. If you don't feel like scrolling, you can filter the components by typing in the field at the bottom:



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 make its content is editable. By default, the Text View contains a bunch of Latin placeholder text (also known as “Lorem Ipsum”).

Copy-paste this new text into the Text View:

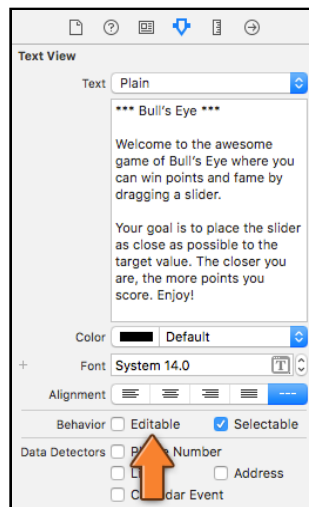
*** Bull's Eye ***

Welcome to the awesome game of Bull's Eye 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 paste 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 Attributes inspector for the text view

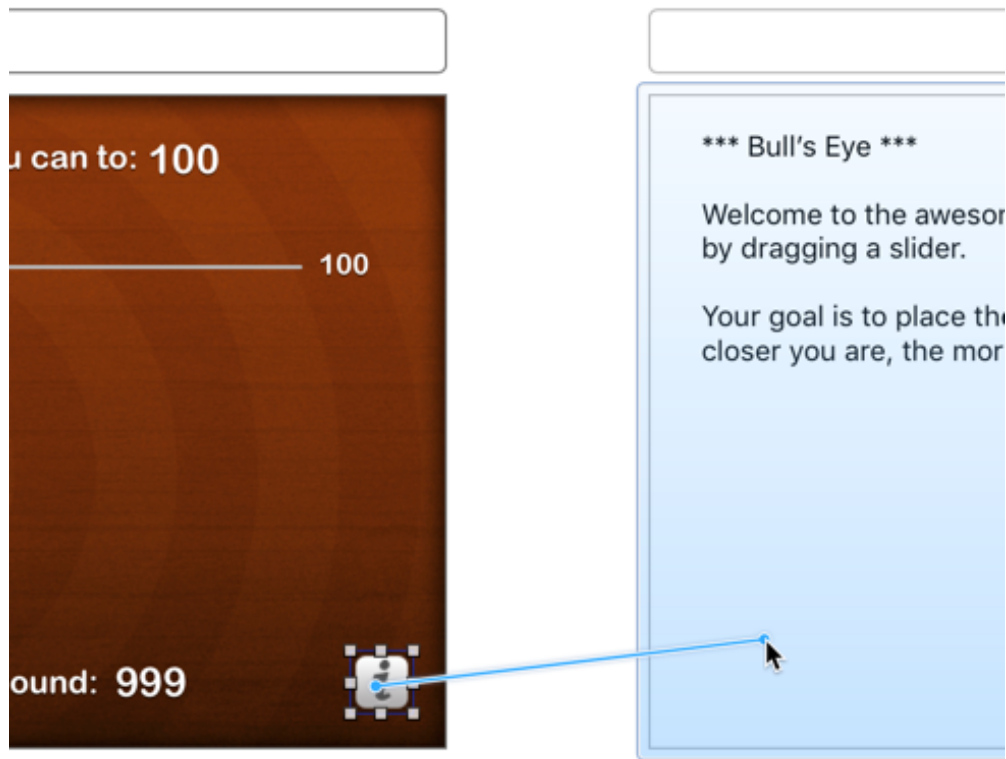
That's the design of the screen done for now.

Show the new view controller

So how do you open this new About screen when the user presses the ⓘ button?

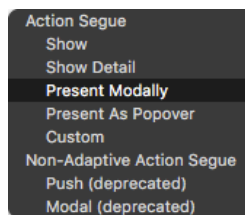
Storyboards have a neat trick for this: *segues* (pronounced “seg-way” like the silly scooters). A segue is a transition from one screen to another. They are really easy to add.

► 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

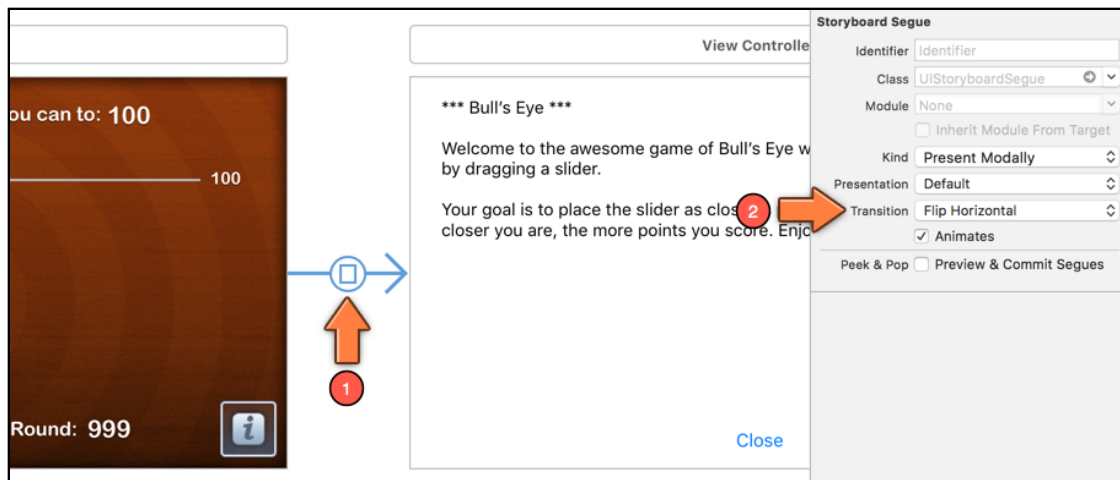
► Let go of the mouse button and a popup appears with several options. Choose **Present Modally**.



Choosing the type of segue to create

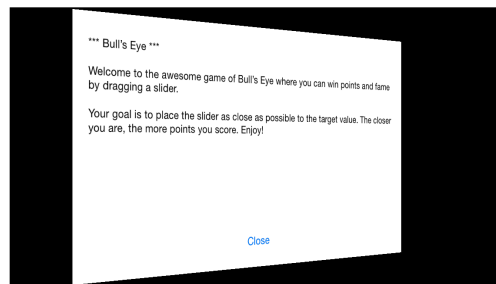
Now an arrow will appear between the two screens. This arrow represents the segue from the main scene to the About scene.

► 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.

Dismiss the About view controller

However, there is an obvious shortcoming 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 to me, 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 {

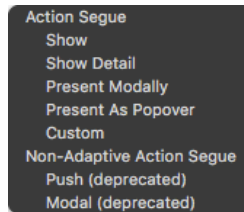
    @IBAction func close() {
        dismiss(animated: true, completion: nil)
    }
}
```

This 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 popup, but it isn't. Instead, this is the same popup you saw when you made the segue:



The “close” action is not listed in the popup

Exercise: Bonus points if you can spot the error. It's a very common – and frustrating! – mistake.

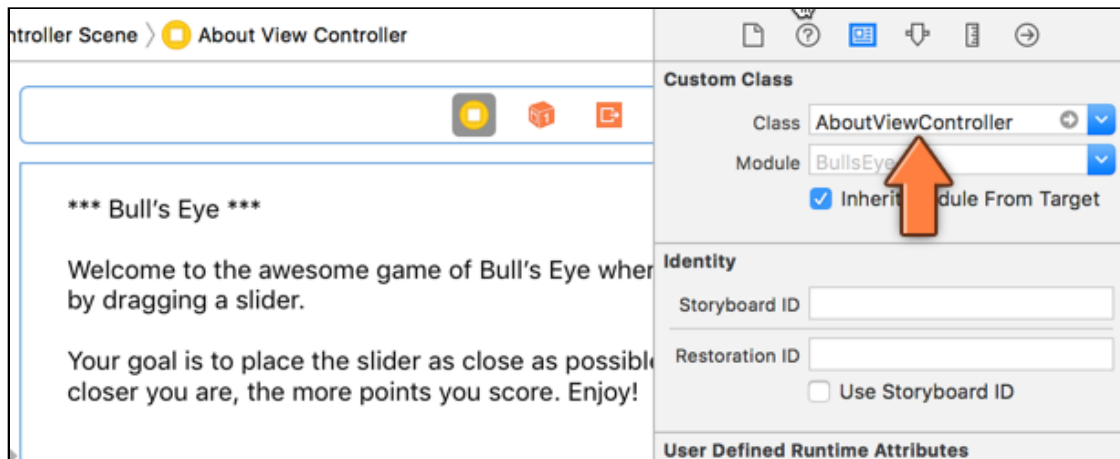
The problem is that this scene in the storyboard doesn't know yet that it is supposed to represent the `AboutViewController`.

Set 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, in fact, belongs to `AboutViewController`. (That's why in the Document Outline it just says View Controller and not About View Controller.)

► Fortunately, this is easily remedied. In Interface Builder, select the About scene's **View Controller** and go to the **Identity inspector** (that's the button to the left of the Attributes inspector).

► Under **Custom Class**, type **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 storyboard). This should be old hat by now. The popup 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. 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?

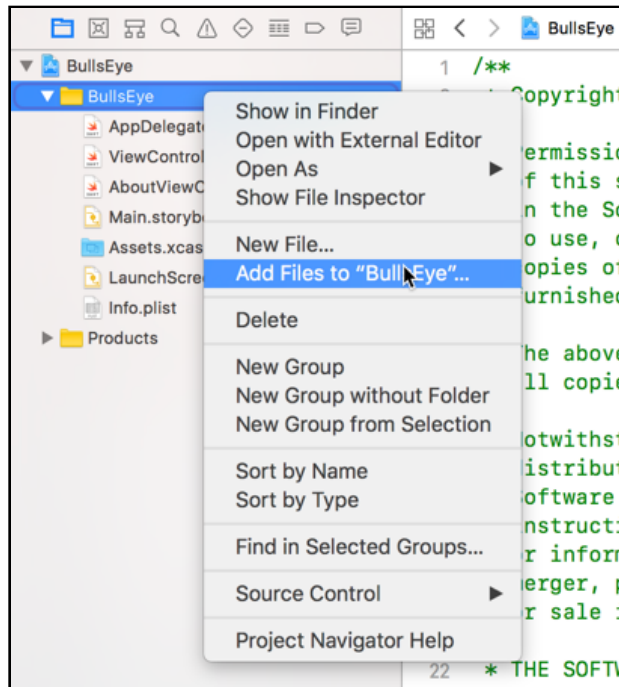
Use a web view for HTML content

- 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 next.)
- Put a **Web View** in its place (as always, you can find this view in the Object Library).

A web view, as its name implies, can show web pages. All you have to do is give it a URL to a web site or the name of a file to load. The web view object is named `UIWebView`.

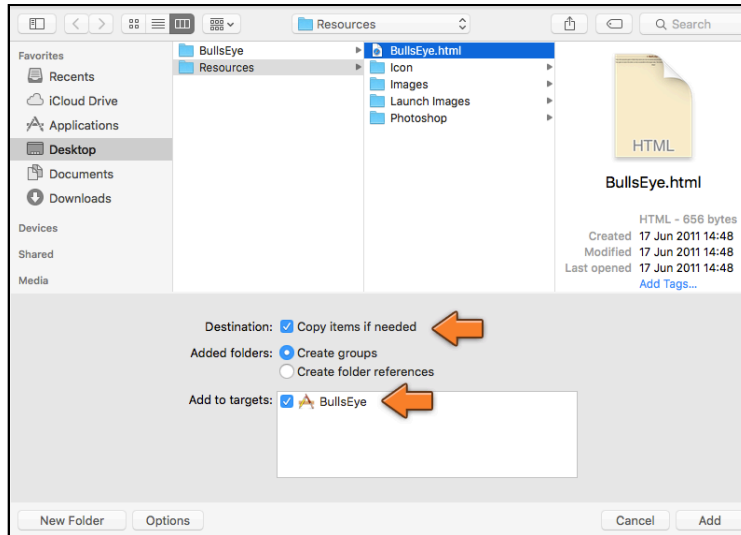
For this app, you will make it display a static HTML page from the application bundle, so it won't actually have to go online and download anything.

- 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.



Choosing the file to add

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: UIWebView!
    . . .
}
```

► In the storyboard file, connect the UIWebView 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**.

(If you do it the other way around, from the Web View to About View Controller, then you'll connect the wrong thing and the web view will stay empty when you run the app.)

► In **AboutViewController.swift**, add a `viewDidLoad()` implementation:

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

    if let url = Bundle.main.url(forResource: "BullsEye",
                                withExtension: "html") {
        if let htmlData = try? Data(contentsOf: url) {
            let baseURL = URL(fileURLWithPath: Bundle.main.bundlePath)
            webView.load(htmlData, mimeType: "text/html",
                        textEncodingName: "UTF-8",
                        baseURL: baseURL)
        }
    }
}
```

This displays the HTML file using the web view.

The code may look scary but what goes on is not really that complicated: first it finds the **BullsEye.html** file in the application bundle, then loads it into a `Data` object, and finally it asks the web view to show the contents of this data object.

► 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

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 **06 - The New Look** in the Source Code folder.

Chapter 7: The Final App

By Fahim Farook and Matthijs Hollemans

You might be thinking, "OK, *Bull's Eye* is now done, and I can move on to the next app!" If you were, I'm afraid you are in for disappointment - there's just a teensy bit more to do in the game.

"What? What's left to do? We finished the task list!" you say? You are right. The game is indeed complete. However, all this time, you've been developing and testing for a 4" iPhone screen found on devices such as the iPhone 5, 5c, and SE. But what about other iPhones such as the 4.7-inch iPhone, the 5.5-inch iPhone Plus, or the 5.8-inch iPhone X which have bigger screens? Or the iPad with its multiple screen sizes? Will the game work correctly on all these different screen sizes?

And if not, shouldn't we fix it?

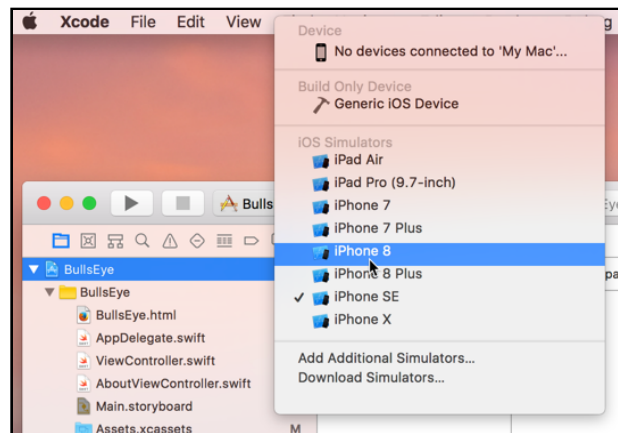
This chapter covers the following:

- **Support different screen sizes:** Ensure that the app will run correctly on all the different iPhone and iPad screen sizes.
- **Crossfade:** Add some animation to make the transition to the start of a new game a bit more dynamic.
- **The icon:** Add the app icon.
- **Display name:** Set the display name for the app.
- **Run on device:** How to configure everything to run your app on an actual device.

Support different screen sizes

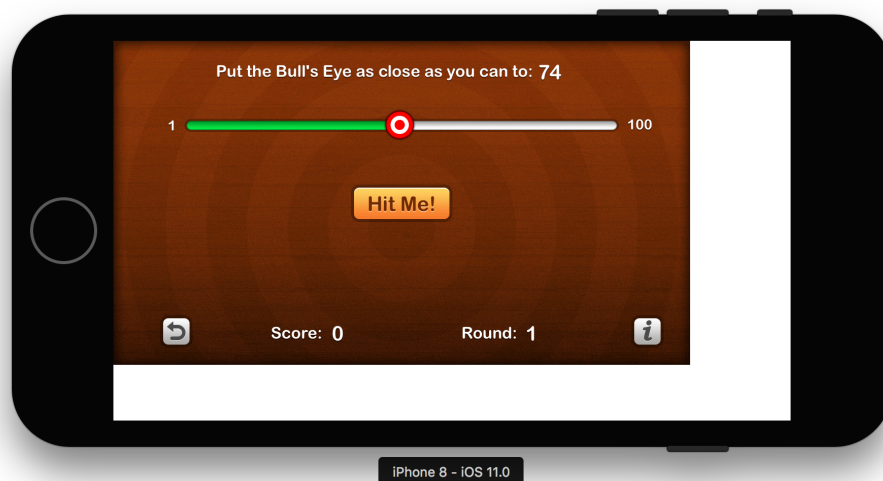
First, let's check if there is indeed an issue running Bull's Eye on a device with a larger screen. It's always good to verify that there's indeed an issue before we do extra work, right? Why fix it, if it isn't broken? :]

► To see how the app looks on a larger screen, run the app on an iPhone simulator like the **iPhone 8**. 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 up the entire screen

Obviously, this won't do. Not everybody is going to be using a 4" iOS device. And you don't want the game to display on only part of the screen for the rest of the people!

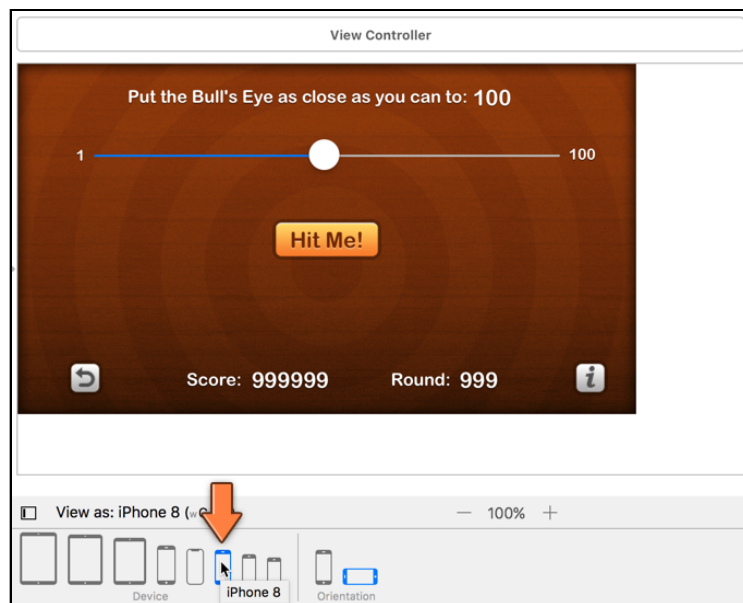
This is a good opportunity to learn about *Auto Layout*, a core UIKit technology that makes it easy to support many different screen sizes in your apps, including the larger screens of the 4.7-inch, 5.5-inch, and 5.8-inch iPhones, and the iPad.

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 iPad one, can be monsters! Also, with Xcode 9 onwards, you can resize a simulator window by simply dragging on 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** device. (You may need to change the orientation back to landscape.)



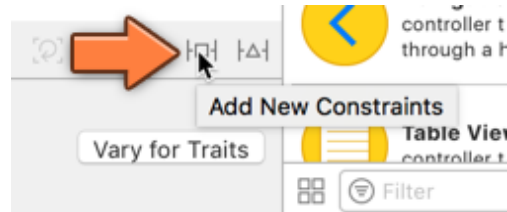
Viewing the storyboard on iPhone 8

The storyboard should look like your screen from when you ran on the iPhone 8 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:

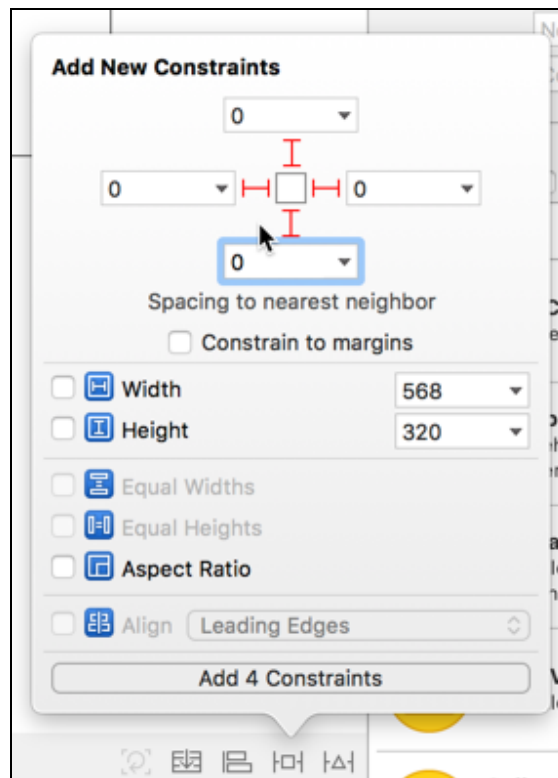


The Add New Constraints button

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 see soon enough how it works in practice.

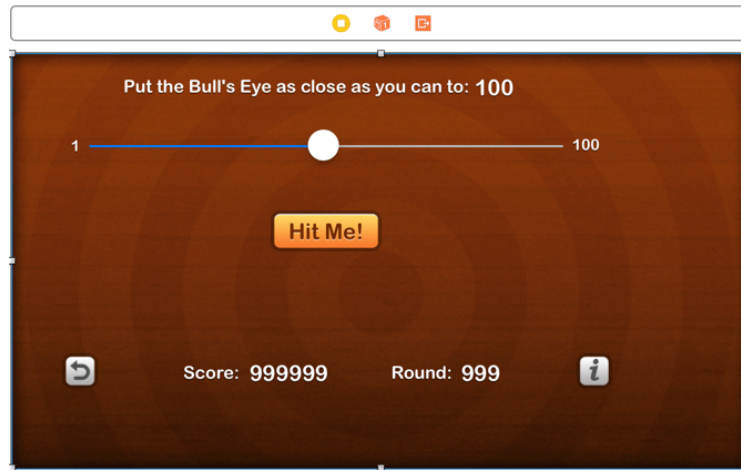
In order for the background image to stretch from edge-to-edge on the screen, the left, top, right, and bottom edges of the image should be flush against the screen edges. The way to do this with Auto Layout is to create two alignment constraints, one horizontal and one vertical.

► In the **Add New Constraints** menu, set the **left**, **top**, **right**, and **bottom** spacing to zero and make sure that the red I-beam markers next to (or below) each item is enabled. (The red I-beams are used to specify which constraints are enabled when adding new constraints.):



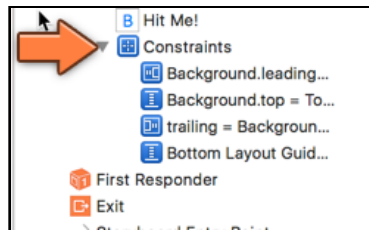
Using the Add New Constraints menu to position the background image

► 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

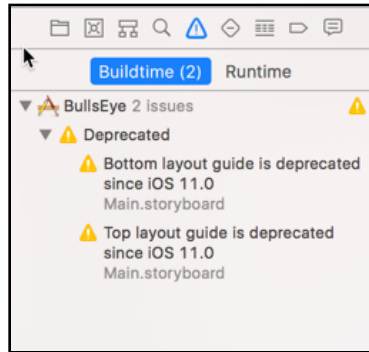
There should be four constraints listed there, one for each edge of the image.

► Run the app again on the iPhone 8 Simulator and also on the iPhone SE Simulator. In both cases, the background should display correctly now. (Of course, the other controls are still off-center, but we'll fix that soon.)

If you use the **View as:** panel to switch the storyboard back to the iPhone SE, the background should display correctly there too.

Compiler warnings

When you run the app after adding your first autolayout constraints, sometimes you might see some compiler warnings similar to this:



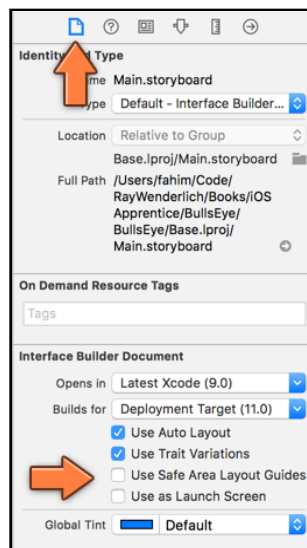
Auto Layout deprecation warnings

If this happens to you, that is because in iOS 11 there were some changes to how autolayout works and how constraints are set up. In previous versions of iOS, you had markers called *top layout guide* and *bottom layout guide* which defined the usable area of a screen. These guides were useful in setting your own views to stretch to the top edge (or the bottom of edge) of the screen without covering any on-screen elements provided by the OS such as navigation bars or tab bars.

However, in iOS 11, they introduced a new layout mechanism which was more flexible than the previously used top and bottom layout guides. These new layout guides are known as the *safe area layout guides*.

So how do you use these new safe area layout guides, you ask? Simple enough, you just have to enable them for your storyboard :]

Switch to your **storyboard**, select your view controller, and then on the right-hand pane, go to the **File Inspector**.



Enable Safe Area Layout Guides

Under the **Interface Builder Document** section, there should be a checkbox for **Use Safe Area Layout Guides** - check it. That's it, you are now using safe area layout guides in your storyboard and the compiler warnings should go away!

If you do not see the **Use Safe Area Layout Guides** checkbox, make sure that you have the view controller selected - that particular option appears only when you have a view controller selected.

The About screen

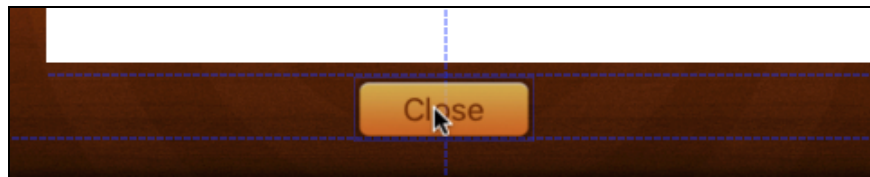
Let's repeat the background image fix for the About screen, too.

► 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 the bottom guide.

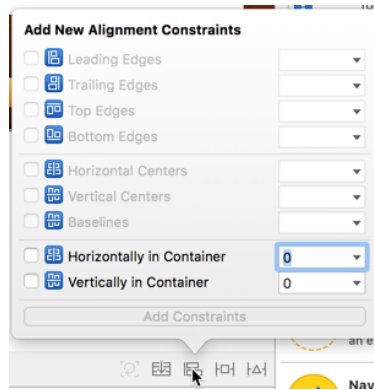
Interface Builder shows a handy guide, the dotted blue line, near the edges of the screen, which is useful for aligning objects by hand.



The dotted blue lines are guides that help position your UI elements

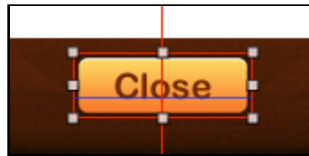
You want to create a centering constraint that keeps the Close button in the middle of the screen, regardless of how wide the screen is.

► 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 red bar to represent the constraint, and a red box around the button as well.



The Close button has red constraints

That's a problem: the bars are all supposed to be blue, not red. Red indicates that something is wrong with the constraints, usually that there aren't enough of them.

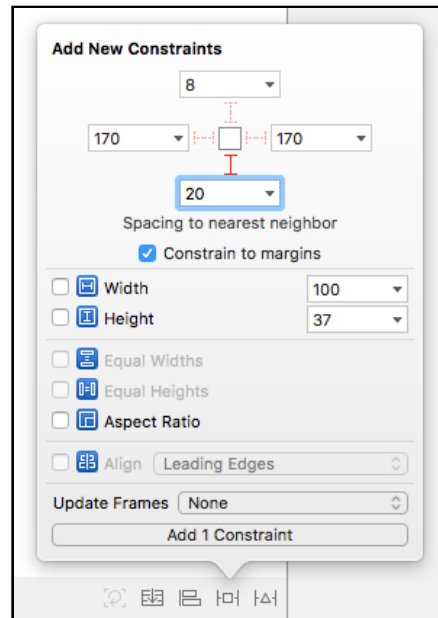
The thing to remember is this: for each view, there must always be enough constraints to define both its position and its size. The Close button already knows its size – you typed this into the Size inspector earlier – but for its position there is only a constraint for the X-coordinate (the alignment 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.

➤ 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.

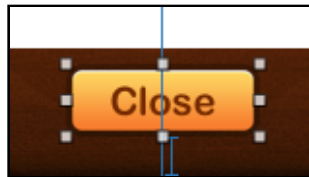
➤ In the **Add New Constraints** menu, in the **Spacing to nearest neighbor** section, set the bottom spacing to **20** and make sure that the I-beam above the text box is enabled.



The red I-beams decide the sides that are pinned down

► Click **Add 1 Constraint** to finish.

The red constraints will now turn blue, meaning that everything is OK:



The constraints on the Close button are valid

If at this point you don't see blue bars but orange ones, 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 the bars 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 now always be perfectly centered, regardless of the device screen size.

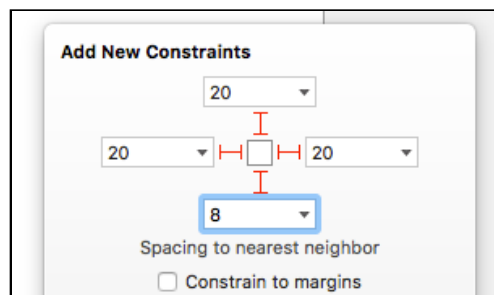
Note: What happens if you don't add any constraints to your views? In that case, Xcode will automatically add constraints when it builds the app. That is why you didn't need to bother with any of this before.

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. (Afterall, 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 is one thing left to fix in the About screen and that is the web view.

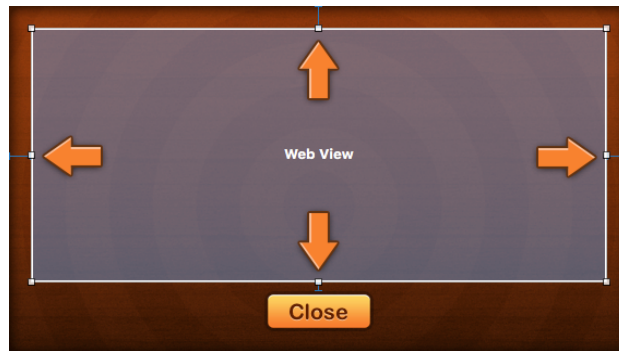
► Select the **Web View** and open the **Add New Constraints** menu. First, make sure **Constrain to margins** is unchecked. Then click all four I-beam icons so they become solid red and set their spacing to 20 points, except the bottom one which should be 8 points:



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

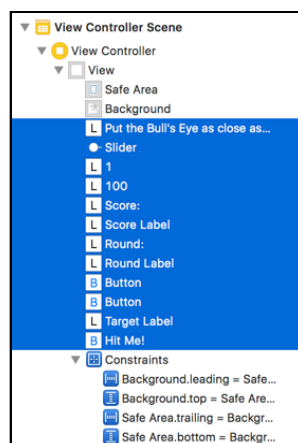
Three of these pin the web view to the main view, so that it always resizes along with it, and one connects it to the Close button. This is enough to determine the size and position of the web view in any scenario.

Fix the rest of the main scene

Back to the main game scene, which still needs some work.

The game looks a bit lopsided now on bigger screens. You will 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.

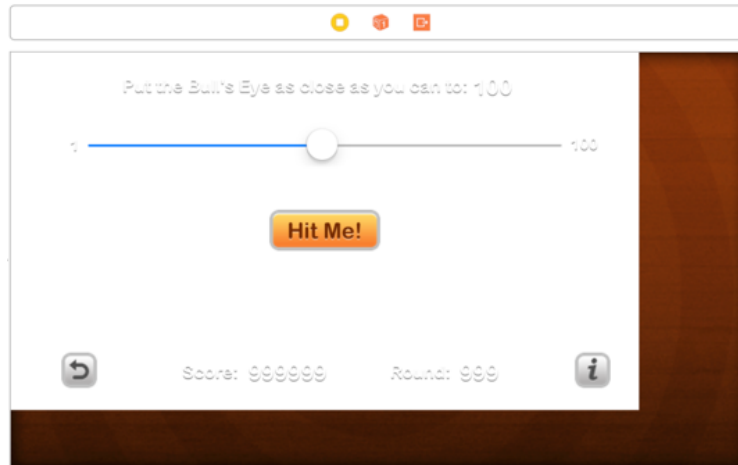
► Select all the labels, buttons, and the slider. You can hold down ⌘ and click them individually, but an easier method is to go to the **Document Outline**, click on the first view (for me that is the “Put the Bull’s Eye as close as you can to:” label), then hold down Shift and click on the last view (in my case the Hit Me! button):



Selecting the views from the Document Outline

You should have selected everything but the background image view.

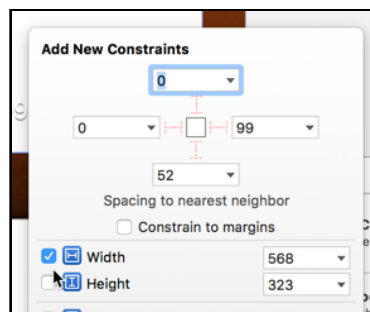
► From Xcode's menu bar, choose **Editor** → **Embed In** → **View**. This places the selected views inside a new container view:



The views are embedded in a new container view

This new view is completely white, which is not what you want eventually, but it does make 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** in order 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

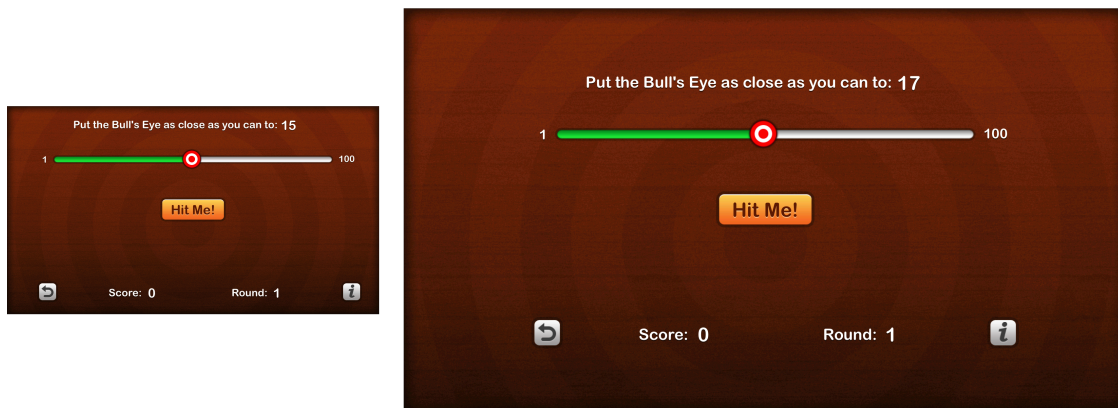
Interface Builder now draws several bars around the view that represent the Width and Height constraints that you just made, but they are red. Don't panic! It only means there are not enough constraints yet. No problem, you'll add the missing constraints next.

► With the container view still selected, open the **Align menu**. Check the **Horizontally in Container** and **Vertically in Container** options. Click **Add 2 Constraints**.

All 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) and then set up the necessary autolayout constraints for each control separately.

Are you up to the challenge of doing this on your own?

Crossfade

There's one final bit of knowledge I want to impart before calling the game complete - Core Animation. This technology makes it very easy to create really sweet animations, with just a few lines of code, in your apps. Adding subtle animations (with emphasis on subtle!) can make your app a delight to use.

You will add a simple crossfade after the Start Over button is pressed, so the transition back to round one won't seem so abrupt.

► In **ViewController.swift**, add the following line at the top, right below the other import:

```
import QuartzCore
```

Core Animation lives in its own framework, QuartzCore. With the `import` statement you tell the compiler that you want to use the objects from this framework.

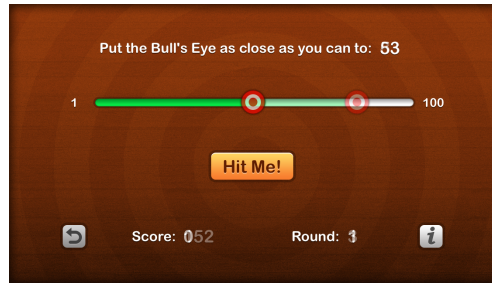
► Change `startNewGame()` to:

```
@IBAction func startNewGame() {  
    ...  
    startNewRound()  
    // Add the following lines  
    let transition = CATransition()  
    transition.type = kCATransitionFade  
    transition.duration = 1  
    transition.timingFunction = CAMediaTimingFunction(name:  
                                                kCAMediaTimingFunctionEaseOut)  
    view.layer.add(transition, forKey: nil)  
}
```

Everything after the comment telling you to add the following lines, all the `CATransition` stuff, is new.

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()` – reset the slider to center position and reset the values of the labels.

► Run the app and move the slider so that it is no longer in the center. Press the Start Over button and you should see a subtle crossfade animation.

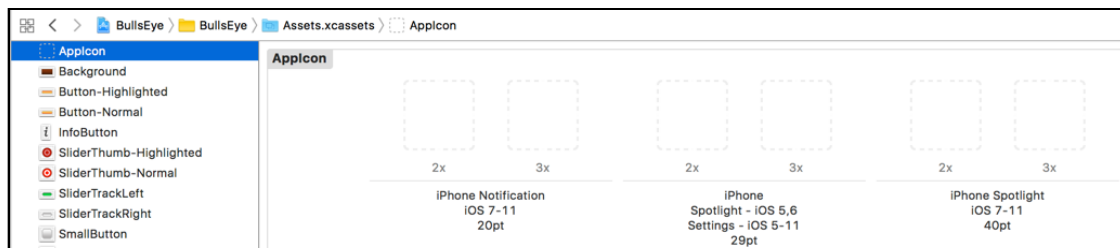


The screen crossfades between the old and new states

The 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!

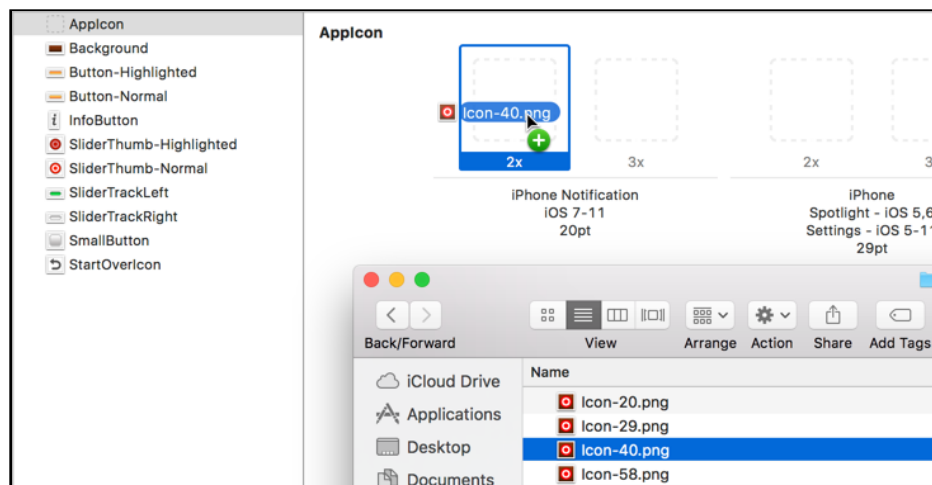
- Open the asset catalog (**Assets.xcassets**) and select **AppIcon**:



The AppIcon group in the asset catalog

This has ten groups for the different types of icons the app needs.

- In Finder, open the **Icon** folder from the resources. Drag the **Icon-40.png** file into the first slot, **iPhone Notification 20pt**:



Dragging the icon into the asset catalog

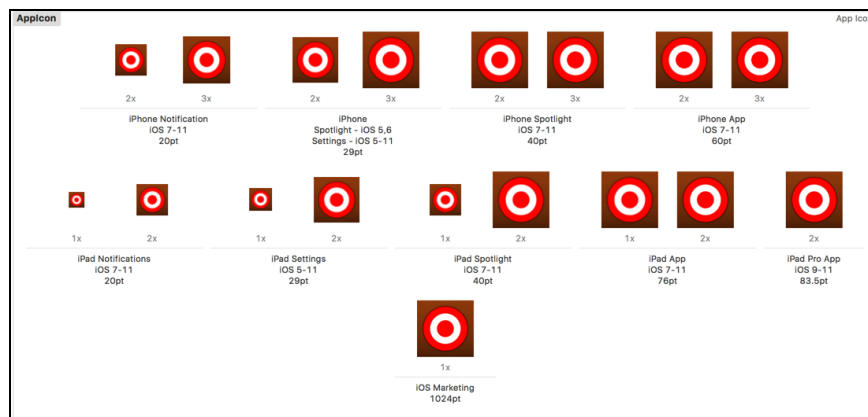
You may be wondering why you're dragging the **Icon-40.png** file and not the **Icon-20.png** into the slot for 20pt. Notice that this slot says **2x**, which means it's for Retina devices and on Retina screens one point counts as two pixels. So, 20pt = 40px. And the 40 in the icon name is for the size of the icon in pixels. Makes sense?

- 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 **iPhone Spotlight & Settings 29pt**, 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 **iPhone Spotlight 40pt**, drag the **Icon-80.png** file into the 2x slot and **Icon-120.png** into the 3x slot.
- For **iPhone App 60pt**, 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!

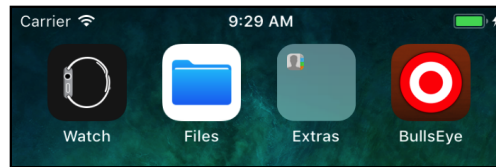
The other AppIcon groups are mostly for the iPad.

- Drag the specific icons (based on size) into the proper slots for iPad. Notice that the iPad icons need to be supplied in 1x as well as 2x sizes (but not 3x). You may need to do some mental arithmetic here to figure out which icon goes into which slot!



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).



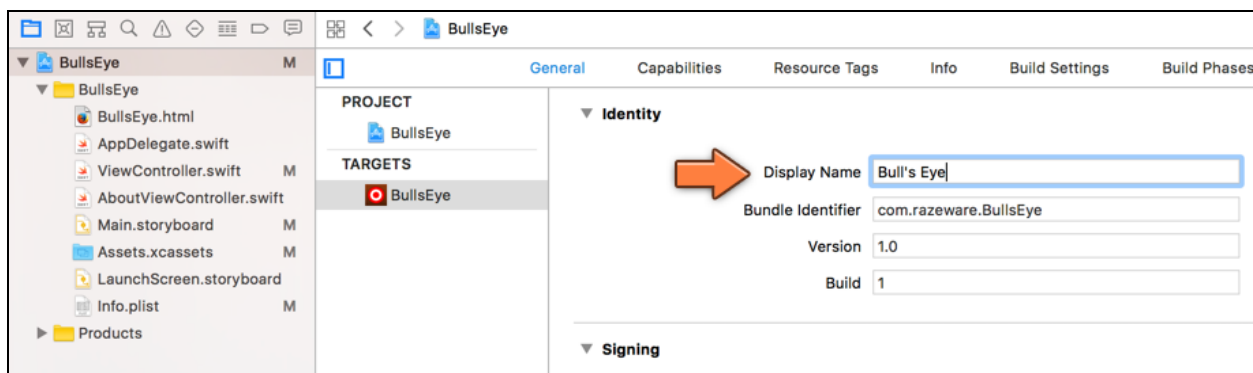
The icon on the Simulator's springboard

Display name

One last thing. You named the project **BullsEye** and that is the name that shows up under the icon. However, I'd prefer to spell it “**Bull's Eye**”.

There is only limited space under the icon and for apps with longer names you have to get creative to make the name fit. For this game, however, there is enough room to add the space and the apostrophe.

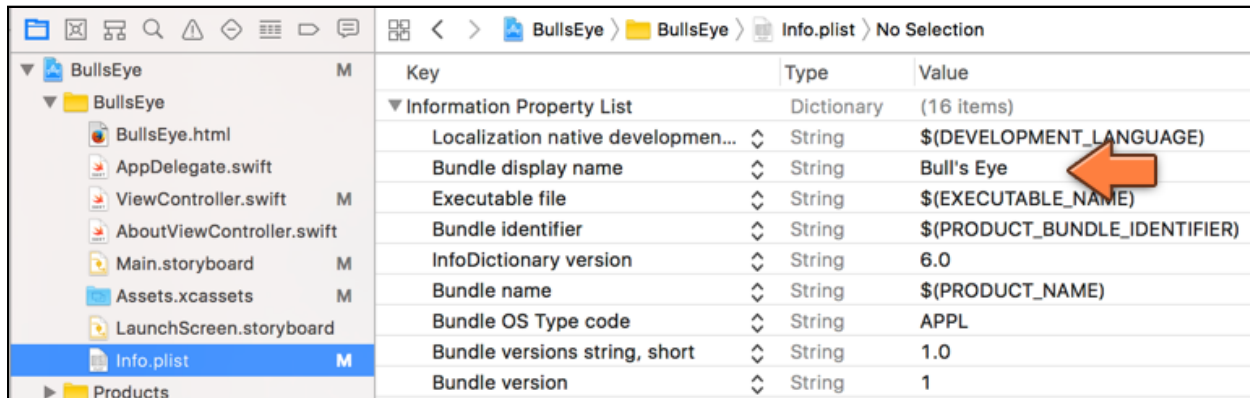
► Go to the **Project Settings** screen. The very first option is **Display Name**. Change this to **Bull's Eye**.



Changing the display name of the app

Like many of the project's settings you can also find the display name in the app's Info.plist file. Let's have 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

Awesome, that completes your very first app!

You can find the project files for the finished app under **08 - The Final App** in the Source Code folder.

Run on device

So far, you've run the app on the Simulator. That's nice and all but probably not why you're learning iOS development. You want to make apps that run on real iPhones and iPads! There's hardly a thing more exciting than running an app that you made on your own phone. And, of course, to show 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, I spend most of my time with the Simulator and only test the app on my iPhone every so often.

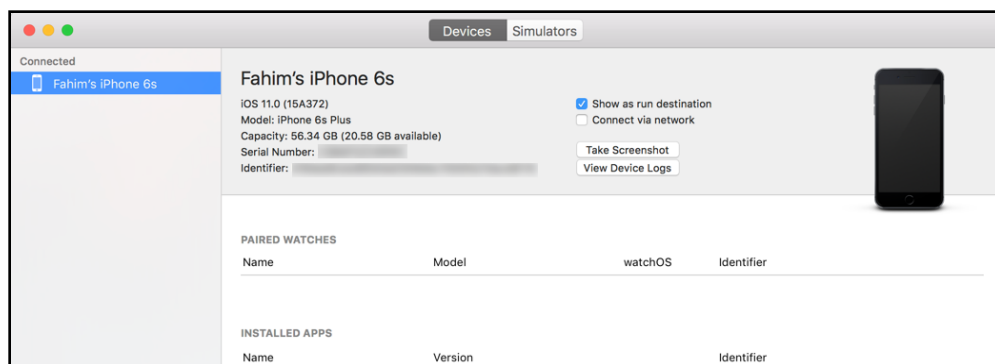
However, you do need to run your creations on a real device in order to test them properly. There are some things the Simulator simply cannot do. If your app needs the iPhone's accelerometer, for example, you have no choice but to test that functionality on an actual device. Don't sit there and shake your Mac!

Until a few years back, you needed a paid Developer Program account to run apps on your iPhone. Since Xcode 7, however, you can do it for free. All you need is an Apple ID. And the latest Xcode makes it easier than ever before.

Configure 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.

Mine looks like this (I'm using an iPhone 6s):



The Devices and Simulators window

On the left is a list of devices that are currently connected to my Mac and which can be used for development.

- Click your device name to select it.

If this is the first time you're using the device with Xcode, the Devices window will say something like, "iPhone is not paired with your computer." To pair the device with Xcode, you need to unlock the device first (hold the home button). After unlocking, 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 (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.

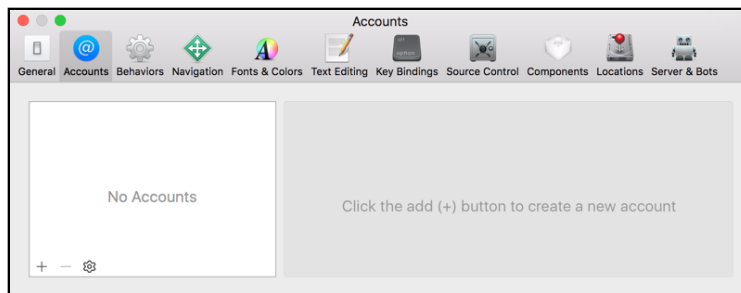
Also, note the checkbox which says **Connect via network?** That checkbox (gasp!) allows you to run and debug code on your iPhone over WiFi! Yes, that's new in Xcode 9. (I still prefer to do my debugging with my phone connected via USB cable since the last time I checked, the over network debugging was very slow. But your mileage may vary - so give it a try...)

Cool, that is the device sorted.

Add your developer account to Xcode

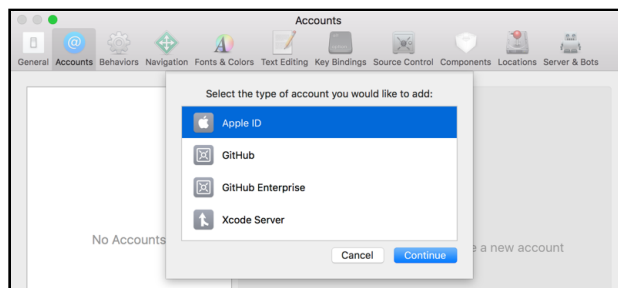
The next step is setting up your Apple ID with Xcode. It's OK to use the same Apple ID that you're already using with iTunes and your iPhone, but if you run a business, you might want to create a new Apple ID to keep things separate. Of course, if you've already registered for a paid Developer Program account, you should use that Apple ID.

➤ Open the **Accounts** pane in the Xcode Preferences window:



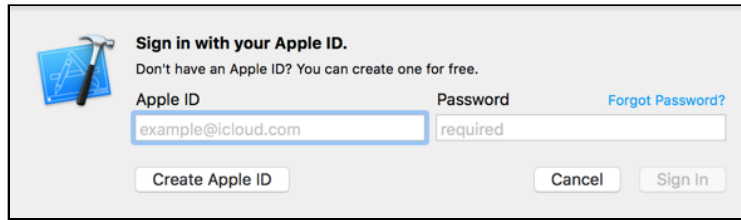
The Accounts preferences

➤ Click the + button at the bottom and select **Add Apple ID** from the list of options.



Xcode Account Type selection

Xcode will ask for your Apple ID:



Adding your Apple ID to Xcode

► Type your Apple ID username and password and click **Sign In**.

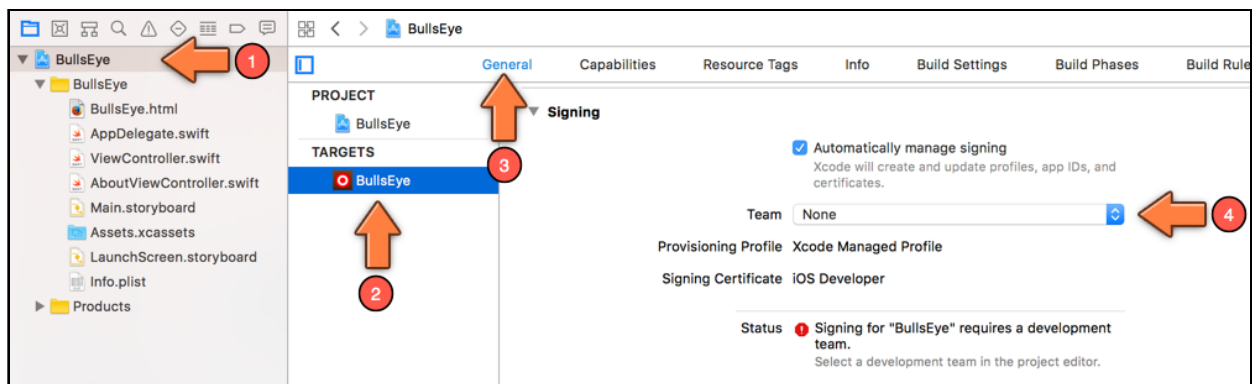
Xcode verifies your account details and adds it to the stored list of accounts.

Note: It's possible that Xcode is unable to use the Apple ID your 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** section.



The Signing options in the Project Settings screen

In order to allow Xcode to put an app on your iPhone, 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.

Apps that you want to submit to the App Store must be signed with another certificate, the **Distribution Certificate**. To use the distribution certificate you must be a member of the paid Developer Program, but using the development certificate is free.

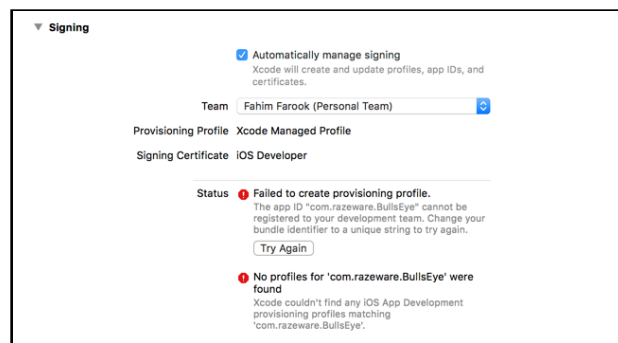
In addition to a valid certificate, you also need a **Provisioning Profile** for each app you make. Xcode uses this profile to sign the app for use on your particular device (or devices). The specifics don't really matter, just know that you need a provisioning profile or the app won't go on your device.

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.

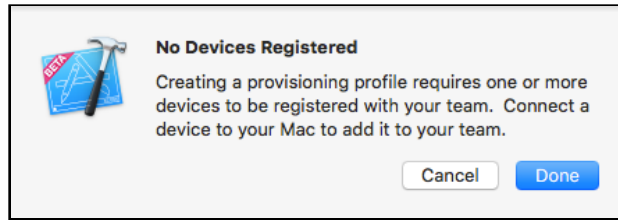
You could get some signing errors like these:



Signing/team set up errors

The app's Bundle Identifier – or App ID as it's called here – must be unique. If another app is already using that identifier, then you cannot use it anymore. That's why you're supposed to 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):



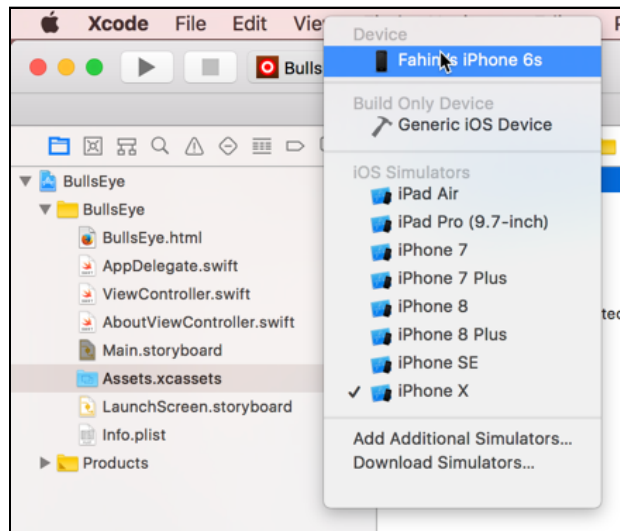
No devices registered

Xcode must know about the device that you're going to run the app on. That's why I asked you to connect your device first. Double-check that your iPhone or iPad is still connected to your Mac and that it is listed in the Devices window.

Run on device

If everything goes smoothly, go back to Xcode's main window and click on the box in the toolbar to change where you will run the app. The name of your device should be in that list somewhere.

On my system it looks like this:



Setting the active device

You're all set and ready to go!

► Press **Run** to launch the app.

At this point you may get a popup 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...

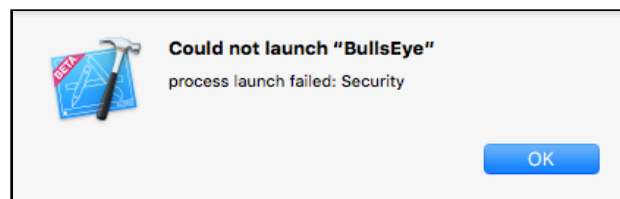
There are a few things that can go wrong when you try to put the app on your device, especially if you've never done this before, so don't panic if you run into problems.

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:



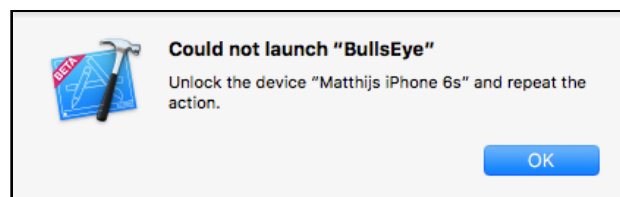
Quick, call security!

On the device itself there will be a popup 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 (hold the home button or type in the 4-digit passcode) and press 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.

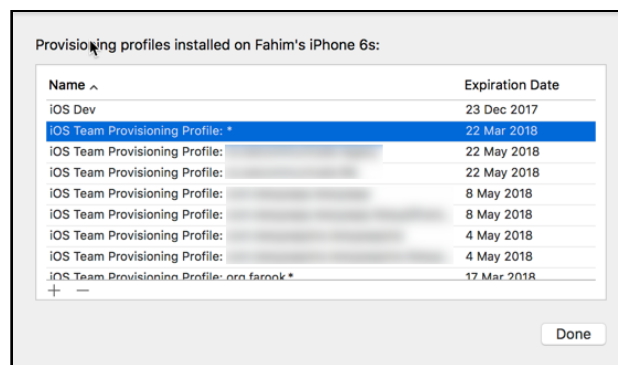
This brings up another panel, listing your signing certificates:



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**:



The provisioning profiles on your device

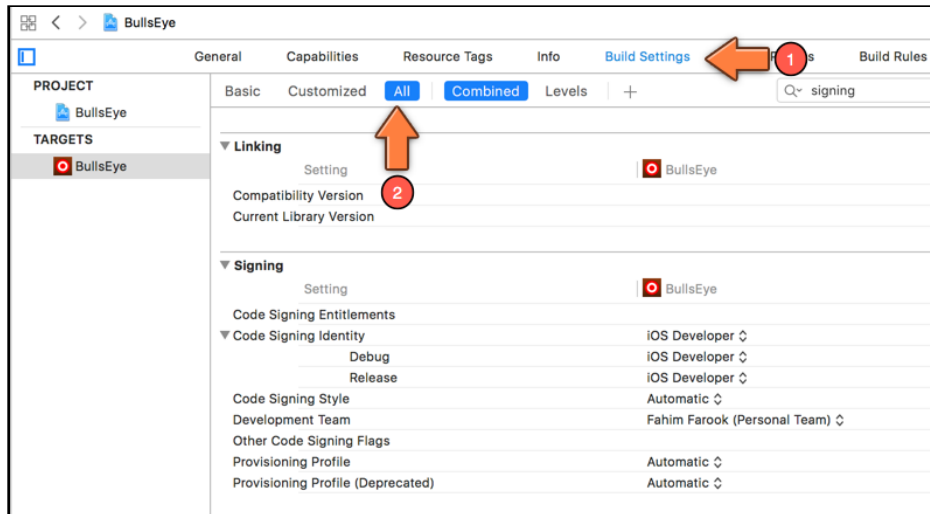
The “iOS Team Provisioning Profile: *” is the thing that allows you to run the app on your device. (By the way, they call it the “team” profile because often there is more than one developer working on an app and they can all share the same profile.)

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 has 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 least you should have gotten some insight into what it takes to develop an app.

I don't expect you to understand exactly everything that you did, especially not the parts that involved writing Swift code. 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 chapter you can find the complete source code for the *Bull's Eye* app. If you're still unclear about some of what you did, it might be a good idea to look at this cleaned up source code.

If you're interested in how I made the graphics, then 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 to grips with more code, let's switch gears and start a deep dive into the Swift programming language itself! :]

Section II: Programming in Swift

The second section of this book corresponds to the **Programming in Swift** sections of the course. You'll learn fundamental Swift programming concepts, like variables, loops, collections, structures, classes, optionals, closures, and more. By the time you're done, you'll be comfortable with the fundamental concepts of programming in Swift, and you'll be ready to switch back to iOS development and make your second app.

Chapter 8: Expressions, Variables, and Constants

Chapter 9: Types and Operations

Chapter 10: Basic Control Flow

Chapter 11: Advanced Control Flow

Chapter 12: Functions

Chapter 13: Optionals

Chapter 14: Arrays, Dictionaries, and Sets

Chapter 15: Collection Iteration with Closures

Chapter 16: Strings

Chapter 17: Structures

Chapter 18: Properties

Chapter 19: Methods

Chapter 20: Classes

Chapter 21: Advanced Classes

Chapter 8: Expressions, Variables & Constants

By Matt Galloway

So far in this book, you've been coding your first iOS app, and learning Swift along the way. But to be a professional Swift developer, you really need to have a solid understanding of the Swift language itself.

In the next few sections, you're going to switch gears by taking a deep dive into the Swift language. This will give you a solid foundation that you can use to continue building apps.

You'll start by learning how code works. Then you'll learn about the tools you'll be using to write Swift code.

Then, you'll learn some Swift basics such as code comments, arithmetic operations, constants and variables. These are some of the fundamental building blocks of any language, and Swift is no different.

First of all, you'll cover the basic workings of computers, because it really pays to have a grounding before you get into more complicated aspects of programming.

How a computer works

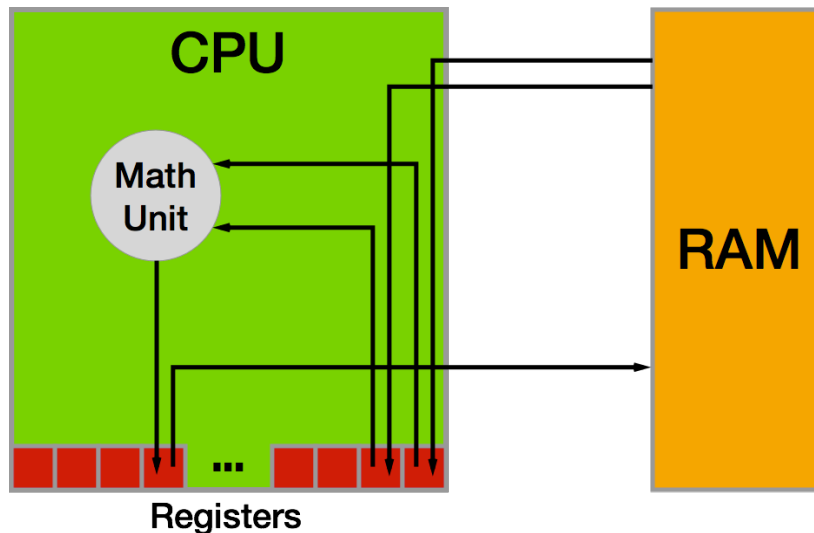
You may not believe me when I say it, but a computer is not very smart on its own. The power of computers is all derived from how they're programmed by people like you and me. If you want to successfully harness the power of a computer — and I assume you do, if you're reading this book — it's important to understand how computers work.

It may also surprise you to learn that computers themselves are rather simple machines. At the heart of a computer is a **Central Processing Unit (CPU)**. This is essentially a math machine. It performs addition, subtraction, and other arithmetical

operations on numbers. Everything you see when you operate your computer is all built upon a CPU crunching numbers many millions of times per second. Isn't it amazing what can come from just numbers?

The CPU stores the numbers it acts upon in small memory units called **registers**. The CPU is able to read numbers into registers from the computer's main memory, known as **Random Access Memory (RAM)**. It's also able to write the number stored in a register back into RAM. This allows the CPU to work with large amounts of data that wouldn't all fit in the bank of registers.

Here is a diagram of how this works:



As the CPU pulls values from RAM into its registers, it uses those values in its math unit and stores the results back in another register.

Each time the CPU makes an addition, a subtraction, a read from RAM or a write to RAM, it's executing a single **instruction**. Each computer program is usually made up of thousands to millions of instructions. A complex computer program such as your operating system, macOS (yes, that's a computer program too!), may have many millions of instructions in total.

It's entirely possible to write individual instructions to tell a computer what to do, but for all but the simplest programs, it would be immensely time-consuming and tedious. This is because most computer programs aim to do much more than simple math — computer programs let you surf the internet, manipulate images, and allow you to chat with your friends.

Instead of writing individual instructions, you write **code** in a specific **programming language**, which in your case will be Swift. This code is put through a computer

program called a **compiler**, which converts the code into instructions the CPU knows how to execute. Each line of code you write will turn into many instructions — some lines could end up being tens of instructions!

Representing numbers

As you know by now, numbers are a computer's bread and butter, the fundamental basis of everything it does. Whatever information you send to the compiler will eventually become a number. For example, each character within a block of text is represented by a number. You'll learn more about this in Chapter 3, which delves into types including **strings**, the computer term for a block of text.

Images are no exception. In a computer, each image is also represented by a series of numbers. An image is split into many thousands, or even millions, of picture elements called **pixels**, where each pixel is a solid color. If you look closely at your computer screen, you may be able to make out these blocks. That is unless you have a particularly high-resolution display where the pixels are incredibly small! Each of these solid color pixels is usually represented by three numbers: one for the amount of red, one for the amount of green and one for the amount of blue. For example, an entirely red pixel would be 100% red, 0% green and 0% blue.

The numbers the CPU works with are notably different from those you are used to. When you deal with numbers in day-to-day life, you work with them in **base 10**, otherwise known as the **decimal** system. Having used this numerical system for so long, you intuitively understand how it works. So that you can appreciate the CPU's point of view, consider how base 10 works.

The decimal or base 10 number **423** contains **three units**, **two tens** and **four hundreds**:

1000	100	10	1
0	4	2	3

In the base 10 system, each digit of a number can have a value of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9, giving a total of 10 possible values for each digit. Yep, that's why it's called base 10!

But the true value of each digit depends on its position within the number. Moving from right to left, each digit gets multiplied by an increasing power of 10. So the multiplier for the far-right position is 10 to the power of 0, which is 1. Moving to the left, the next multiplier is 10 to the power of 1, which is 10. Moving again to the left, the next multiplier is 10 to the power of 2, which is 100. And so on.

This means each digit has a value ten times that of the digit to its right. The number **423** is equal to the following:

$$(0 * 1000) + (4 * 100) + (2 * 10) + (3 * 1) = 423$$

Binary numbers

Because you've been trained to operate in base 10, you don't have to think about how to read most numbers — it feels quite natural. But to a computer, base 10 is way too complicated! Computers are simple-minded, remember? They like to work with base 2.

Base 2 is often called **binary**, which you've likely heard of before. It follows that base 2 has only two options for each digit: 0 or 1.

Almost all modern computers use binary because at the physical level, it's easiest to handle only two options for each digit. In digital electronic circuitry, which is mostly what comprises a computer, the presence of an electrical voltage is 1 and the absence is 0 — that's base 2!

Note: There have been computers both real and imagined that use the ternary numeral system, which has three possible values instead of two. Computer scientists, engineers and dedicated hackers continue to explore the possibilities of a base-3 computer. See https://en.wikipedia.org/wiki/Ternary_computer and <http://hackaday.com/tag/ternary-computer/>.

Here's a representation of the base 2 number 1101:

8	4	2	1
1	1	0	1

In the base 10 number system, the place values increase by a factor of 10: 1, 10, 100, 1000, etc. In base 2, they increase by a factor of 2: 1, 2, 4, 8, 16, etc. The general rule is to multiply each digit by an increasing power of the base number — in this case, powers of 2 — moving from right to left.

So the far-right digit represents $(1 * 2^0)$, which is $(1 * 1)$, which is 1. The next digit to the left represents $(0 * 2^1)$, which is $(0 * 2)$, which is 0. In the illustration above, you can see the powers of 2 on top of the blocks.

Put another way, every power of 2 either is (1) or isn't (0) present as a component of a binary number. The decimal version of a binary number is the sum of all the powers of 2 that make up that number. So the binary number 1101 is equal to:

$$(1 * 8) + (1 * 4) + (0 * 2) + (1 * 1) = 13$$

And if you wanted to convert the base 10 number 423 into binary, you would simply need to break down 423 into its component powers of 2. You would wind up with the following:

$$(1 * 256) + (1 * 128) + (0 * 64) + (1 * 32) + (0 * 16) + (0 * 8) + (1 * 4) + (1 * 2) + (1 * 1) = 423$$

As you can see by scanning the binary digits in the above equation, the resulting binary number is 110100111. You can prove to yourself that this is equal to 423 by doing the math!

The computer term given to each digit of a binary number is a **bit** (a contraction of “binary digit”). Eight bits make up a **byte**. Four bits is called a **nibble**, a play on words that shows even old school computer scientists had a sense of humor.

A computer's limited memory means it can normally deal with numbers up to a certain length. Each register, for example, is usually 32 or 64 bits in length, which is why we speak of 32-bit and 64-bit CPUs.

Therefore, a 32-bit CPU can handle a maximum base-number of 4,294,967,295, which is the base 2 number 11111111111111111111111111111111. That is 32 ones—count them!

It's possible for a computer to handle numbers that are larger than the CPU maximum, but the calculations have to be split up and managed in a special and longer way, much like the long multiplication you performed in school.

Hexadecimal numbers

As you can imagine, working with binary numbers can become quite tedious, because it can take a long time to write or type them. For this reason, in computer programming, we often use another number format known as **hexadecimal**, or **hex** for short. This is **base 16**.

Of course, there aren't 16 distinct numbers to use for digits; there are only 10. To supplement these, we use the first six letters, **a** through **f**.

They are equivalent to decimal numbers like so:

- a = 10

- `b = 11`
- `c = 12`
- `d = 13`
- `e = 14`
- `f = 15`

Here's a base 16 example using the same format as before:

4096	256	16	1
c	0	d	e

Notice first that you can make hexadecimal numbers look like words. That means you can have a little bit of fun. :]

Now the values of each digit refer to powers of 16. In the same way as before, you can convert this number to decimal like so:

$$(12 * 4096) + (0 * 256) + (13 * 16) + (14 * 1) = 49374$$

You translate the letters to their decimal equivalents and then perform the usual calculations.

But why bother with this?

Hexadecimal is important because each hexadecimal digit can represent precisely four binary digits. The binary number 1111 is equivalent to hexadecimal `f`. It follows that you can simply concatenate the binary digits representing each hexadecimal digit, creating a hexadecimal number that is shorter than its binary or decimal equivalents.

For example, consider the number `c0de` from above:

```
c = 1100
0 = 0000
d = 1101
e = 1110

c0de = 1100 0000 1101 1110
```

This turns out to be rather helpful, given how computers use long 32-bit or 64-bit binary numbers. Recall that the longest 32-bit number in decimal is 4,294,967,295. In hexadecimal, it is `ffffffff`. That's much more compact and clear.

How code works

Computers have a lot of constraints, and by themselves, they can only do a small number of things. The power that the computer programmer adds, through coding, is putting these small things together, in the right order, to produce something much bigger.

Coding is much like writing a recipe. You assemble ingredients (the data) and give the computer a step-by-step recipe for how to use them.

Here's an example:

```
Step 1. Load photo from hard drive.  
Step 2. Resize photo to 400 pixels wide by 300 pixels high.  
Step 3. Apply sepia filter to photo.  
Step 4. Print photo.
```

This is what's known as **pseudo-code**. It isn't written in a valid computer programming language, but it represents the **algorithm** that you want to use. In this case, the algorithm takes a photo, resizes it, applies a filter and then prints it. It's a relatively straightforward algorithm, but it's an algorithm nonetheless!

Swift code is just like this: a step-by-step list of instructions for the computer. These instructions will get more complex as you read through this book, but the principle is the same: You are simply telling the computer what to do, one step at a time.

Each programming language is a high-level, pre-defined way of expressing these steps. The compiler knows how to interpret the code you write and convert it into instructions that the CPU can execute.

There are many different programming languages, each with its own advantages and disadvantages. Swift is an extremely modern language. It incorporates the strengths of many other languages while ironing out some of their weaknesses. In years to come, programmers will look back on Swift as being old and crusty, too. But for now, it's an extremely exciting language because it is quickly evolving.

This has been a brief tour of computer hardware, number representation and code, and how they all work together to create a modern program. That was a lot to cover in one section! Now it's time to learn about the tools you'll use to write in Swift as you follow along with this book.

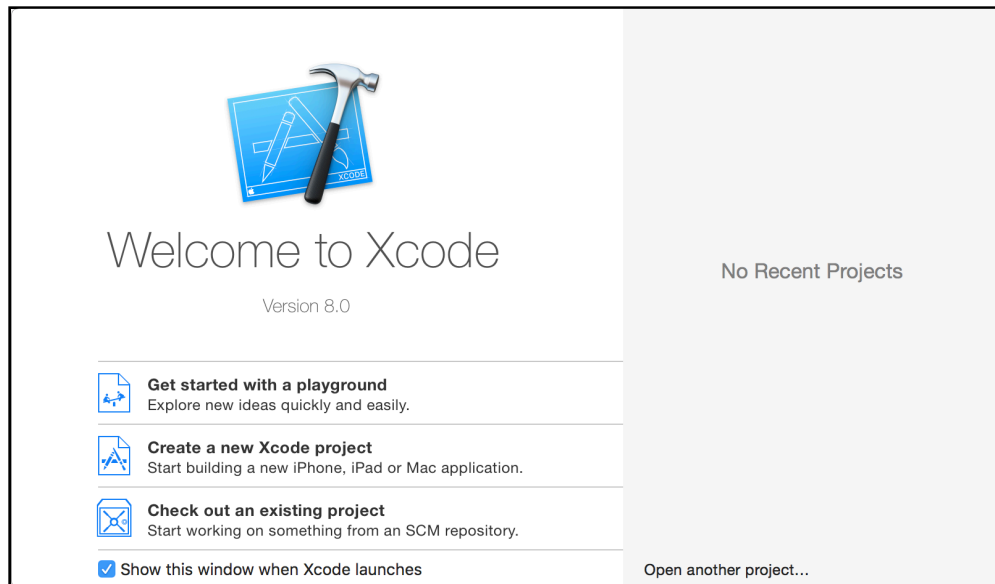
Playgrounds

The set of tools you use to write software is often referred to as the **toolchain**. The part of the toolchain into which you write your code is known as the **Integrated Development Environment (IDE)**. The most commonly used IDE for Swift is called Xcode, and that's what you'll be using.

Xcode includes a handy feature called a **playground**, which allows you to quickly write and test code without needing to build a complete app. You'll use playgrounds throughout the book to practice coding, so it's important to understand how they work. That's what you'll learn during the rest of this chapter.

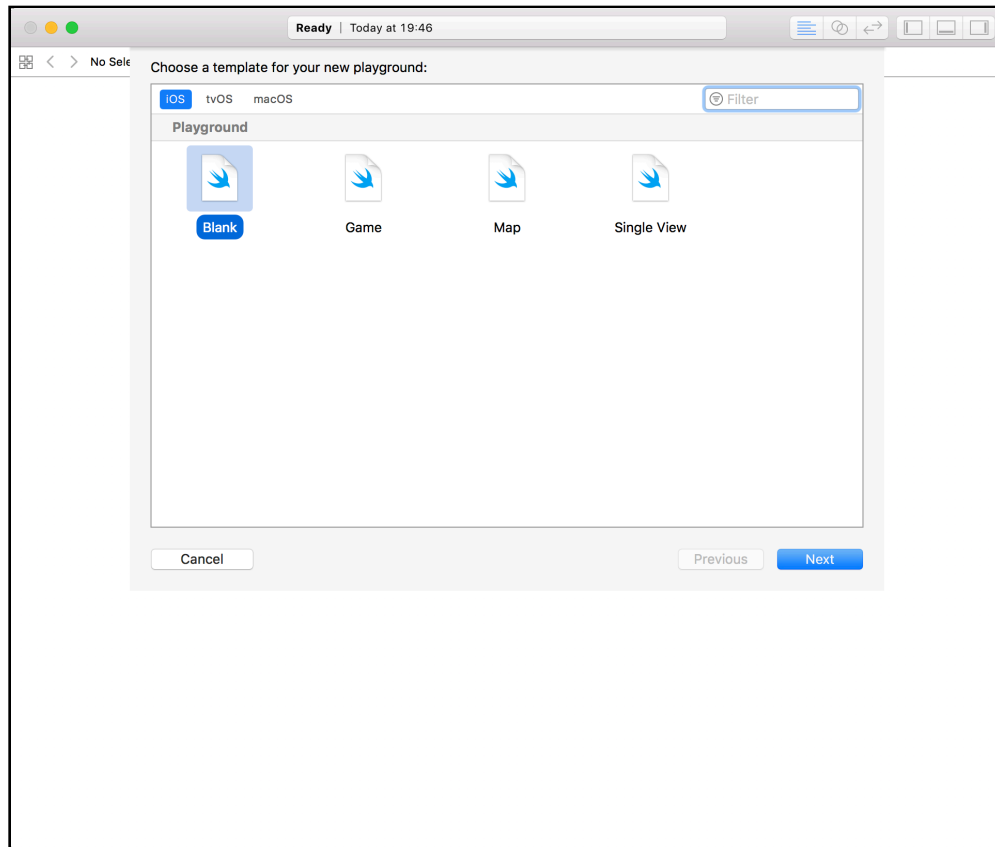
Creating a playground

When you open Xcode, it will greet you with the following welcome screen:



If you don't see this screen, it's most likely because the "Show this window when Xcode launches" option was unchecked. You can also open the screen by pressing **Command-Shift-1** or clicking **Window\Welcome to Xcode** from the menu bar.

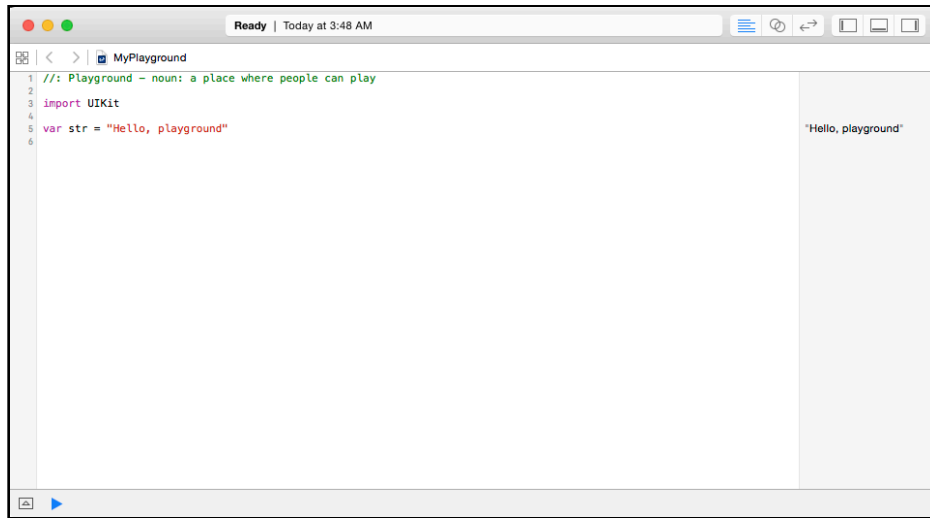
From the welcome screen, you can jump quickly into a playground by clicking on **Get started with a playground**. Click on that now and Xcode will present you with a choice of templates.



The platform you choose simply defines which version of the template Xcode will use to create the playground. Currently, your options are **iOS**, **macOS** or **tvOS**. Each platform comes with its own environment set up and ready for you to begin playing around with code. For the purposes of this book, choose whichever platform you wish. You won't be writing any platform-specific code; instead, you'll be learning the core principles of the Swift language.

Select the **Blank** template and click **Next**. Xcode will now ask you to name the playground and select a location to save it. The name is merely cosmetic and for your own use; when you create your playgrounds, feel free to choose names that will help you remember what they're about. For example, while you're working through Chapter 2, you may want to name your playground **Chapter2**.

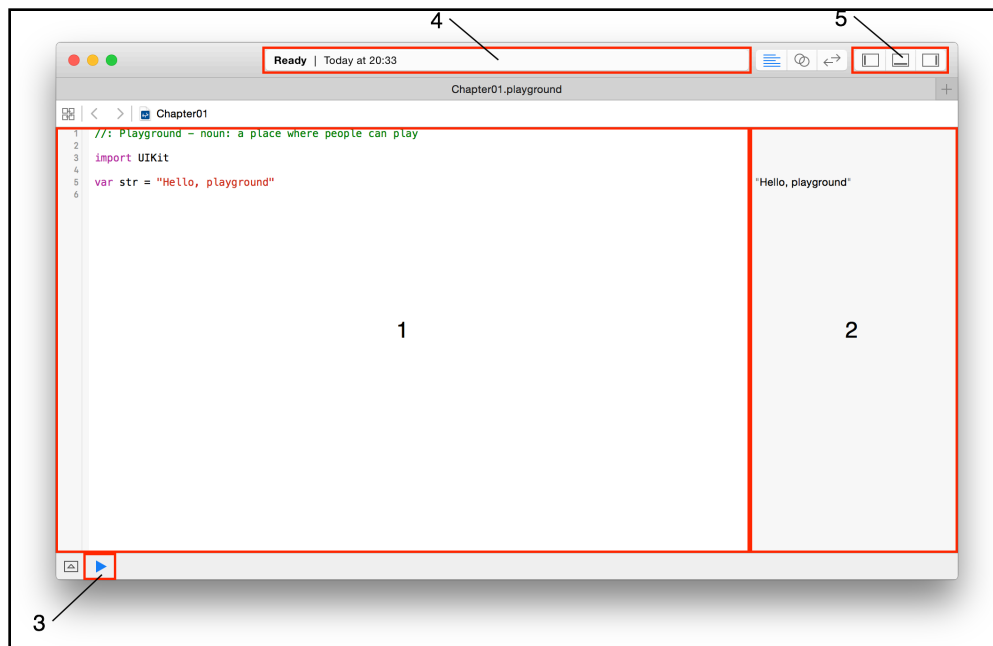
Click **Create** to create and save the playground. Xcode then presents you with the playground, like so:



Even blank playgrounds don't start entirely empty but have some basic starter code to get you going. Don't worry — you'll soon learn what this code means.

Playgrounds overview

At first glance, a playground may look like a rather fancy text editor. Well, here's some news for you: It is essentially just that!



The above screenshot highlights the first and most important things to know about:

1. **Source editor:** This is the area in which you'll write your Swift code. It's much like a text editor such as Notepad or TextEdit. You'll notice the use of what's known as a monospaced font, meaning all characters are the same width. This makes the code much easier to read and format.
2. **Results sidebar:** This area shows the results of your code. You'll learn more about how code is executed as you read through the book. The results sidebar will be the main place you'll look to confirm your code is working as expected.
3. **Execution control:** Playgrounds execute automatically by default, meaning you can write code and immediately see the output. This control allows you to execute the playground again. Holding down the button allows you to switch between automatic execution and manual execution modes.
4. **Activity viewer:** This shows the status of the playground. In the screenshot, it shows that the playground has finished executing and is ready to handle more code in the source editor. When the playground is executing, this viewer will indicate this with a spinner.
5. **Panel controls:** These toggle switches show and hide three panels, one that appears on the left, one on the bottom and one on the right. The panels each display extra information that you may need to access from time to time. You'll usually keep them hidden, as they are in the screenshot. You'll learn more about each of these panels as you move through the book.

Playgrounds execute the code in the source editor from top to bottom. Every time you change the code, the playground will re-execute everything. You can also force a re-execution by clicking **Editor\Execute Playground**. Alternatively, you can use the execution control.

You can turn on line numbers on the left side of the source editor by clicking **Xcode\Preferences...\Text Editing\Line Numbers**. Line numbers can be very useful when you want to refer to parts of your code.

Once the playground execution is finished, Xcode updates the results sidebar to show the results of the corresponding line in the source editor. You'll see how to interpret the results of your code as you work through the examples in this book.

Getting started with Swift

Now that you know how computers work and know what this "playground" thing is, it's time to start writing some Swift!

You may wish to follow along with your own playground. Simply create one and type in the code as you go!

First up is something that helps you organize your code. Read on!

Code comments

The Swift compiler generates executable code from your source code. To accomplish this, it uses a detailed set of rules you will learn about in this book. Sometimes these details can obscure the big picture of *why* you wrote your code a certain way or even what problem you are solving. To prevent this, it's good to document what you wrote so that the next human who passes by will be able to make sense of your work. That next human, after all, may be a future you.

Swift, like most other programming languages, allows you to document your code through the use of what are called **comments**. These allow you to write any text directly along side your code which is ignored by the compiler.

The first way to write a comment is like so:

```
// This is a comment. It is not executed.
```

This is a **single line comment**.

You could stack these up like so to allow you to write paragraphs:

```
// This is also a comment.  
// Over multiple lines.
```

However, there is a better way to write comments which span multiple lines. Like so:

```
/* This is also a comment.  
   Over many..  
   many...  
   many lines. */
```

This is a **multi-line comment**. The start is denoted by `/*` and the end is denoted by `*/`. Simple!

Swift also allows you to nest comments, like so:

```
/* This is a comment.  
  
/* And inside it  
is  
another comment.  
*/  
  
Back to the first.  
*/
```

This might not seem particularly interesting, but it may be if you have seen other programming languages. Many do not allow you to nest comments like this as when it sees the first `*/` it thinks you are closing the first comment.

You should use code comments where necessary to document your code, explain your reasoning, or simply to leave jokes for your colleagues. :]

Printing out

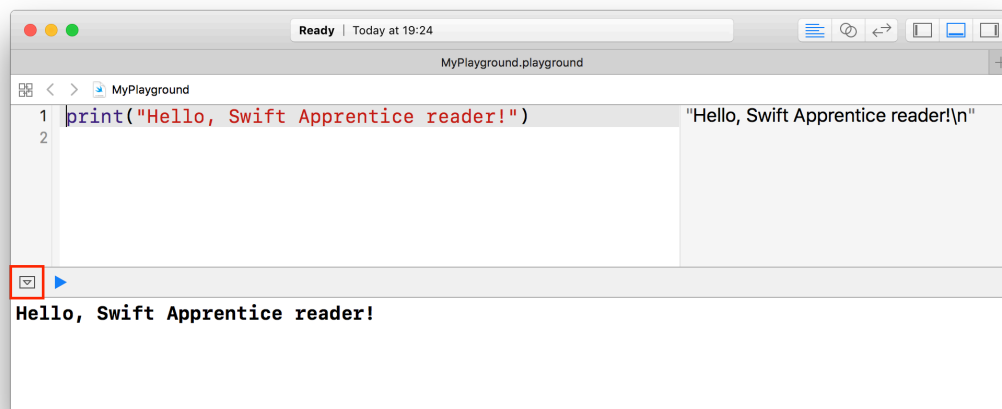
It's also useful to see the results of what your code is doing. In Swift, you can achieve this through the use of the `print` command.

`print` will output whatever you want to the **debug area** (sometimes referred to as the console).

For example, consider the following code:

```
print("Hello, Swift Apprentice reader!")
```

This will output a nice message to the debug area, like so:



You can hide or show the debug area using the button highlighted with the red box in the picture above. You can also click **View\Debug Area\Show Debug Area** to do the same thing.

Arithmetic operations

When you take one or more pieces of data and turn them into another piece of data, this is known as an **operation**.

The simplest way to understand operations is to think about arithmetic. The addition operation takes two numbers and converts them into the sum of the two numbers. The subtraction operation takes two numbers and converts them into the difference of the two numbers.

You'll find simple arithmetic all over your apps; from tallying the number of “likes” on a post, to calculating the correct size and position of a button or a window, numbers are indeed everywhere!

In this section, you'll learn about the various arithmetic operations that Swift has to offer by considering how they apply to numbers. In later chapters, you see operations for types other than numbers.

Simple operations

All operations in Swift use a symbol known as the **operator** to denote the type of operation they perform. Consider the four arithmetic operations you learned in your early school days: addition, subtraction, multiplication and division. For these simple operations, Swift uses the following operators:

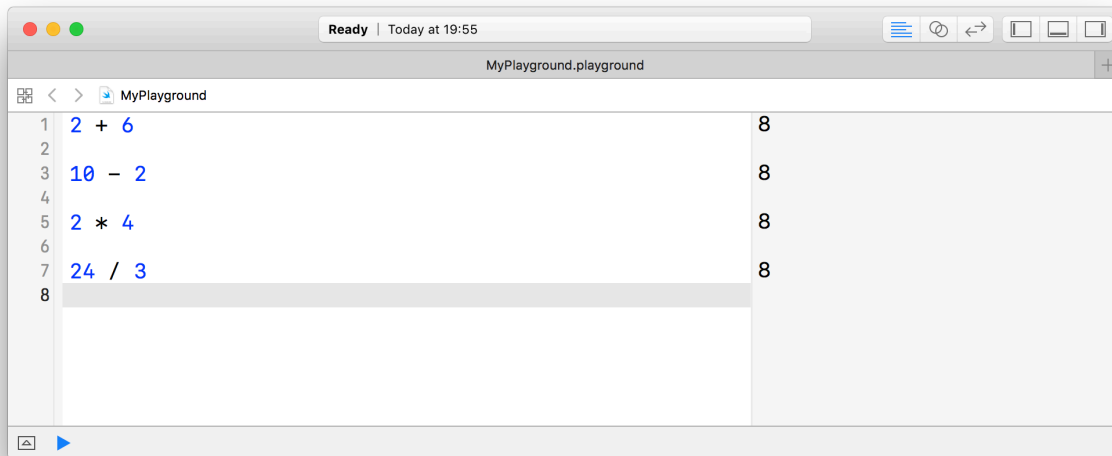
- Add: +
- Subtract: −
- Multiply: *
- Divide: /

These operators are used like so:

```
2 + 6
10 − 2
2 * 4
24 / 3
```

Each of these lines is what is known as an **expression**. An expression has a value. In these cases, all four expressions have the same value: 8. You write the code to perform these arithmetic operations much as you would write it if you were using pen and paper.

In your playground, you can see the values of these expressions in the right-hand bar, known as the **results sidebar**, like so:



If you want, you can remove the whitespace surrounding the operator:

```
2+6
```

Removing the whitespace is an all or nothing, you can't mix styles. For example:

```
2+6    // OK
2 + 6  // OK
2 +6   // ERROR
2+ 6   // ERROR
```

It's often easier to read expressions if you have white space on either side of the operator.

Decimal numbers

All of the operations above have used whole numbers, more formally known as **integers**. However, as you will know, not every number is whole.

As an example, consider the following:

```
22 / 7
```

This, you may be surprised to know, results in the number 3. This is because if you only use integers in your expression, Swift makes the result an integer also. In this case, the result is rounded down to the next integer.

You can tell Swift to use decimal numbers by changing it to the following:

```
22.0 / 7.0
```

This time, the result is 3.142857142857143 as expected.

The remainder operation

The four operations you've seen so far are easy to understand because you've been doing them for most of your life. Swift also has more complex operations you can use, all of them standard mathematical operations, just less common ones. Let's turn to them now.

The first of these is the **remainder** operation, also called the modulo operation. In division, the denominator goes into the numerator a whole number of times, plus a remainder. This remainder is exactly what the remainder operation gives. For example, 10 modulo 3 equals 1, because 3 goes into 10 three times, with a remainder of 1.

In Swift, the remainder operator is the % symbol, and you use it like so:

```
28 % 10
```

In this case, the result equals 8, because 10 goes into 28 twice with a remainder of 8. If you want to compute the same thing using decimal numbers you do it like so:

```
(28.0).truncatingRemainder(dividingBy: 10.0)
```

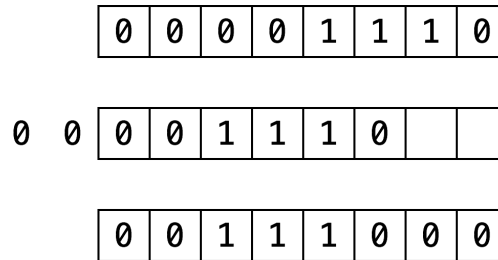
This computes divides 28 by 10 and then **truncates** the result, chopping off any extra decimals and returns the remainder of that. The result is identical to % when there are no decimals.

Shift operations

The **shift left** and **shift right** operations take the binary form of a decimal number and shift the digits left or right, respectively. Then they return the decimal form of the new binary number.

For example, the decimal number 14 in binary, padded to 8 digits, is 00001110. Shifting this left by two places results in 00111000, which is 56 in decimal.

Here's an illustration of what happens during this shift operation:



The digits that come in to fill the empty spots on the right become 0. The digits that fall off the end on the left are lost.

Shifting right is the same, but the digits move to the right.

The operators for these two operations are as follows:

- Shift left: <<
- Shift right: >>

These are the first operators you've seen that contain more than one character. Operators can contain any number of characters, in fact.

Here's an example that uses both of these operators:

```
1 << 3
32 >> 2
```

Both of these values equal the number 8.

One reason for using shifts is to make multiplying or dividing by powers of two easy. Notice that shifting left by one is the same as multiplying by two, shifting left by two is the same as multiplying by four, and so on. Likewise, shifting right by one is the same as dividing by two, shifting right by two is the same as dividing by four, and so on.

In the old days, code often made use of this trick because shifting bits is much simpler for a CPU to do than complex multiplication and division arithmetic. Therefore the code was quicker if it used shifting. However these days, CPUs are much faster and compilers can even convert multiplication and division by powers of two into shifts for you. So you'll see shifting only for binary twiddling, which you probably won't see unless you become an embedded systems programmer!

Order of operations

Of course, it's likely that when you calculate a value, you'll want to use multiple operators. Here's an example of how to do this in Swift:

```
((8000 / (5 * 10)) - 32) >> (29 % 5)
```

Note the use of parentheses, which in Swift serve two purposes: to make it clear to anyone reading the code — including yourself — what you meant, and to disambiguate. For example, consider the following:

```
350 / 5 + 2
```

Does this equal 72 (350 divided by 5, plus 2) or 50 (350 divided by 7)? Those of you who paid attention in school will be screaming “72!” And you would be right!

Swift uses the same reasoning and achieves this through what's known as **operator precedence**. The division operator (/) has a higher precedence than the addition operator (+), so in this example, the code executes the division operation first.

If you wanted Swift to do the addition first — that is, to return 50 — then you could use parentheses like so:

```
350 / (5 + 2)
```

The precedence rules follow the same that you learned in math at school. Multiply and divide have the same precedence, higher than add and subtract which also have the same precedence.

Math functions

Swift also has a vast range of math functions for you to use when necessary. You never know when you need to pull out some trigonometry, especially when you're a pro at Swift and writing those complex games!

Note: Not all of these functions are part of Swift. Some are provided by the operating system. Don't remove the import statement that comes as part of the playground template or Xcode will tell you it can't find these functions.

For example, consider the following:

```
sin(45 * Double.pi / 180)  
// 0.7071067811865475
```



```
cos(135 * Double.pi / 180)
// -0.7071067811865475
```

These compute the sine and cosine respectively. Notice how both make use of `Double.pi` which is a constant Swift provides us, ready-made with `pi` to as much precision as is possible by the computer. Neat!

Then there's this:

```
(2.0).squareRoot()
// 1.414213562373095
```

This computes the square root of 2. Did you know that $\sin(45^\circ)$ equals 1 over the square root of 2?

Not to mention these would be a shame:

```
max(5, 10)
// 10

min(-5, -10)
// -10
```

These compute the maximum and minimum of two numbers respectively.

If you're particularly adventurous you can even combine these functions like so:

```
max((2.0).squareRoot(), Double.pi / 2)
// 1.570796326794897
```

Naming data

At its simplest, computer programming is all about manipulating data. Remember, everything you see on your screen can be reduced to numbers that you send to the CPU. Sometimes you yourself represent and work with this data as various types of numbers, but other times the data comes in more complex forms such as text, images and collections.

In your Swift code, you can give each piece of data a name you can use to refer to it later. The name carries with it an associated **type** that denotes what sort of data the name refers to, such as text, numbers, or a date.

You'll learn about some of the basic types in this chapter, and you'll encounter many other types throughout the rest of this book.

Constants

Take a look at this:

```
let number: Int = 10
```

This declares a constant called `number` which is of type `Int`. Then it sets the value of the constant to the number 10.

Note: Thinking back to operators, here's another one. The equals sign, `=`, is known as the **assignment operator**.

The type `Int` can store integers. The way you store decimal numbers is like so:

```
let pi: Double = 3.14159
```

This is similar to the `Int` constant, except the name and the type are different. This time, the constant is a `Double`, a type that can store decimals with high precision.

There's also a type called `Float`, short for floating point, that stores decimals with lower precision than `Double`. In fact, `Double` has about double the precision of `Float`, which is why it's called `Double` in the first place. A `Float` takes up less memory than a `Double` but generally, memory use for numbers isn't a huge issue and you'll see `Double` used in most places.

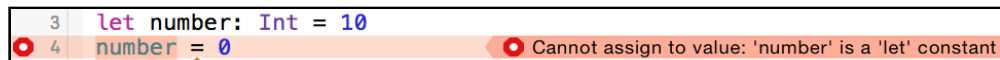
Once you've declared a constant, you can't change its data. For example, consider the following code:

```
number = 0
```

This code produces an error:

```
Cannot assign to value: 'number' is a 'let' constant
```

In Xcode, you would see the error represented this way:



```
3 let number: Int = 10
4 number = 0
```

Cannot assign to value: 'number' is a 'let' constant

Constants are useful for values that aren't going to change. For example, if you were modeling an airplane and needed to keep track of the total number of seats available, you could use a constant.

You might even use a constant for something like a person's age. Even though their age will change as their birthday comes, you might only be concerned with their age at this particular instant.

Variables

Often you want to change the data behind a name. For example, if you were keeping track of your bank account balance with deposits and withdrawals, you might use a variable rather than a constant.

If your program's data never changed, then it would be a rather boring program! But as you've seen, it's not possible to change the data behind a constant.

When you know you'll need to change some data, you should use a variable to represent that data instead of a constant. You declare a variable in a similar way, like so:

```
var variableNumber: Int = 42
```

Only the first part of the statement is different: You declare constants using `let`, whereas you declare variables using `var`.

Once you've declared a variable, you're free to change it to whatever you wish, as long as the type remains the same. For example, to change the variable declared above, you could do this:

```
variableNumber = 0  
variableNumber = 1_000_000
```

To change a variable, you simply assign it a new value.

Note: In Swift, you can optionally use underscores to make larger numbers more human-readable. The quantity and placement of the underscores is up to you.

This is a good time to take a closer look at the results sidebar of the playground. When you type the code above into a playground, you'll see that the results sidebar on the right shows the current value of `variableNumber` at each line:

8	<code>var variableNumber: Int = 42</code>	42
9	<code>variableNumber = 0</code>	0
10	<code>variableNumber = 1_000_000</code>	1,000,000

The results sidebar will show a relevant result for each line if one exists. In the case of a variable or constant, the result will be the new value, whether you've just declared a constant, or declared or reassigned a variable.

Using meaningful names

Always try to choose meaningful names for your variables and constants. Good names can act as documentation and make your code easy to read.

A good name *specifically* describes the role of variable or constant. Here are some examples of good names:

- personAge
- numberOfPeople
- gradePointAverage

Often a bad name is simply not descriptive enough. Here are some examples of bad names:

- a
- temp
- average

The key is to ensure that you'll understand what the variable or constant refers to when you read it again later. Don't make the mistake of thinking you have an infallible memory! It's common in computer programming to look back at your own code as early as a day or two later and have forgotten what it does. Make it easier for yourself by giving your variables and constants intuitive, precise names.

Also, note how the names above are written. In Swift, it is common to **camel case** names. For variables and constants, follow these rules to properly case your names:

- Start with a lowercase letter.
- If the name is made up of multiple words, join them together and start every other word with an uppercase letter.
- If one of these words is an abbreviation, write the entire abbreviation in the same case (e.g.: `sourceURL` and `urlDescription`)

In Swift, you can even use the full range of Unicode characters. For example, you could declare a variable like so:

```
var 🙈💩: Int = -1
```

That might make you laugh, but use caution with special characters like these. They are harder to type and therefore may end up causing you more pain than amusement.

Special characters like these probably make more sense in *data* that you store rather than in Swift code; you'll learn more about Unicode in Chapter 9, "Strings."

Increment and decrement

A common operation that you will need is to be able to increment or decrement a variable. In Swift, this is achieved like so:

```
var counter: Int = 0

counter += 1
// counter = 1

counter -= 1
// counter = 0
```

The counter variable begins as 0. The increment sets its value to 1, and then the decrement sets its value back to 0.

These operators are similar to the assignment operator (=), except they also perform an addition or subtraction. They take the current value of the variable, add or subtract the given value and assign the result to the variable.

In other words, the code above is shorthand for the following:

```
var counter: Int = 0
counter = counter + 1
counter = counter - 1
```

Similarly, the *= and /= operators do the equivalent for multiplication and division, respectively:

```
counter = 10

counter *= 3 // same as counter = counter * 3
// counter = 30

counter /= 2 // same as counter = counter / 2
// counter = 15
```

Mini-exercises

If you haven't been following along with the code in Xcode, now's the time to create a new playground and try some exercises to test yourself!

1. Declare a constant of type `Int` called `myAge` and set it to your age.
2. Declare a variable of type `Double` called `averageAge`. Initially, set it to your own age.

Then, set it to the average of your age and my own age of 30.

3. Create a constant called `testNumber` and initialize it with whatever integer you'd like. Next, create another constant called `evenOdd` and set it equal to `testNumber` modulo 2. Now change `testNumber` to various numbers. What do you notice about `evenOdd`?
4. Create a variable called `answer` and initialize it with the value 0. Increment it by 1. Add 10 to it. Multiply it by 10. Then, shift it to the right by 3. After all of these operations, what's the answer?

Key points

- Computers, at their most fundamental level, perform simple mathematics.
- A programming language allows you to write code, which the compiler converts into instructions that the CPU can execute.
- Computers operate on numbers in base 2 form, otherwise known as binary.
- The IDE you use to write Swift code is named Xcode.
- By providing immediate feedback about how code is executing, playgrounds allow you to write and test Swift code quickly and efficiently.
- Code comments are denoted by a line starting with `//` or multiple lines bookended with `/*` and `*/`.
- Code comments can be used to document your code.
- You can use `print` to write things to the debug area.
- The arithmetic operators are:

```
Add: +  
Subtract: -  
Multiply: *  
Divide: /  
Remainder: %
```

- Constants and variables give names to data.
- Once you've declared a constant, you can't change its data, but you can change a variable's data at any time.

- Always give variables and constants meaningful names to save yourself and your colleagues headaches later.
- Operators to perform arithmetic and then assign back to the variable:

```
Add and assign: +=  
Subtract and assign: -=  
Multiply and assign: *=  
Divide and assign: /=
```

Where to go from here?

In this chapter, you've only dealt with only numbers, both integers and decimals. Of course, there's more to the world of code than that! In the next chapter, you're going to learn about more types such as strings, which allow you to store text.

Challenges

Before moving on, here are some challenges to test your knowledge of variables and constants. You can try the code in a playground to check your answers.

1. Declare a constant `exercises` with value 9 and a variable `exercisesSolved` with value 0. Increment this variable every time you solve an exercise (including this one).
2. Given the following code:

```
age = 16  
print(age)  
age = 30  
print(age)
```

Declare `age` so that it compiles. Did you use `var` or `let`?

3. Consider the following code:

```
let a: Int = 46  
let b: Int = 10
```

Work out what answer equals when you replace the final line of code above with each of these options:

```
// 1  
let answer1: Int = (a * 100) + b  
// 2
```

```
let answer2: Int = (a * 100) + (b * 100)
// 3
let answer3: Int = (a * 100) + (b / 10)
```

4. Add parentheses to the following calculation. The parentheses should show the order in which the operations are performed and should not alter the result of the calculation.

```
5 * 3 - 4 / 2 * 2
```

5. Declare two constants `a` and `b` of type `Double` and assign both a value. Calculate the average of `a` and `b` and store the result in a constant named `average`.
6. A temperature expressed in °C can be converted to °F by multiplying by 1.8 then incrementing by 32. In this challenge, do the reverse: convert a temperature from °F to °C. Declare a constant named `fahrenheit` of type `Double` and assign it a value. Calculate the corresponding temperature in °C and store the result in a constant named `celcius`.
7. Suppose the squares on a chessboard are numbered left to right, top to bottom, with 0 being the top-left square and 63 being the bottom-right square. Rows are numbered top to bottom, 0 to 7. Columns are numbered left to right, 0 to 7. Declare a constant `position` and assign it a value between 0 and 63. Calculate the corresponding row and column numbers and store the results in constants named `row` and `column`.
8. A circle is made up of 2π radians, corresponding with 360 degrees. Declare a constant `degrees` of type `Double` and assign it an initial value. Calculate the corresponding angle in radians and store the result in a constant named `radians`.
9. Declare four constants named `x1`, `y1`, `x2` and `y2` of type `Double`. These constants represent the 2-dimensional coordinates of two points. Calculate the distance between these two points and store the result in a constant named `distance`.

Chapter 9: Types & Operations

By Matt Galloway

Now that you know how to perform basic operations and manipulate data using operations, it's time to learn more about **types**. Formally, a **type** describes a set of values and the operations that can be performed on them.

In this chapter, you'll learn about handling different types, including strings which allow you to represent text. You'll learn about converting between types and you'll also be introduced to type inference which makes your life as a programmer a lot simpler.

Finally, you'll learn about tuples which allow you to make your own types made up of multiple values of any type.

Type conversion

Sometimes you'll have data in one format and need to convert it to another. The naïve way to attempt this would be like so:

```
var integer: Int = 100
var decimal: Double = 12.5
integer = decimal
```

Swift will complain if you try to do this and spit out an error on the third line:

```
Cannot assign value of type 'Double' to type 'Int'
```

Some programming languages aren't as strict and will perform conversions like this automatically. Experience shows this kind of automatic conversion is a source of software bugs and often hurts performance. Swift disallows you from assigning a value of one type to another and avoids these issues.

Remember, computers rely on us programmers to tell them what to do. In Swift, that includes being explicit about type conversions. If you want the conversion to happen, you have to say so!

Instead of simply assigning, you need to explicitly say that you want to convert the type. You do it like so:

```
integer = Int(decimal)
```

The assignment on the third line now tells Swift unequivocally that you want to convert from the original type, `Double`, to the new type, `Int`.

Note: In this case, assigning the decimal value to the integer results in a loss of precision: The `integer` variable ends up with the value 12 instead of 12.5. This is why it's important to be explicit. Swift wants to make sure you know what you're doing and that you may end up losing data by performing the type conversion.

Operators with mixed types

So far, you've only seen operators acting independently on integers or doubles. But what if you have an integer that you want to multiply by a double?

You might think you could do it like this:

```
let hourlyRate: Double = 19.5
let hoursWorked: Int = 10
let totalCost: Double = hourlyRate * hoursWorked
```

If you try that, you'll get an error on the final line:

```
Binary operator '*' cannot be applied to operands of type 'Double' and 'Int'
```

This is because in Swift, you can't apply the `*` operator to mixed types. This rule also applies to the other arithmetic operators. It may seem surprising at first, but Swift is being rather helpful.

Swift forces you to be explicit about what you mean when you want an `Int` multiplied by a `Double`, because the result can be only *one* type. Do you want the result to be an `Int`, converting the `Double` to an `Int` before performing the multiplication? Or do you want the result to be a `Double`, converting the `Int` to a `Double` before performing the multiplication?

In this example, you want the result to be a `Double`. You don't want an `Int`, because in that case, Swift would convert the `hourlyRate` constant into an `Int` to perform the multiplication, rounding it down to 19 and losing the precision of the `Double`.

You need to tell Swift you want it to consider the `hoursWorked` constant to be a `Double`, like so:

```
let totalCost: Double = hourlyRate * Double(hoursWorked)
```

Now, each of the operands will be a `Double` when Swift multiplies them, so `totalCost` is a `Double` as well.

Type inference

Up to this point in this book, each time you've seen a variable or constant declared it's been accompanied by a type annotation. You may be asking yourself why you need to bother writing the `: Int` and `: Double`, since the right hand side of the assignment *is already* an `Int` or a `Double`. It's redundant, to be sure; your crazy-clever brain can see this without too much work.

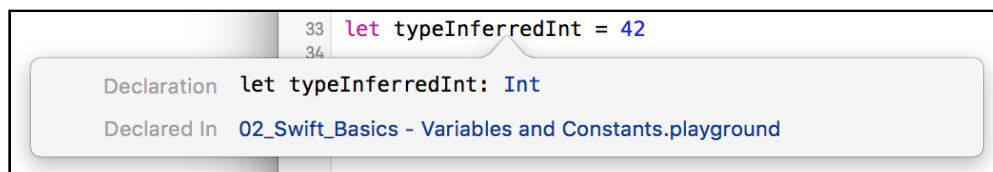
It turns out the Swift compiler can deduce this as well. It doesn't need you to tell it the type all the time — it can figure it out on its own. This is done through a process called **type inference**. Not all programming languages have this, but Swift does, and it's a key component of Swift's power as a language.

So, you can simply drop the type in most places where you see one.

For example, consider the following constant declaration:

```
let typeInferredInt = 42
```

Sometimes it's useful to check the inferred type of a variable or constant. You can do this in a playground by holding down the **Option** key and clicking on the variable or constant's name. Xcode will display a popover like this:

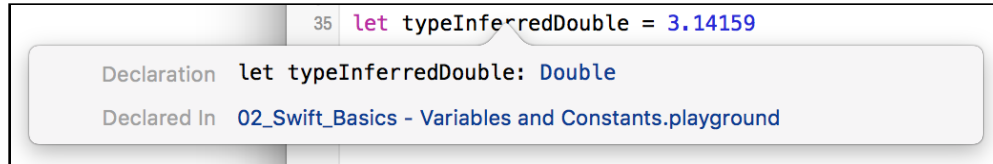


Xcode tells you the inferred type by giving you the declaration you would have had to use if there were no type inference. In this case, the type is `Int`.

It works for other types, too:

```
let typeInferredDouble = 3.14159
```

Option-clicking on this reveals the following:



You can see from this that type inference isn't magic. Swift is simply doing what your brain does very easily. Programming languages that don't use type inference can often feel verbose, because you need to specify the often obvious type each time you declare a variable or constant.

Note: In later chapters, you'll learn about more complex types where sometimes Swift can't infer the type. That's a pretty rare case though, and you'll see type inference used for most of the code examples in this book — except in cases where we want to highlight the type for you.

Sometimes you want to define a constant or variable and ensure it's a certain type, even though what you're assigning to it is a different type. You saw earlier how you can convert from one type to another. For example, consider the following:

```
let wantADouble = 3
```

Here, Swift infers the type of `wantADouble` as `Int`. But what if you wanted `Double` instead?

The first thing you could do is the following:

```
let actuallyDouble = Double(3)
```

This is like you saw before with type conversion.

Another option would be to not use type inference at all and do the following:

```
let actuallyDouble: Double = 3
```

There is a third option, like so:

```
let actuallyDouble = 3 as Double
```

This uses a new keyword you haven't seen before, `as`. It also performs a type conversion, and you will see this throughout the book.

Note: Literal values like 3 don't have a type. It's only when using them in an expression or assigning them to a constant or variable that Swift infers a type for them.

A literal number value that doesn't contain a decimal point can be used as an `Int` as well as a `Double`. This is why you're allowed to assign the value 3 to constant `actuallyDouble`.

Literal number values that *do* contain a decimal point cannot be integers. This means we could have avoided this entire discussion had we started with

```
let wantADouble = 3.0
```

Sorry! :]

Mini-exercises

1. Create a constant called `age1` and set it equal to 42. Create a constant called `age2` and set it equal to 21. Check using Option-click that the type for both has been inferred correctly as `Int`.
2. Create a constant called `avg1` and set it equal to the average of `age1` and `age2` using the naïve operation $(age1 + age2) / 2$. Use Option-click to check the type and check the result of `avg1`. Why is it wrong?
3. Correct the mistake in the above exercise by converting `age1` and `age2` to type `Double` in the formula. Use Option-click to check the type and check the result of `avg1`. Why is it now correct?

Strings

Numbers are essential in programming, but they aren't the only type of data you need to work with in your apps. Text is also an extremely common data type, such as people's names, their addresses, or even the words of a book. All of these are examples of text that an app might need to handle.

Most computer programming languages store text in a data type called a **string**. This chapter introduces you to strings, first by giving you background on the concept of strings and then by showing you how to use them in Swift.

How computers represent strings

Computers think of strings as a collection of individual **characters**. In Chapter 1 of this book, you learned that numbers are the language of CPUs, and all code, in whatever programming language, can be reduced to raw numbers. Strings are no different!

That may sound very strange. How can characters be numbers? At its base, a computer needs to be able to translate a character into the computer's own language, and it does so by assigning each character a different number. This forms a two-way mapping from character to number that is called a **character set**.

When you press a character key on your keyboard, you are actually communicating the number of the character to the computer. Your word processor application converts that number into a picture of the character and finally, presents that picture to you.

Unicode

In isolation, a computer is free to choose whatever character set mapping it likes. If the computer wants the letter **a** to equal the number 10, then so be it. But when computers start talking to each other, they need to use a common character set. If two computers used different character sets, then when one computer transferred a string to the other, they would end up thinking the strings contained different characters.

There have been several standards over the years, but the most modern standard is **Unicode**. It defines the character set mapping that almost all computers use today.

Note: You can read more about Unicode at its official website, <http://unicode.org/>.

As an example, consider the word **cafe**. The Unicode standard tells us that the letters of this word should be mapped to numbers like so:

c	a	f	e
99	97	102	101

The number associated with each character is called a **code point**. So in the example above, **c** uses code point 99, **a** uses code point 97, and so on.

Of course, Unicode is not just for the simple Latin characters used in English, such as **c**, **a**, **f** and **e**. It also lets you map characters from languages around the world. The word **cafe**, as you're probably aware, is derived from French, in which it's written as **café**.



Unicode maps these characters like so:

c	a	f	é
99	97	102	233

And here's an example using Chinese characters (this, according to Google translate, means "Computer Programming"):

电	脑	编	程
30005	33041	32534	31243

You've probably heard of emojis, which are small pictures you can use in your text. These pictures are, in fact, just normal characters and are also mapped by Unicode. For example:

	
128169	128512

This is only two characters. The code points for these are very large numbers, but each is still only a single code point. The computer considers these as no different than any other two characters.

Note: The word "emoji" comes from Japanese, where "e" means picture and "moji" means character.

Strings in Swift

Swift, like any good programming language, can work directly with characters and strings. It does so through the data types `Character` and `String`, respectively. In this section, you'll learn about these data types and how to work with them.

Characters and strings

The Character data type can store a single character. For example:

```
let characterA: Character = "a"
```

This stores the character **a**. It can hold any character — even an emoji:

```
let characterDog: Character = "🐶"
```

But this data type is designed to hold only single characters. The String data type, on the other hand, stores multiple characters. For example:

```
let stringDog: String = "Dog"
```

It's as simple as that! The right-hand side of this expression is what's known as a **string literal**; it's the Swift syntax for representing a string.

Of course, type inference applies here as well. If you remove the type in the above declaration, then Swift does the right thing and makes the stringDog a String constant:

```
let stringDog = "Dog" // Inferred to be of type String
```

Note: There's no such thing as a character literal in Swift. A character is simply a string of length one. However, Swift infers the type of any string literal to be String, so if you want a Character instead, you must make the type explicit.

Concatenation

You can do much more than create simple strings. Sometimes you need to manipulate a string, and one common way to do so is to combine it with another string.

In Swift, you do this in a rather simple way: by using the addition operator. Just as you can add numbers, you can add strings:

```
var message = "Hello" + " my name is "  
let name = "Matt"  
message += name // "Hello my name is Matt"
```

You need to declare message as a variable rather than a constant because you want to modify it. You can add string literals together, as in the first line, and you can add string variables or constants together, as in the last line.

It's also possible to add characters to a string. However, Swift's strictness with types means you have to be explicit when doing so, just as you have to be when you work with numbers if one is an `Int` and the other is a `Double`.

To add a character to a string, you do this:

```
let exclamationMark: Character = "!"  
message += String(exclamationMark) // "Hello my name is Matt!"
```

With this code, you explicitly convert the `Character` to a `String` before you add it to `message`.

Interpolation

You can also build up a string by using **interpolation**, which is a special Swift syntax that lets you build a string in a way that's easy to read:

```
message = "Hello my name is \(name)!" // "Hello my name is Matt!"
```

As I'm sure you'll agree, this is much more readable than the example from the previous section. It's an extension of the string literal syntax, whereby you replace certain parts of the string with other values. You enclose the value you want to insert in parentheses preceded by a backslash.

This syntax works in just the same way to build a string from other data types, such as numbers:

```
let oneThird = 1.0 / 3.0  
let oneThirdLongString = "One third is \(oneThird) as a decimal."
```

Here, you use a `Double` in the interpolation. At the end of this code, your `oneThirdLongString` constant will contain the following:

```
One third is 0.3333333333333333 as a decimal.
```

Of course, it would actually take infinite characters to represent one third as a decimal, because it's a repeating decimal. String interpolation with a `Double` gives you no way to control the precision of the resulting string.

This is an unfortunate consequence of using string interpolation: It's simple to use, but offers no ability to customize the output.

Multi-line strings

Swift has a neat way to express strings that contain multiple lines. This can be rather useful when you need to put a very long string in your code.

You do it like so:

```
let bigString = """
    You can have a string
    that contains multiple
    lines
    by
    doing this.
    """
print(bigString)
```

The three double-quotes signify that this is a multi-line string. Handily, the first and final new lines do not become part of the string. This makes it more flexible as you don't have to have the three double-quotes on the same line as the string.

In the case above, it will print the following:

```
You can have a string
that contains multiple
lines
by
doing this.
```

Notice that the 2-space margin in the multiline string literal is stripped out of the result. Swift looks at number of leading spaces on the final three double-quotes line. Using this as a baseline, Swift requires that all lines above it have at least that much space so it can remove it from each line. This lets you format your code with pretty indentation without effecting the output.

Mini-exercises

1. Create a string constant called `firstName` and initialize it to your first name. Also create a string constant called `lastName` and initialize it to your last name.
2. Create a string constant called `fullName` by adding the `firstName` and `lastName` constants together, separated by a space.
3. Using interpolation, create a string constant called `myDetails` that uses the `fullName` constant to create a string introducing yourself. For example, my string would read: "Hello, my name is Matt Galloway."

Tuples

Sometimes data comes in pairs or triplets. An example of this is a pair of (x, y) coordinates on a 2D grid. Similarly, a set of coordinates on a 3D grid is comprised of an x-value, a y-value and a z-value.

In Swift, you can represent such related data in a very simple way through the use of a *tuple*.

A tuple is a type that represents data composed of more than one value of any type. You can have as many values in your tuple as you like. For example, you can define a pair of 2D coordinates where each axis value is an integer, like so:

```
let coordinates: (Int, Int) = (2, 3)
```

The type of `coordinates` is `(Int, Int)`. The types of the values within the tuple, in this case `Int`, are separated by commas and surrounded by parentheses. The code for creating the tuple is much the same, with each value separated by commas and surrounded by parentheses.

Type inference can infer tuple types too:

```
let coordinates = (2, 3)
```

You could similarly create a tuple of `Double` values, like so:

```
let coordinatesDoubles = (2.1, 3.5)
// Inferred to be of type (Double, Double)
```

Or you could mix and match the types comprising the tuple, like so:

```
let coordinatesMixed = (2.1, 3)
// Inferred to be of type (Double, Int)
```

And here's how to access the data inside a tuple:

```
let x1 = coordinates.0
let y1 = coordinates.1
```

You can reference each item in the tuple by its position in the tuple, starting with zero. So in this example, `x1` will equal 2 and `y1` will equal 3.

Note: Starting with zero is a common practice in computer programming and is called **zero indexing**. You'll see this again in Chapter 7, "Arrays, Dictionaries, Sets".

In the previous example, it may not be immediately obvious that the first value, at index 0, is the x-coordinate and the second value, at index 1, is the y-coordinate. This is another demonstration of why it's important to *always* name your variables in a way that avoids confusion.

Fortunately, Swift allows you to name the individual parts of a tuple, and you can be explicit about what each part represents. For example:

```
let coordinatesNamed = (x: 2, y: 3)
// Inferred to be of type (x: Int, y: Int)
```

Here, the code annotates the values of `coordinatesNamed` to contain a label for each part of the tuple.

Then, when you need to access each part of the tuple, you can access it by its name:

```
let x2 = coordinatesNamed.x
let y2 = coordinatesNamed.y
```

This is much clearer and easier to understand. More often than not, it's helpful to name the components of your tuples.

If you want to access multiple parts of the tuple at the same time, as in the examples above, you can also use a shorthand syntax to make it easier:

```
let coordinates3D = (x: 2, y: 3, z: 1)
let (x3, y3, z3) = coordinates3D
```

This declares three new constants, `x3`, `y3` and `z3`, and assigns each part of the tuple to them in turn. The code is equivalent to the following:

```
let coordinates3D = (x: 2, y: 3, z: 1)
let x3 = coordinates3D.x
let y3 = coordinates3D.y
let z3 = coordinates3D.z
```

If you want to ignore a certain element of the tuple, you can replace the corresponding part of the declaration with an underscore. For example, if you were performing a 2D calculation and wanted to ignore the z-coordinate of `coordinates3D`, then you'd write the following:

```
let (x4, y4, _) = coordinates3D
```

This line of code only declares `x4` and `y4`. The `_` is special and simply means you're ignoring this part for now.

Note: You'll find that you can use the underscore (also called the wildcard operator) throughout Swift to ignore a value.

Mini-exercises

1. Declare a constant tuple that contains three `Int` values followed by a `Double`. Use this to represent a date (month, day, year) followed by an average temperature for that date.
2. Change the tuple to name the constituent components. Give them names related to the data that they contain: month, day, year and averageTemperature.
3. In one line, read the day and average temperature values into two constants. You'll need to employ the underscore to ignore the month and year.
4. Up until now, you've only seen constant tuples. But you can create variable tuples, too. Change the tuple you created in the exercises above to a variable by using `var` instead of `let`. Now change the average temperature to a new value.

A whole lot of number types

You've been using `Int` to represent whole numbers. An `Int` is represented with 64 bits on most modern hardware and with 32 bits on older, or more resource constrained systems. Swift provides many more number types that use different amounts of storage. For whole numbers, you can use the explicit **signed** types `Int8`, `Int16`, `Int32`, `Int64`. These types consume 1, 2, 4, and 8 bytes of storage respectively. Each of these types use 1 bit to represent the sign.

If you are only dealing with non-negative values there are a set of explicit **unsigned** types that you can use. These include `UInt8`, `UInt16`, `UInt32` and `UInt64`. While you cannot represent negative values with these, the extra 1 bit lets you represent values that are twice as big as their **signed** counterparts.

Here is a summary of the different integer types and their storage size in bytes. Most of the time you will just want to use an `Int`. These become useful if your code is interacting with another piece of software that uses one of these more exact sizes or if you need to optimize for storage size.

Type	Minimum value	Maximum value	Storage size
Int8	-128	127	1
UInt8	0	255	1
Int16	-32768	32767	2
UInt16	0	65535	2
Int32	-2147483648	2147483647	4
UInt32	0	4294967295	4
Int64	-9223372036854775808	9223372036854775807	8
UInt64	0	18446744073709551615	8

You've been using `Double` to represent fractional numbers. Swift offers a `Float` type which has less range and precision than `Double` but requires half as much storage. Modern hardware has been optimized for `Double` so it is the one that you should reach for unless you have good reason not to.

Type	Minimum value	Maximum value	Precision	Storage size
Float	1.175494E-38	3.402823E+38	6 digits	4
Double	2.225073e-308	1.797693E+308	15 digits	8

Most of the time you will just use `Int` and `Double` to represent numbers but every once in a while you might encounter the other types.

For example, suppose you need to add together an `Int16` with a `UInt8` and an `Int32`. You can do that like so:

```
let a: Int16 = 12
let b: UInt8 = 255
let c: Int32 = -100000

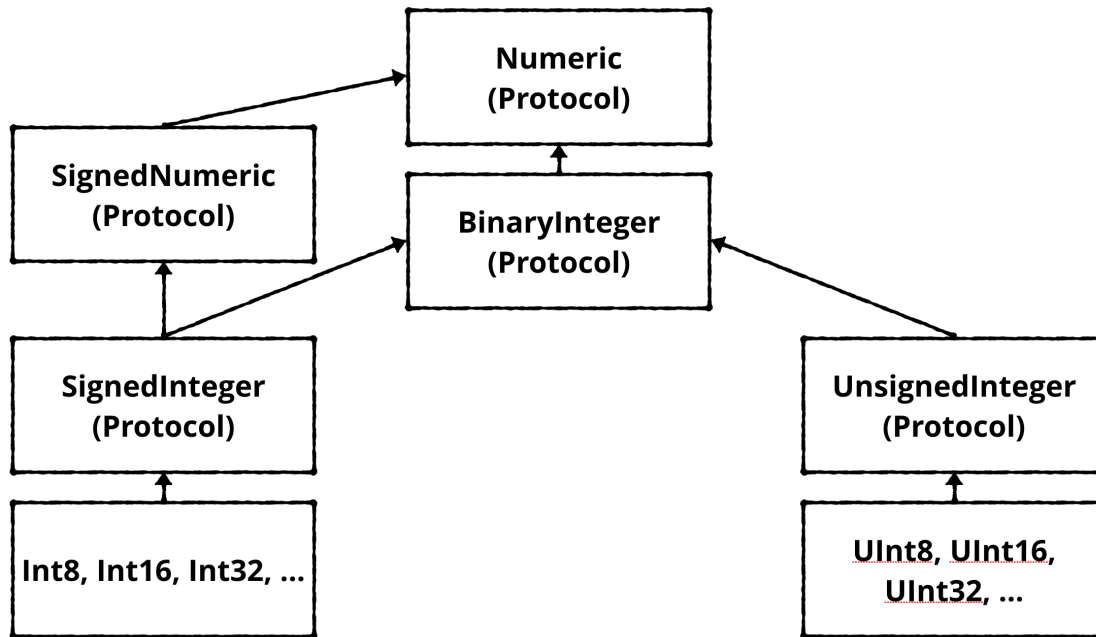
let answer = Int(a) + Int(b) + Int(c) // answer is an Int
```

A peek behind the curtains: Protocols

Even though there are a dozen different numeric types, they are pretty easy to understand and use. This is because they all roughly support the same operations. In other words, once you know how to use an `Int`, using any one of the flavors is very straight-forward.

One of the truly great features of Swift is that it formalizes the idea of type commonality using what are known as **protocols**. By learning a protocol, you instantly understand how an entire family of types that use that protocol work.

In the case of integers, the functionality can be diagrammed like so:



The arrows indicate conformance to (sometimes called *adoption of*) a protocol. While this graph does not show all of the protocols that integer types conform to, it gives you insight about how things are organized.

Swift is the first protocol-based language. As you begin to understand the protocols that underly the types, you can leverage the system in ways not possible with other languages.

By the end of this book you will be hooking into existing protocols and even creating new ones of your own.

Key points

- Type conversion allows you to convert values of one type into another.
- Type conversion is required when using an operator, such as the basic arithmetic operators (+, −, *, /), with mixed types.
- Type inference allows you to omit the type when Swift already knows it.
- **Unicode** is the standard for mapping characters to numbers.
- A single mapping in Unicode is called a **code point**.

- The `Character` data type stores single characters. The `String` data type stores collections of characters, or strings.
- You can combine strings by using the addition operator.
- You can use **string interpolation** to build a string in-place.
- You can use tuples to group data into a single data type.
- Tuples can either be unnamed or named. Their elements are accessed with index numbers for unnamed tuples, or programmer given names for named tuples.
- There are many kinds of numeric types with different storage and precision capabilities.
- Protocols are how types are organized in Swift.

Where to go from here?

Types are a fundamental part of programming. They're what allow you to correctly store your data. You've seen a few more here, including strings and tuples as well as a bunch of numeric types. Later on in the book you'll learn how to define your own types with **structs**, **enums** and **classes**.

In the next chapter, you'll learn about Boolean logic and simple control flow. This is required for any program to be able to make decisions about how the program should proceed based on the data it's manipulating.

Give the following challenges a try to solidify your knowledge and you'll be ready to move on!

Challenges

1. Create a constant called `coordinates` and assign a tuple containing two and three to it.
2. Create a constant called `namedCoordinate` with a `row` and `column` component.
3. Which of the following are valid statements?

```
let character: Character = "Dog"
let character: Character = "🐶"
let string: String = "Dog"
```



```
let string: String = "🐼"
```

4. Is this valid code?

```
let tuple = (day: 15, month: 8, year: 2015)
let day = tuple.Day
```

5. What is wrong with the following code?

```
let name = "Matt"
name += " Galloway"
```

6. What is the type of the constant named value?

```
let tuple = (100, 1.5, 10)
let value = tuple.1
```

7. What is the value of the constant named month?

```
let tuple = (day: 15, month: 8, year: 2015)
let month = tuple.month
```

8. What is the value of the constant named summary?

```
let number = 10
let multiplier = 5
let summary = "\(number) multiplied by \(multiplier) equals \(number * multiplier)"
```

9. What is the sum of a and b, minus c?

```
let a = 4
let b: Int32 = 100
let c: UInt8 = 12
```

10. What is the numeric difference between Double.pi and Float.pi?

Chapter 10: Basic Control Flow

By Matt Galloway

When writing a computer program, you need to be able to tell the computer what to do in different scenarios. For example, a calculator app would need to do one thing if the user tapped the addition button and another thing if the user tapped the subtraction button.

In computer-programming terms, this concept is known as **control flow**. It is so named because the flow of the program is controlled by various methods. In this chapter, you'll learn how to make decisions and repeat tasks in your programs by using syntax to control the flow. You'll also learn about **Booleans**, which represent true and false values, and how you can use these to compare data.

Comparison operators

You've seen a few types now, such as `Int`, `Double` and `String`. Here you'll learn about another type, one that will let you compare values through the **comparison operators**.

When you perform a comparison, such as looking for the greater of two numbers, the answer is either *true* or *false*. Swift has a data type just for this! It's called a `Bool`, which is short for Boolean, after a rather clever man named George Boole who invented an entire field of mathematics around the concept of true and false.

This is how you use a Boolean in Swift:

```
let yes: Bool = true
let no: Bool = false
```

And because of Swift's type inference, you can leave off the type annotation:

```
let yes = true
let no = false
```

A Boolean can only be either true or false, denoted by the keywords `true` and `false`. In the code above, you use the keywords to set the state of each constant.

Boolean operators

Booleans are commonly used to compare values. For example, you may have two values and you want to know if they're equal: either they are (true) or they aren't (false).

In Swift, you do this using the **equality operator**, which is denoted by `==`:

```
let doesOneEqualTwo = (1 == 2)
```

Swift infers that `doesOneEqualTwo` is a `Bool`. Clearly, 1 does not equal 2, and therefore `doesOneEqualTwo` will be `false`.

Similarly, you can find out if two values are *not* equal using the `!=` operator:

```
let doesOneNotEqualTwo = (1 != 2)
```

This time, the comparison is true because 1 does not equal 2, so `doesOneNotEqualTwo` will be `true`.

The prefix `!` operator, also called the not-operator, toggles true to false and false to true. Another way to write the above is:

```
let alsoTrue = !(1 == 2)
```

Because 1 does not equal 2, `(1 == 2)` is `false`, and then `!` flips it to `true`.

Two more operators let you determine if a value is greater than (`>`) or less than (`<`) another value. You'll likely know these from mathematics:

```
let isOneGreaterThanTwo = (1 > 2)
let isOneLessThanTwo = (1 < 2)
```

And it's not rocket science to work out that `isOneGreaterThanTwo` will equal `false` and `isOneLessThanTwo` will equal `true`.

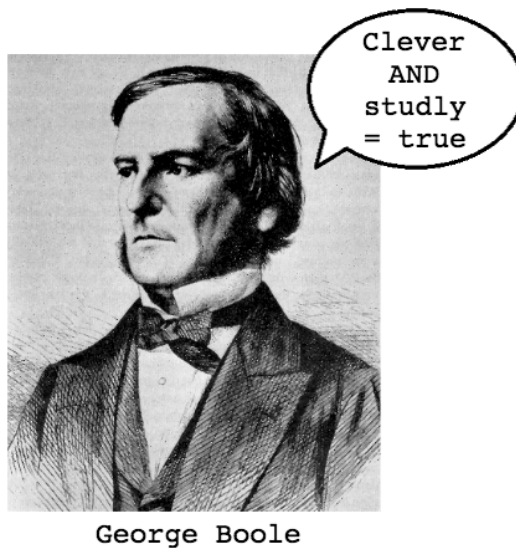
There's also an operator that lets you test if a value is less than *or* equal to another value: `<=`. It's a combination of `<` and `==`, and will therefore return `true` if the first value is either less than the second value or equal to it.

Similarly, there's an operator that lets you test if a value is greater than or equal to another — you may have guessed that it's `>=`.

Boolean logic

Each of the examples above tests just one condition. When George Boole invented the Boolean, he had much more planned for it than these humble beginnings. He invented Boolean logic, which lets you combine multiple conditions to form a result.

One way to combine conditions is by using **AND**. When you AND together two Booleans, the result is another Boolean. If both input Booleans are `true`, then the result is `true`. Otherwise, the result is `false`.



George Boole

In Swift, the operator for Boolean AND is `&&`, used like so:

```
let and = true && true
```

In this case, `and` will be `true`. If either of the values on the right was `false`, then `and` would be `false`.

Another way to combine conditions is by using **OR**. When you OR together two Booleans, the result is `true` if *either* of the input Booleans is `true`. Only if *both* input Booleans are `false` will the result be `false`.

In Swift, the operator for Boolean OR is `||`, used like so:

```
let or = true || false
```

In this case, `or` will be `true`. If both values on the right were `false`, then `or` would be `false`. If both were `true`, then `or` would still be `true`.

In Swift, Boolean logic is usually applied to multiple conditions. Maybe you want to determine if two conditions are true; in that case, you'd use `AND`. If you only care about whether one of two conditions is true, then you'd use `OR`.

For example, consider the following code:

```
let andTrue = 1 < 2 && 4 > 3
let andFalse = 1 < 2 && 3 > 4

let orTrue = 1 < 2 || 3 > 4
let orFalse = 1 == 2 || 3 == 4
```

Each of these tests two separate conditions, combining them with either `AND` or `OR`.

It's also possible to use Boolean logic to combine more than two comparisons. For example, you can form a complex comparison like so:

```
let andOr = (1 < 2 && 3 > 4) || 1 < 4
```

The parentheses disambiguates the expression. First Swift evaluates the sub-expression inside the parentheses, and then it evaluates the full expression, following these steps:

```
1. (1 < 2 && 3 > 4) || 1 < 4
2. (true && false) || true
3. false || true
4. true
```

String equality

Sometimes you want to determine if two strings are equal. For example, a children's game of naming an animal in a photo would need to determine if the player answered correctly.

In Swift, you can compare strings using the standard equality operator, `==`, in exactly the same way as you compare numbers. For example:

```
let guess = "dog"
let dogEqualsCat = guess == "cat"
```

Here, `dogEqualsCat` is a Boolean that in this case equals `false`, because `"dog"` does not equal `"cat"`. Simple!

Just as with numbers, you can compare not just for equality, but also to determine if one value is greater than or less than another value. For example:

```
let order = "cat" < "dog"
```

This syntax checks if one string comes before another alphabetically. In this case, `order` equals `true` because "cat" comes before "dog".

Note: You will learn more about string equality in Chapter 9: Strings. There are some interesting things that crop up when strings contain special characters.

Mini-exercises

1. Create a constant called `myAge` and set it to your age. Then, create a constant named `isTeenager` that uses Boolean logic to determine if the age denotes someone in the age range of 13 to 19.
2. Create another constant named `theirAge` and set it to my age, which is 30. Then, create a constant named `bothTeenagers` that uses Boolean logic to determine if both you and I are teenagers.
3. Create a constant named `reader` and set it to your name as a string. Create a constant named `author` and set it to my name, Matt Galloway. Create a constant named `authorIsReader` that uses string equality to determine if `reader` and `author` are equal.
4. Create a constant named `readerBeforeAuthor` which uses string comparison to determine if `reader` comes before `author`.

The if statement

The first and most common way of controlling the flow of a program is through the use of an **if statement**, which allows the program to do something only *if* a certain condition is true. For example, consider the following:

```
if 2 > 1 {  
    print("Yes, 2 is greater than 1.")  
}
```

This is a simple `if` statement. If the condition is true, then the statement will execute the code between the braces. If the condition is false, then the statement won't execute the code between the braces. It's as simple as that!

You can extend an `if` statement to provide code to run in case the condition turns out to be false. This is known as the **else clause**. Here's an example:

```
let animal = "Fox"

if animal == "Cat" || animal == "Dog" {
    print("Animal is a house pet.")
} else {
    print("Animal is not a house pet.")
}
```

Here, if `animal` equals either "Cat" or "Dog", then the statement will run the first block of code. If `animal` does not equal either "Cat" or "Dog", then the statement will run the block inside the `else` part of the `if` statement, printing the following to the debug area:

```
Animal is not a house pet.
```

But you can go even further than that with `if` statements. Sometimes you want to check one condition, then another. This is where **else-if** comes into play, nesting another `if` statement in the `else` clause of a previous `if` statement.

You can use it like so:

```
let hourOfDay = 12
let timeOfDay: String

if hourOfDay < 6 {
    timeOfDay = "Early morning"
} else if hourOfDay < 12 {
    timeOfDay = "Morning"
} else if hourOfDay < 17 {
    timeOfDay = "Afternoon"
} else if hourOfDay < 20 {
    timeOfDay = "Evening"
} else if hourOfDay < 24 {
    timeOfDay = "Late evening"
} else {
    timeOfDay = "INVALID HOUR!"
}

print(timeOfDay)
```

These nested `if` statements test multiple conditions one by one until a true condition is found. Only the code associated with that first true condition is executed, regardless of whether subsequent `else-if` conditions are true. In other words, the order of your conditions matters!

You can add an `else` clause at the end to handle the case where none of the conditions are true. This `else` clause is optional if you don't need it; in this example you *do* need it, to ensure that `timeOfDay` has a valid value by the time you print it out.

In this example, the `if` statement takes a number representing an hour of the day and converts it to a string representing the part of the day to which the hour belongs. Working with a 24-hour clock, the statements are checked in order, one at a time:

- The first check is to see if the hour is less than 6. If so, that means it's early morning.
- If the hour is not less than 6, the statement continues to the first `else-if`, where it checks the hour to see if it's less than 12.
- Then in turn, as conditions prove false, the statement checks the hour to see if it's less than 17, then less than 20, then less than 24.
- Finally, if the hour is out of range, the statement prints that information to the console.

In the code above, the `hourOfDay` constant is 12. Therefore, the code will print the following:

```
Afternoon
```

Notice that even though both the `hourOfDay < 20` and `hourOfDay < 24` conditions are also true, the statement only executes the first block whose condition is true; in this case, the block with the `hourOfDay < 17` condition.

Short circuiting

An important fact about `if` statements is what happens when there are multiple Boolean conditions separated by ANDs (`&&`) or ORs (`||`).

Consider the following code:

```
if 1 > 2 && name == "Matt Galloway" {  
    // ...  
}
```

The first condition of the `if` statement, `1 > 2` is false. Therefore the whole expression cannot ever be true. So Swift will not even bother to check the second part of the expression, namely the check of `name`.

Similarly, consider the following code:

```
if 1 < 2 || name == "Matt Galloway" {  
    // ...  
}
```


Since `1 < 2` is `true`, the whole expression must be `true` as well. Therefore once again, the check of `name` is not executed. This will come in handy later on when you start dealing with more complex data types.

Encapsulating variables

`if` statements introduce a new concept **scope**, which is a way to encapsulate variables through the use of braces.

Imagine you want to calculate the fee to charge your client. Here's the deal you've made:

You earn \$25 for every hour up to 40 hours, and \$50 for every hour thereafter.

Using Swift, you can calculate your fee in this way:

```
var hoursWorked = 45

var price = 0
if hoursWorked > 40 {
    let hoursOver40 = hoursWorked - 40
    price += hoursOver40 * 50
    hoursWorked -= hoursOver40
}
price += hoursWorked * 25

print(price)
```

This code takes the number of hours and checks if it's over 40. If so, the code calculates the number of hours over 40 and multiplies that by \$50, then adds the result to the price. The code then subtracts the number of hours over 40 from the hours worked. It multiplies the remaining hours worked by \$25 and adds that to the total price.

In the example above, the result is as follows:

1250

The interesting thing here is the code inside the `if` statement. There is a declaration of a new constant, `hoursOver40`, to store the number of hours over 40. Clearly, you can use it inside the `if` statement. But what happens if you try to use it at the end of the above code?

```
...  
print(price)  
print(hoursOver40)
```

This would result in the following error:

```
Use of unresolved identifier 'hoursOver40'
```

This error informs you that you're only allowed to use the `hoursOver40` constant within the scope in which it was created. In this case, the `if` statement introduced a new scope, so when that scope is finished, you can no longer use the constant.

However, each scope can use variables and constants from its parent scope. In the example above, the scope inside of the `if` statement uses the `price` and `hoursWorked` variables, which you created in the parent scope.

The ternary conditional operator

Now I want to introduce a new operator, one you didn't see in Chapter 2. It's called the **ternary conditional operator** and it's related to `if` statements.

If you wanted to determine the minimum and maximum of two variables, you could use `if` statements, like so:

```
let a = 5  
let b = 10  
  
let min: Int  
if a < b {  
    min = a  
} else {  
    min = b  
}  
  
let max: Int  
if a > b {  
    max = a  
} else {  
    max = b  
}
```

By now you know how this works, but it's a lot of code. Wouldn't it be nice if you could shrink this to just a couple of lines? Well, you can, thanks to the ternary conditional operator!

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition was true or false. The syntax is as follows:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

You can use this operator to rewrite your long code block above, like so:

```
let a = 5
let b = 10

let min = a < b ? a : b
let max = a > b ? a : b
```

In the first example, the condition is `a < b`. If this is true, the result assigned back to `min` will be the value of `a`; if it's false, the result will be the value of `b`.

I'm sure you'll agree that's much simpler! This is a useful operator that you'll find yourself using regularly.

Note: Because finding the greater or smaller of two numbers is such a common operation, the Swift standard library provides two functions for this purpose: `max` and `min`. If you were paying attention earlier in the book, then you'll recall you've already seen these.

Mini-exercises

1. Create a constant named `myAge` and initialize it with your age. Write an `if` statement to print out `Teenager` if your age is between 13 and 19, and `Not a teenager` if your age is not between 13 and 19.
2. Create a constant named `answer` and use a ternary condition to set it equal to the result you print out for the same cases in the above exercise. Then print out `answer`.

Loops

Loops are Swift's way of executing code multiple times. In this section, you'll learn about one type of loop: the `while` loop. If you know another programming language, you'll find the concepts and maybe even the syntax to be familiar.

While loops

A **while loop** repeats a block of code while a condition is true.

You create a while loop this way:

```
while <CONDITION> {  
    <LOOP CODE>  
}
```

The loop checks the condition for every iteration, . If the condition is `true`, then the loop executes and moves on to another iteration. If the condition is `false`, then the loop stops. Just like `if` statements, while loops introduce a scope.

The simplest while loop takes this form:

```
while true {  
}
```

This is a while loop that never ends because the condition is always `true`. Of course, you would never write such a while loop, because your program would spin forever! This situation is known as an **infinite loop**, and while it might not cause your program to crash, it will very likely cause your computer to freeze.

Here's a more useful example of a while loop:

```
var sum = 1  
  
while sum < 1000 {  
    sum = sum + (sum + 1)  
}
```

This code calculates a mathematical sequence, up to the point where the value is greater than 1000.

The loop executes as follows:

- **Before iteration 1:** `sum = 1`, loop condition = `true`
- **After iteration 1:** `sum = 3`, loop condition = `true`
- **After iteration 2:** `sum = 7`, loop condition = `true`
- **After iteration 3:** `sum = 15`, loop condition = `true`
- **After iteration 4:** `sum = 31`, loop condition = `true`
- **After iteration 5:** `sum = 63`, loop condition = `true`
- **After iteration 6:** `sum = 127`, loop condition = `true`

- **After iteration 7:** sum = 255, loop condition = true
- **After iteration 8:** sum = 511, loop condition = true
- **After iteration 9:** sum = 1023, loop condition = false

After the ninth iteration, the sum variable is 1023, and therefore the loop condition of `sum < 1000` becomes false. At this point, the loop stops.

Repeat-while loops

A variant of the while loop is called the **repeat-while loop**. It differs from the while loop in that the condition is evaluated *at the end* of the loop rather than at the beginning.

You construct a repeat-while loop like this:

```
repeat {  
    <LOOP CODE>  
} while <CONDITION>
```

Here's the example from the last section, but using a repeat-while loop:

```
sum = 1  
  
repeat {  
    sum = sum + (sum + 1)  
} while sum < 1000
```

In this example, the outcome is the same as before. However, that isn't always the case — you might get a different result with a different condition.

Consider the following while loop:

```
sum = 1  
  
while sum < 1 {  
    sum = sum + (sum + 1)  
}
```

Consider the corresponding repeat-while loop, which uses the same condition:

```
sum = 1  
  
repeat {  
    sum = sum + (sum + 1)  
} while sum < 1
```

In the case of the regular `while` loop, the condition `sum < 1` is false right from the start. That means the body of the loop won't be reached! The value of `sum` will equal 1 because the loop won't execute any iterations.

In the case of the `repeat-while` loop, `sum` will equal 3 because the loop executes once.

Breaking out of a loop

Sometimes you want to break out of a loop early. You can do this using the `break` statement, which immediately stops the execution of the loop and continues on to the code after the loop.

For example, consider the following code:

```
sum = 1

while true {
    sum = sum + (sum + 1)
    if sum >= 1000 {
        break
    }
}
```

Here, the loop condition is `true`, so the loop would normally iterate forever. However, the `break` means the `while` loop will exit once the `sum` is greater than or equal to 1000.

You've seen how to write the same loop in different ways, demonstrating that in computer programming, there are often many ways to achieve the same result. You should choose the method that's easiest to read and conveys your intent in the best way possible. This is an approach you'll internalize with enough time and practice.

Mini-exercises

1. Create a variable named `counter` and set it equal to 0. Create a `while` loop with the condition `counter < 10` which prints out `counter is X` (where `X` is replaced with `counter` value) and then increments `counter` by 1.
2. Create a variable named `counter` and set it equal to 0. Create another variable named `roll` and set it equal to 0. Create a `repeat-while` loop. Inside the loop, set `roll` equal to `Int(arc4random_uniform(6))` which means to pick a random number between 0 and 5. Then increment `counter` by 1. Finally, print `After X rolls, roll is Y` where `X` is the value of `counter` and `Y` is the value of `roll`. Set the loop condition such that the loop finishes when the first 0 is rolled.

Key points

- You use the Boolean data type `Bool` to represent true and false.
- The comparison operators, all of which return a Boolean, are:

```
Equal: `==`  
Not equal: `!=`  
Less than: `<`  
Greater than: `>`  
Less than or equal: `<=`  
Greater than or equal: `>=`
```

- You can use Boolean logic to combine comparison conditions.
- You use `if` statements to make simple decisions based on a condition.
- You use `else` and `else-if` within an `if` statement to extend the decision-making beyond a single condition.
- Short circuiting ensures that only the minimal required parts of a Boolean expression are evaluated.
- You can use the ternary operator in place of simple `if` statements.
- Variables and constants belong to a certain scope, beyond which you cannot use them. A scope inherits visible variables and constants from its parent.
- `while` loops allow you to perform a certain task a number of times until a condition is met.
- The `break` statement lets you break out of a loop.

Where to go from here?

Apps very rarely run all the way through the same way every time; depending on what data comes in from the Internet or from user input, your code will need to make decisions on which path to take. With `if` and `else`, you can have your code make decisions on what to do based on some condition.

In the next chapter, you'll see how to use more advanced control flow statements. This will involve more loops like the `while` loop you saw in this chapter, and a new construct called the `switch` statement.

Challenges

1. What's wrong with the following code?

```
let firstName = "Matt"

if firstName == "Matt" {
    let lastName = "Galloway"
} else if firstName == "Ray" {
    let lastName = "Wenderlich"
}

let fullName = firstName + " " + lastName
```

2. In each of the following statements, what is the value of the Boolean answer constant?

```
let answer = true && true
let answer = false || false
let answer = (true && 1 != 2) || (4 > 3 && 100 < 1)
let answer = ((10 / 2) > 3) && ((10 % 2) == 0)
```

3. Suppose the squares on a chessboard are numbered left to right, top to bottom, with 0 being the top-left square and 63 being the bottom-right square. Rows are numbered top to bottom, 0 to 7. Columns are numbered left to right, 0 to 7. Given a current position on the chessboard, expressed as a row and column number, calculate the next position on the chessboard, again expressed as a row and column number. The ordering is determined by the numbering from 0 to 63. The position after 63 is again 0.
4. Given the coefficients a, b and c, calculate the solutions to a quadratic equation with these coefficients. Take into account the different number of solutions (0, 1 or 2). If you need a math refresher, this Wikipedia article on the quadratic equation will help https://en.wikipedia.org/wiki/Quadratic_formula.
5. Given a month (represented with a String in all lowercase) and the current year (represented with an Int), calculate the number of days in the month. Remember that because of leap years, "february" has 29 days when the year is a multiple of 4 but not a multiple of 100. February also has 29 days when the year is a multiple of 400.
6. Given a number, determine if this number is a power of 2. (Hint: you can use `log2(number)` to find the base 2 logarithm of number. `log2(number)` will return a whole number if number is a power of two. You can also solve the problem using a loop and no logarithm.)
7. Print a table of the first 10 powers of 2.

8. Given a number n , calculate the n -th Fibonacci number. (Recall Fibonacci is 1, 1, 2, 3, 5, 8, 13, ... Start with 1 and 1 and add these values together to get the next value. The next value is the sum of the previous two. So the next value in this case is $8+13 = 21$.)
9. Given a number n , calculate the factorial of n . (Example: 4 factorial is equal to $1 * 2 * 3 * 4$.)
10. Given a number between 2 and 12, calculate the odds of rolling this number using two six-sided dice. Compute it by exhaustively looping through all of the combinations and counting the fraction of outcomes that give you that value. Don't use a formula.

Chapter 11: Advanced Control Flow

By Matt Galloway

In the previous chapter, you learned how to control the flow of execution using the decision-making powers of `if` statements and the `while` loop. In this chapter, you'll continue to learn how to control the flow of execution. You'll learn about another loop known as the `for` loop.

Loops may not sound very interesting, but they're very common in computer programs. For example, you might have code to download an image from the cloud; with a loop, you could run that multiple times to download your entire photo library. Or if you have a game with multiple computer-controlled characters, you might need a loop to go through each one and make sure it knows what to do next.

You'll also learn about `switch` statements, which are particularly powerful in Swift. They let you inspect a value and decide what to do based on that value. They're incredibly powerful when used with some advanced Swift features such as pattern matching.

Countable Ranges

Before you dive into the `for` loop statement, you need to know about the **Countable Range** data types, which let you represent a sequence of countable integers. Let's look at two types of ranges.

First, there's **countable closed range**, which you represent like so:

```
let closedRange = 0...5
```

The three dots (`...`) indicate that this range is closed, which means the range goes from 0 to 5 inclusive. That's the numbers (0, 1, 2, 3, 4, 5).

Second, there's **countable half-open range**, which you represent like so:

```
let halfOpenRange = 0..<5
```

Here, you replace the three dots with two dots and a less-than sign (**..**<****). Half-open means the range goes from 0 up to, but not including, 5. That's the numbers (0, 1, 2, 3, 4).

Both open and half-open ranges must always be increasing. In other words, the second number must always be greater than or equal to the first.

Countable ranges are commonly used in both **for** loops and **switch** statements, which means that throughout the rest of the chapter, you'll use ranges as well!

For loops

In the previous chapter you looked at **while** loops. Now that you know about ranges, it's time to look at another type of loop: the **for loop**. This is probably the most common loop you'll see, and you'll use it to run code a certain number of times.

You construct a **for** loop like this:

```
for <CONSTANT> in <COUNTABLE RANGE> {  
    <LOOP CODE>  
}
```

The loop begins with the **for** keyword, followed by a name given to the loop constant (more on that shortly), followed by **in**, followed by the range to loop through.

Here's an example:

```
let count = 10  
var sum = 0  
for i in 1...count {  
    sum += i  
}
```

In the code above, the **for** loop iterates through the range 1 to **count**. At the first iteration, **i** will equal the first element in the range: 1. Each time around the loop, **i** will increment until it's equal to **count**; the loop will execute one final time and then finish.

Note: If you'd used a half-open range, the the last iteration would see `i` equal to `count - 1`.

Inside the loop, you add `i` to the `sum` variable; it runs 10 times to calculate the sequence `1 + 2 + 3 + 4 + 5 + ...` all the way up to 10.

Here are the values of the constant `i` and variable `sum` for each iteration:

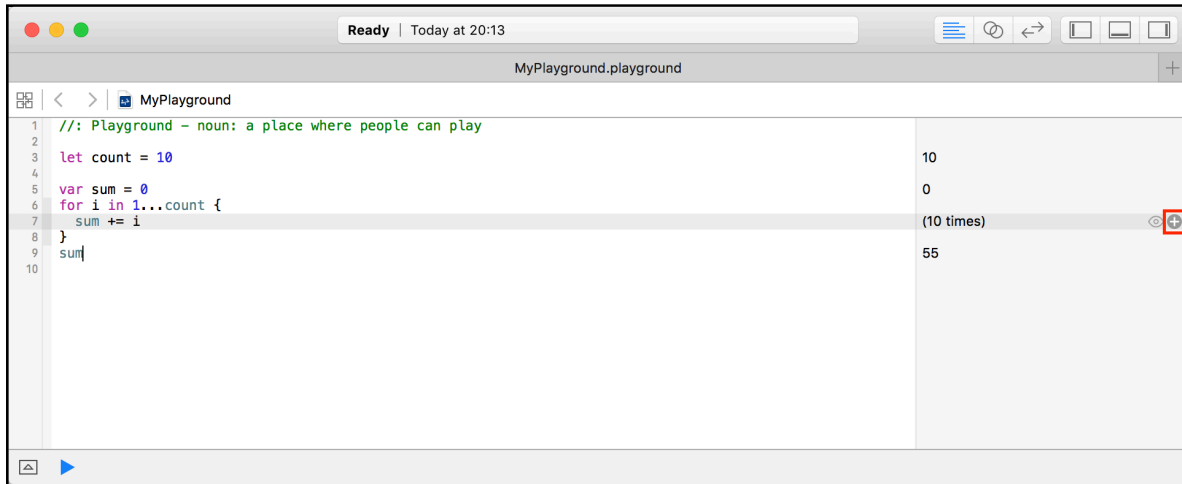
- **Start of iteration 1:** `i = 1`, `sum = 0`
- **Start of iteration 2:** `i = 2`, `sum = 1`
- **Start of iteration 3:** `i = 3`, `sum = 3`
- **Start of iteration 4:** `i = 4`, `sum = 6`
- **Start of iteration 5:** `i = 5`, `sum = 10`
- **Start of iteration 6:** `i = 6`, `sum = 15`
- **Start of iteration 7:** `i = 7`, `sum = 21`
- **Start of iteration 8:** `i = 8`, `sum = 28`
- **Start of iteration 9:** `i = 9`, `sum = 36`
- **Start of iteration 10:** `i = 10`, `sum = 45`
- **After iteration 10:** `sum = 55`

In terms of scope, the `i` constant is only visible inside the scope of the `for` loop, which means it's not available outside of the loop.

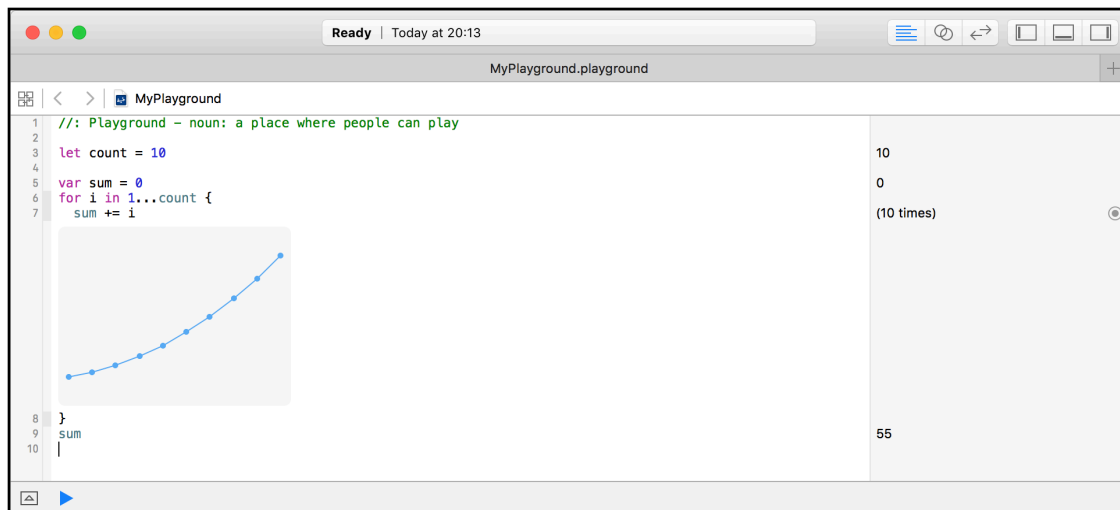
Note: If you're mathematically astute, you might notice that this example computes **triangle numbers**. Here's a quick explanation: <http://bbc.in/1O89TGP>

Xcode's playground gives you a handy way to visualize such an iteration. Hover over the `sum += i` line in the results pane, and you'll see a white dot on the right.

Hover over that dot to reveal a plus (+) button:



Click this plus (+) button and Xcode will display a graph underneath the line within the playground code editor:



This graph lets you visualize the sum variable as the loop iterates.

Finally, sometimes you only want to loop a certain number of times, and so you don't need to use the loop constant at all. In that case, you can employ the underscore to indicate you're ignoring it, like so:

```

sum = 1
var lastSum = 0

for _ in 0..

```

This code doesn't require a loop constant; the loop simply needs to run a certain number of times. In this case, the range is 0 up to, but not including, `count` and is half-open. This is the usual way of writing loops that run a certain number of times.

It's also possible to only perform the iteration under certain conditions. For example, imagine you wanted to compute a sum similar to that of triangle numbers, but only for odd numbers:

```
sum = 0
for i in 1...count where i % 2 == 1 {
    sum += i
}
```

The previous loop has a `where` clause in the `for` loop statement. The loop still runs through all values in the range 1 to `count`, but it will only execute the loop's code block when the `where` condition is true; in this case, where `i` is odd.

Continue and labeled statements

Sometimes you'd like to skip a loop iteration for a particular case without breaking out of the loop entirely. You can do this with the `continue` statement, which immediately ends the current iteration of the loop and starts the next iteration.

Note: In many cases, you can use the simpler `where` clause you just learned about. The `continue` statement gives you a higher level of control, letting you decide where and when you want to skip an iteration.

Take the example of an 8 by 8 grid, where each cell holds a value of the row multiplied by the column. It looks much like a multiplication table, doesn't it?

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	9	12	15	18	21
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	25	30	35
6	0	6	12	18	24	30	36	42
7	0	7	14	21	28	35	42	49

Let's say you wanted to calculate the sum of all cells but exclude all even rows, as shown below:

	0	1	2	3	4	5	6	7
0								
1	0	1	2	3	4	5	6	7
2								
3	0	3	6	9	12	15	18	21
4								
5	0	5	10	15	20	25	30	35
6								
7	0	7	14	21	28	35	42	49

Using a for loop, you can achieve this as follows:

```
sum = 0
for row in 0..<8 {
    if row % 2 == 0 {
        continue
    }

    for column in 0..<8 {
        sum += row * column
    }
}
```

When the row modulo 2 equals 0, the row is even. In this case, `continue` makes the for loop skip to the next row.

Just like `break`, `continue` works with both for loops and while loops.

The second code example will calculate the sum of all cells, excluding those where the column is greater than or equal to the row.

To illustrate, it should sum the following cells:

	0	1	2	3	4	5	6	7
0								
1	0							
2	0	2						
3	0	3	6					
4	0	4	8	12				
5	0	5	10	15	20			
6	0	6	12	18	24	30		
7	0	7	14	21	28	35	42	

Using a for loop, you can achieve this as follows:

```
sum = 0
rowLoop: for row in 0..<8 {
    columnLoop: for column in 0..<8 {
        if row == column {
            continue rowLoop
        }
        sum += row * column
    }
}
```

The previous code block makes use of **labeled statements**, labeling the two loops as rowLoop and the columnLoop, respectively. When the row equals the column inside the inner columnLoop, the outer rowLoop will continue.

You can use labeled statements like these with break to break out of a certain loop. Normally, break and continue work on the innermost loop, so you need to use labeled statements if you want to manipulate an outer loop.

Mini-exercises

1. Create a constant named range and set it equal to a range starting at 1 and ending with 10 inclusive. Write a for loop which iterates over this range and prints the square of each number.
2. Write a for loop to iterate over the same range as in the exercise above and print the square root of each number. You'll need to type convert your loop constant.

3. Above, you saw a for loop which iterated over only the even rows like so:

```
sum = 0
for row in 0..<8 {
    if row % 2 == 0 {
        continue
    }
    for column in 0..<8 {
        sum += row * column
    }
}
```

Change this to use a where clause on the first for loop to skip even rows instead of using continue. Check that the sum is 448 as in the initial example.

Switch statements

Another way to control flow is through the use of a switch statement, which lets you execute different bits of code depending on the value of a variable or constant. Here's a very simple switch statement that acts on an integer:

```
let number = 10

switch number {
case 0:
    print("Zero")
default:
    print("Non-zero")
}
```

In this example, the code will print the following:

```
Non-zero
```

The purpose of this switch statement is to determine whether or not a number is zero. It will get more complex — I promise!

To handle a specific case, you use case followed by the value you want to check for, which in this case is 0. Then, you use default to signify what should happen for all other values.

Here's another example:

```
switch number {
case 10:
    print("It's ten!")
default:
```

```
    break  
}
```

This time you check for 10, in which case, you print a message. Nothing should happen for other values. When you want nothing to happen for a case, you use the `break` statement. This tells Swift that you *meant* to not write any code here and that nothing should happen. Cases can never be empty, so you *must* write some code, even if it's just a `break`!

Of course, `switch` statements also work with data types other than integers. They work with any data type!

Here's an example of switching on a string:

```
let string = "Dog"  
  
switch string {  
case "Cat", "Dog":  
    print("Animal is a house pet.")  
default:  
    print("Animal is not a house pet.")  
}
```

This will print the following:

```
Animal is a house pet.
```

In this example, you provide two values for the case, meaning that if the value is equal to either "Cat" or "Dog", then the statement will execute the case.

Advanced switch statements

You can also give your `switch` statements more than one case. In the previous chapter, you saw an `if` statement that used multiple `else` clauses to convert an hour of the day to a string describing that part of the day. You could rewrite that more succinctly with a `switch` statement, like so:

```
let hourOfDay = 12  
let timeOfDay: String  
  
switch hourOfDay {  
case 0, 1, 2, 3, 4, 5:  
    timeOfDay = "Early morning"  
case 6, 7, 8, 9, 10, 11:  
    timeOfDay = "Morning"  
case 12, 13, 14, 15, 16:  
    timeOfDay = "Afternoon"  
case 17, 18, 19:
```

```
timeOfDay = "Evening"
case 20, 21, 22, 23:
    timeOfDay = "Late evening"
default:
    timeOfDay = "INVALID HOUR!"
}

print(timeOfDay)
```

This code will print the following:

Afternoon

Remember ranges? Well, you can use ranges to simplify this `switch` statement. You can rewrite the above code using ranges:

```
switch hourOfDay {
case 0...5:
    timeOfDay = "Early morning"
case 6...11:
    timeOfDay = "Morning"
case 12...16:
    timeOfDay = "Afternoon"
case 17...19:
    timeOfDay = "Evening"
case 20...<24:
    timeOfDay = "Late evening"
default:
    timeOfDay = "INVALID HOUR!"
}
```

This is more succinct than writing out each value individually for all cases.

When there are multiple cases, the statement will execute the first one that matches. You'll probably agree that this is more succinct and more clear than using an `if` statement for this example.

It's slightly more precise as well, because the `if` statement method didn't address negative numbers, which here are correctly deemed to be invalid.

It's also possible to match a case to a condition based on a property of the value. As you learned in Chapter 2, you can use the modulo operator to determine if an integer is even or odd. Consider this code:

```
switch number {
case let x where x % 2 == 0:
    print("Even")
default:
    print("Odd")
}
```

This will print the following:

Even

This switch statement uses the `let-where` syntax, meaning the case will match only when a certain condition is true. The `let` part binds a value to a name, while the `where` part provides a Boolean condition that must be true for the case to match.

In this example, you've designed the case to match if the value is even — that is, if the value modulo 2 equals 0.

The method by which you can match values based on conditions is known as **pattern matching**.

In the previous example, the binding introduced a unnecessary constant `x`; it's simply another name for `number`.

You are allowed to use `number` in the `where` clause and replace the binding with an underscore to ignore it:

```
switch number {
case _ where number % 2 == 0:
    print("Even")
default:
    print("Odd")
}
```

Partial matching

Another way you can use switch statements with matching to great effect is as follows:

```
let coordinates = (x: 3, y: 2, z: 5)

switch coordinates {
case (0, 0, 0): // 1
    print("Origin")
case (_, 0, 0): // 2
    print("On the x-axis.")
case (0, _, 0): // 3
    print("On the y-axis.")
case (0, 0, _): // 4
    print("On the z-axis.")
default: // 5
    print("Somewhere in space")
}
```

This switch statement makes use of **partial matching**. Here's what each case does, in order:

1. Matches precisely the case where the value is `(0, 0, 0)`. This is the origin of 3D space.
2. Matches `y=0, z=0` and any value of `x`. This means the coordinate is on the x-axis.
3. Matches `x=0, z=0` and any value of `y`. This means the coordinate is on the y-axis.
4. Matches `x=0, y=0` and any value of `z`. This means the coordinate is on the z-axis.
5. Matches the remainder of coordinates.

You're using the underscore to mean that you don't care about the value. If you don't want to ignore the value, then you can bind it and use it in your switch statement.

Here's an example of how to do this:

```
switch coordinates {
case (0, 0, 0):
    print("Origin")
case (let x, 0, 0):
    print("On the x-axis at x = \(x)")
case (0, let y, 0):
    print("On the y-axis at y = \(y)")
case (0, 0, let z):
    print("On the z-axis at z = \(z)")
case let (x, y, z):
    print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Here, the axis cases use the `let` syntax to pull out the pertinent values. The code then prints the values using string interpolation to build the string.

Notice how you don't need a default in this switch statement. This is because the final case is essentially the default; it matches anything, because there are no constraints on any part of the tuple. If the switch statement exhausts all possible values with its cases, then no default is necessary.

Also notice how you could use a single `let` to bind all values of the tuple: `let (x, y, z)` is the same as `(let x, let y, let z)`.

Finally, you can use the same `let-where` syntax you saw earlier to match more complex cases. For example:

```
switch coordinates {
case let (x, y, _) where y == x:
    print("Along the y = x line.")
case let (x, y, _) where y == x * x:
```

```
print("Along the y = x^2 line.")
default:
    break
}
```

Here, you match the “y equals x” and “y equals x squared” lines.

And those are the basics of switch statements!

Mini-exercises

1. Write a switch statement that takes an age as an integer and prints out the life stage related to that age. You can make up the life stages, or use my categorization as follows: 0-2 years, Infant; 3-12 years, Child; 13-19 years, Teenager; 20-39, Adult; 40-60, Middle aged; 61+, Elderly.
2. Write a switch statement that takes a tuple containing a string and an integer. The string is a name, and the integer is an age. Use the same cases that you used in the previous exercise and let syntax to print out the name followed by the life stage. For example, for myself it would print out "Matt is an adult."

Key points

- You can use **ranges** to create a sequence of numbers, incrementing to move from one value to another.
- **Closed ranges** include both the start and end values.
- **Half-open ranges** include the start value and stop one before the end value.
- **For loops** allow you to iterate over a range.
- The **continue** statement lets you finish the current iteration of a loop and begin the next iteration.
- **Labeled statements** let you use break and continue on an outer loop.
- You use **switch** statements to decide which code to run depending on the value of a variable or constant.
- The power of a switch statement comes from leveraging **pattern matching** to compare values using complex rules.

Where to go from here?

You've learned about the core language features for dealing with data over these past few chapters, from data types to variables, then on to decision-making with Booleans and loops with ranges. In the next chapter you'll learn one of the key ways to make your code more reusable and easy to read through the use of functions.

Challenges

1. In the following for loop, what will be the value of `sum`, and how many iterations will happen?

```
var sum = 0
for i in 0...5 {
    sum += i
}
```

2. In the while loop below, how many instances of "a" will there be in `aLotOfAs`? Hint: `aLotOfAs.count` tells you how many characters are in the string `aLotOfAs`.

```
var aLotOfAs = ""
while aLotOfAs.count < 10 {
    aLotOfAs += "a"
}
```

3. Consider the following switch statement:

```
switch coordinates {
case let (x, y, z) where x == y && y == z:
    print("x = y = z")
case (_, _, 0):
    print("On the x/y plane")
case (_, 0, _):
    print("On the x/z plane")
case (0, _, _):
    print("On the y/z plane")
default:
    print("Nothing special")
}
```

What will this code print when `coordinates` is each of the following?

```
let coordinates = (1, 5, 0)
let coordinates = (2, 2, 2)
let coordinates = (3, 0, 1)
let coordinates = (3, 2, 5)
let coordinates = (0, 2, 4)
```

4. A closed range can never be empty. Why?
5. Print a countdown from 10 to 0. (Note: do not use the `reversed()` method, which will be introduced later.)
6. Print 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0. (Note: do not use the `stride(from:by:to:)` function, which will be introduced later.)

Chapter 12: Functions

By Matt Galloway

Functions are a core part of many programming languages. Simply put, a function lets you define a block of code that performs a task. Then, whenever your app needs to execute that task, you can run the function instead of having to copy and paste the same code everywhere.

In this chapter, you'll learn how to write your own functions, and see firsthand how Swift makes them easy to use.

Function basics

Imagine you have an app that frequently needs to print your name. You can write a function to do this:

```
func printMyName() {  
    print("My name is Matt Galloway.")  
}
```

The code above is known as a **function declaration**. You define a function using the `func` keyword. After that comes the name of the function, followed by parentheses. You'll learn more about the need for these parentheses in the next section.

After the parentheses comes an opening brace, followed by the code you want to run in the function, followed by a closing brace. With your function defined, you can use it like so:

```
printMyName()
```

This prints out the following:

```
My name is Matt Galloway.
```

If you suspect that you’ve already used a function in previous chapters, you’re correct! `print`, which prints the text you give it to the console, is indeed a function.

This leads nicely into the next section, in which you’ll learn how to pass data to a function and get data back in return.

Function parameters

In the previous example, the function simply prints out a message. That’s great, but sometimes you want to **parameterize** your function, which lets the function perform differently depending on the data passed into it via its **parameters**.

As an example, consider the following function:

```
func printMultipleOfFive(value: Int) {  
    print("\(value) * 5 = \(value * 5)")  
}  
printMultipleOfFive(value: 10)
```

Here, you can see the definition of one parameter inside the parentheses after the function name, named `value` and of type `Int`. In any function, the parentheses contain what’s known as the **parameter list**. These parentheses are required both when declaring and when invoking the function, even if the parameter list is empty.

This function will print out any given multiple of five. In the example, you call the function with an **argument** of 10, so the function prints the following:

```
10 * 5 = 50
```

Note: Take care not to confuse the terms “parameter” and “argument”. A function declares its *parameters* in its parameter list. When you call a function, you provide values as *arguments* for the functions parameters.

You can take this one step further and make the function more general. With two parameters, the function can print out a multiple of any two values:

```
func printMultipleOf(multiplier: Int, andValue: Int) {  
    print("\(multiplier) * \(andValue) = \(multiplier * andValue)")  
}
```

```
printMultipleOf(multiplier: 4, andValue: 2)
```

There are now two parameters inside the parentheses after the function name: one named `multiplier` and the other named `andValue`, both of type `Int`.

Notice that you need to apply the labels in the parameter list to the arguments when you call a function. In the example above you need to put `multiplier:` before the `multiplier` and `andValue:` before the value to be multiplied.

In Swift, you should try to make your function calls read like a sentence. In the example above, you would read the last line of code like this:

Print multiple of multiplier 4 and value 2

You can make this even clearer by giving a parameter a different external name. For example, you can change the name of the `andValue` parameter:

```
func printMultipleOf(multiplier: Int, and value: Int) {  
    print("\(multiplier) * \(value) = \(multiplier * value)")  
}  
printMultipleOf(multiplier: 4, and: 2)
```

You assign a different external name by writing it in front of the parameter name. In this example, the internal name of the parameter is now `value` while the external name (the argument label) in the function call is now `and`. You can read the new call as:

Print multiple of multiplier 4 and 2

If you want to have no external name at all, then you can employ the underscore `_`, as you've seen in previous chapters:

```
func printMultipleOf(_ multiplier: Int, and value: Int) {  
    print("\(multiplier) * \(value) = \(multiplier * value)")  
}  
printMultipleOf(4, and: 2)
```

This makes it even more readable. The function call now reads like so:

Print multiple of 4 and 2

You could, if you so wished, take this even further and use `_` for all parameters, like so:

```
func printMultipleOf(_ multiplier: Int, _ value: Int) {  
    print("\(multiplier) * \(value) = \(multiplier * value)")  
}  
printMultipleOf(4, 2)
```

In this example, all parameters have no external name. But this illustrates how you use the underscore wisely. Here, your expression is still understandable, but more complex functions that take many parameters can become confusing and unwieldy with no external parameter names. Imagine if a function took five parameters!

You can also give default values to parameters:

```
func printMultipleOf(_ multiplier: Int, _ value: Int = 1) {  
    print("\(multiplier) * \(value) = \(multiplier * value)")  
}  
printMultipleOf(4)
```

The difference is the `= 1` after the second parameter, which means that if no value is provided for the second parameter, it defaults to 1.

Therefore, this code prints the following:

```
4 * 1 = 4
```

It can be useful to have a default value when you expect a parameter to be one particular value the majority of the time, and it will simplify your code when you call the function.

Return values

All of the functions you've seen so far have performed a simple task: printing something out. Functions can also return a value. The caller of the function can assign the return value to a variable or constant, or use it directly in an expression.

This means you can use a function to manipulate data. You simply take in data through parameters, manipulate it and then return it. Here's how you define a function that returns a value:

```
func multiply(_ number: Int, by multiplier: Int) -> Int {  
    return number * multiplier  
}  
let result = multiply(4, by: 2)
```

To declare that a function returns a value, you add a `->` followed by the type of the return value after the set of parentheses and before the opening brace. In this example, the function returns an `Int`.

Inside the function, you use a `return` statement to return the value. In this example, you return the product of the two parameters.

It's also possible to return multiple values through the use of tuples:

```
func multiplyAndDivide(_ number: Int, by factor: Int)
    -> (product: Int, quotient: Int) {
    return (number * factor, number / factor)
}
let results = multiplyAndDivide(4, by: 2)
let product = results.product
let quotient = results.quotient
```

This function returns *both* the product and quotient of the two parameters: It returns a tuple containing two `Int` values with appropriate member value names.

The ability to return multiple values through tuples is one of the many things that makes it such a pleasure to work with Swift. And it turns out to be a very useful feature, as you'll see shortly.

Advanced parameter handling

Function parameters are constants by default, which means they can't be modified.

To illustrate this point, consider the following code:

```
func incrementAndPrint(_ value: Int) {
    value += 1
    print(value)
}
```

This results in an error:

```
Left side of mutating operator isn't mutable: 'value' is a 'let' constant
```

The parameter `value` is the equivalent of a constant declared with `let`. Therefore, when the function attempts to increment it, the compiler emits an error.

An important point to note is that Swift copies the value before passing it to the function, a behavior known as **pass-by-value**.

Note: Pass-by-value and making copies is the standard behavior for all of the types you’ve seen so far in this book. You’ll see another way for things to be passed into functions in Chapter 14, “Classes”.

Usually you want this behavior. Ideally, a function doesn’t alter its parameters. If it did, then you couldn’t be sure of the parameters’ values and you might make incorrect assumptions in your code, leading to the wrong data.

Sometimes you *do* want to let a function change a parameter directly, a behavior known as **copy-in copy-out** or **call by value result**. You do it like so:

```
func incrementAndPrint(_ value: inout Int) {  
    value += 1  
    print(value)  
}
```

`inout` before the parameter type indicates that this parameter should be copied in, that local copy used within the function, and copied back out when the function returns.

You need to make a slight tweak to the function call to complete this example. Add an ampersand (&) before the argument, which makes it clear at the call site that you are using copy-in copy-out:

```
var value = 5  
incrementAndPrint(&value)  
print(value)
```

Now the function can change the value however it wishes.

This example will print the following:

```
6  
6
```

The function increments `value`, which retains its modified data after the function finishes. The value goes *in* to the function and comes back *out* again, thus the keyword `inout`.

Under certain conditions, the compiler can simplify copy-in copy-out to what is called *pass-by-reference*. The argument value isn’t copied into the parameter. Instead, the parameter will hold a reference to the memory of original value. This optimization satisfies all requirements of copy-in copy-out while removing the need for copies.

Overloading

Did you notice how you used the same function name for several different functions in the previous examples?

```
func printMultipleOf(multiplier: Int, andValue: Int)
func printMultipleOf(multiplier: Int, and value: Int)
func printMultipleOf(_ multiplier: Int, and value: Int)
func printMultipleOf(_ multiplier: Int, _ value: Int)
```

This is called **overloading** and lets you define similar functions using a single name.

However, the compiler must still be able to tell the difference between these functions. Whenever you call a function, it should always be clear which function you're calling. This is usually achieved through a difference in the parameter list:

- A different number of parameters.
- Different parameter types.
- Different external parameter names, such as the case with `printMultipleOf`.

You can also overload a function name based on a different return type, like so:

```
func getValue() -> Int {
    return 31
}

func getValue() -> String {
    return "Matt Galloway"
}
```

Here, there are two functions called `getValue()`, which return different types. One an `Int` and the other a `String`.

Using these is a little more complicated. Consider the following:

```
let value = getValue()
```

How does Swift know which `getValue()` to call? The answer is, it doesn't. And it will print the following error:

```
error: ambiguous use of 'getValue()'
```

There's no way of knowing which one to call. It's a chicken and egg situation. It's unknown what type `value` is, so Swift doesn't know which `getValue()` to call or what the return type of `getValue()` should be.

To fix this, you can declare what type you want `value` to be, like so:

```
let valueInt: Int = getValue()  
let valueString: String = getValue()
```

This will correctly call the `Int` version of `getValue()` in the first instance, and the `String` version of `getValue()` in the second instance.

It's worth noting that overloading should be used with care. Only use overloading for functions that are related and similar in behavior. When only the return type is overloaded, as in the above example, you lose type inference and so is not recommended.

Mini-exercises

1. Write a function named `printFullName` that takes two strings called `firstName` and `lastName`. The function should print out the full name defined as `firstName + " " + lastName`. Use it to print out your own full name.
2. Change the declaration of `printFullName` to have no external name for either parameter.
3. Write a function named `calculateFullName` that returns the full name as a string. Use it to store your own full name in a constant.
4. Change `calculateFullName` to return a tuple containing both the full name and the length of the name. You can find a string's length by using the `count` property. Use this function to determine the length of your own full name.

Functions as variables

This may come as a surprise, but functions in Swift are simply another data type. You can assign them to variables and constants just as you can any other type of value, such as an `Int` or a `String`.

To see how this works, consider the following function:

```
func add(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

This function takes two parameters and returns the sum of their values.

You can assign this function to a variable, like so:

```
var function = add
```

Here, the name of the variable is `function` and its type is inferred as `(Int, Int) -> Int` from the `add` function you assign to it.

Notice how the function type `(Int, Int) -> Int` is written in the same way you write the parameter list and return type in a function declaration. Here, the function variable is of a function type that takes two `Int` parameters and returns an `Int`.

Now you can use the function variable in just the same way you'd use `add`, like so:

```
function(4, 2)
```

This returns 6.

Now consider the following code:

```
func subtract(_ a: Int, _ b: Int) -> Int {  
    return a - b  
}
```

Here, you declare another function that takes two `Int` parameters and returns an `Int`. You can set the function variable from before to your new `subtract` function, because the parameter list and return type of `subtract` are compatible with the type of the function variable.

```
function = subtract  
function(4, 2)
```

This time, the call to `function` returns 2.

The fact that you can assign functions to variables comes in handy because it means you can pass functions to other functions. Here's an example of this in action:

```
func printResult(_ function: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    let result = function(a, b)  
    print(result)  
}  
printResult(add, 4, 2)
```

`printResult` takes three parameters:

1. `function` is of a function type that takes two `Int` parameters and returns an `Int`, declared like so: `(Int, Int) -> Int`.
2. `a` is of type `Int`.

3. `b` is of type `Int`.

`printResult` calls the passed-in function, passing into it the two `Int` parameters. Then it prints the result to the console:

```
6
```

It's extremely useful to be able to pass functions to other functions, and it can help you write reusable code. Not only can you pass data around to manipulate, but passing functions as parameters also means you can be flexible about what code executes.

The land of no return

There are some functions which are designed to never, ever, return. This may sound confusing, but consider the example of a function that is designed to crash an application. This may sound strange, but if an application is about to work with corrupt data, it's often best to crash rather than continue in an unknown and potentially dangerous state.

Another example of a non-returning function is one which handles an event loop. An event loop is at the heart of every modern application which takes input from the user and displays things on a screen. The event loop services requests coming from the user, then passes these events to the application code, which in turn causes the information to be displayed on the screen. The loop then cycles back and services the next event.

These event loops are often started in an application by calling a function which is known to never return. Once you're coding iOS or Mac apps, think back to this paragraph when you encounter `UIApplicationMain` or `NSApplicationMain`.

Swift has a way to tell the compiler that a function is known to never return. It is done like so:

```
func noReturn() -> Never {  
}
```

Notice the special return type `Never`. This is what indicates that this function will never return.

If you wrote this code you would get the following error:

```
Return from a 'noreturn' function
```

Understandable! Thanks Swift!

A crude, but honest, implementation of a function that wouldn't return would be as follows:

```
func infiniteLoop() -> Never {  
    while true {  
    }  
}
```

You may be wondering why bother with this special return type. It's useful because by the compiler knowing that the function won't ever return, it can make certain optimizations when generating the code to call the function. Essentially, the code which calls the function doesn't need to bother doing anything after the function call, because it knows that this function will never end before the application is terminated.

Writing good functions

There are many ways to solve problems with functions. The best (easiest to use and understand) functions do *one simple task* rather than trying to do many. This makes them easier to mix and match and assemble into more complex behaviors. Good functions also have a well defined set of inputs that produce the same output every time. This makes them easier to reason about and test in isolation. Keep these rules-of-thumb in mind as you create functions.

Key points

- You use a **function** to define a task, which you can execute as many times as you like without having to write the code multiple times.
- Functions can take zero or more **parameters** and optionally return a value.
- You can add an external name to a function parameter to change the label you use in a function call, or you can use an underscore to denote no label.
- Parameters are passed as constants, unless you mark them as `inout`, in which case they are copied-in and copied-out.
- Functions can have the same name with different parameters. This is called **overloading**.
- Functions can have a special `Never` return type to inform Swift that this function will never exit.

- You can assign functions to variables and pass them to other functions.
- Strive to create functions that are clearly named and have one job with repeatable inputs and outputs.

Where to go from here?

Functions are the first step in grouping small pieces of code together into a larger unit.

In the next chapter you'll learn about **optionals**, which are an important part of Swift's syntactic arsenal. Before you move on, check out the challenges ahead as you'll need to understand functions before understanding some of the upcoming topics!

Challenges

Challenge 1: Looping with stride functions

In the last chapter you wrote some for loops with countable ranges. Countable ranges are limited in that they must always be increasing by one. The Swift `stride(from:to:by:)` and `stride(from:through:by:)` functions let you loop much more flexibly. For example, if you wanted to loop from 10 to 20 by 4's you can write:

```
for index in stride(from: 10, to: 22, by: 4) {  
    print(index)  
}  
// prints 10, 14, 18  
  
for index in stride(from: 10, through: 22, by: 4) {  
    print(index)  
}  
// prints 10, 14, 18, and 22
```

- What is the difference between the two stride function overloads?
- Write a loop that goes from 10.0 to (and including) 9.0, decrementing by 0.1.

Challenge 2: It's prime time

When I'm acquainting myself with a programming language, one of the first things I do is write a function to determine whether or not a number is prime. That's your second challenge.

First, write the following function:

```
func isNumberDivisible(_ number: Int, by divisor: Int) -> Bool
```

You'll use this to determine if one number is divisible by another. It should return `true` when number is divisible by divisor.

Hint: You can use the modulo (%) operator to help you out here.

Next, write the main function:

```
func isPrime(_ number: Int) -> Bool
```

This should return `true` if number is prime, and `false` otherwise. A number is prime if it's only divisible by 1 and itself. You should loop through the numbers from 1 to the number and find the number's divisors. If it has any divisors other than 1 and itself, then the number isn't prime. You'll need to use the `isNumberDivisible(_:by:)` function you wrote earlier.

Use this function to check the following cases:

```
isPrime(6) // false
isPrime(13) // true
isPrime(8893) // true
```

Hint 1: Numbers less than 0 should not be considered prime. Check for this case at the start of the function and return early if the number is less than 0.

Hint 2: Use a `for` loop to find divisors. If you start at 2 and end before the number itself, then as soon as you find a divisor, you can return `false`.

Hint 3: If you want to get *really* clever, you can simply loop from 2 until you reach the square root of number, rather than going all the way up to number itself. I'll leave it as an exercise for you to figure out why. It may help to think of the number 16, whose square root is 4. The divisors of 16 are 1, 2, 4, 8 and 16.

Challenge 3: Recursive functions

In this challenge, you're going to see what happens when a function calls *itself*, a behavior called **recursion**. This may sound unusual, but it can be quite useful.

You're going to write a function that computes a value from the **Fibonacci sequence**. Any value in the sequence is the sum of the previous two values. The sequence is defined such that the first two values equal 1. That is, `fibonacci(1) = 1` and `fibonacci(2) = 1`.

Write your function using the following declaration:

```
func fibonacci(_ number: Int) -> Int
```

Then, verify you've written the function correctly by executing it with the following numbers:

```
fibonacci(1)  // = 1
fibonacci(2)  // = 1
fibonacci(3)  // = 2
fibonacci(4)  // = 3
fibonacci(5)  // = 5
fibonacci(10) // = 55
```

Hint 1: For values of number less than 0, you should return 0.

Hint 2: To start the sequence, hard-code a return value of 1 when number equals 1 or 2.

Hint 3: For any other value, you'll need to return the sum of calling `fibonacci` with `number - 1` and `number - 2`.

Chapter 13: Optionals

By Matt Galloway

All the variables and constants you’ve dealt with so far have had concrete values. When you had a string variable, like `var name`, it had a string value associated with it, like `"Matt Galloway"`. It could have been an empty string, like `""`, but nevertheless, there was a value to which you could refer.

That’s one of the built-in safety features of Swift: If the type says `Int` or `String`, then there’s an actual integer or string there, guaranteed.

This chapter will introduce you to the concept of **optionals**, a special Swift type that can represent not just a value, but also the absence of a value. By the end of this chapter, you’ll know why you need optionals and how to use them safely.

Introducing nil

Sometimes, it’s useful to be able to represent the absence of a value. Imagine a scenario where you need to refer to a person’s identifying information; you want to store the person’s name, age and occupation. Name and age are both things that must have a value — everyone has them. But not everyone is employed, so the absence of a value for occupation is something you need to be able to handle.

Without knowing about optionals, this is how you might represent the person’s name, age and occupation:

```
var name = "Matt Galloway"  
var age = 30  
var occupation = "Software Developer & Author"
```

But what if I become unemployed? Maybe I've won the lottery and want to give up work altogether (I wish!). This is when it would be useful to be able to refer to the absence of a value.

Why couldn't you just use an empty string? You *could*, but optionals are a much better solution. Read on to see why.

Sentinel values

A valid value that represents a special condition such as the absence of a value is known as a **sentinel value**. That's what your empty string would be in the previous example.

Let's look at another example. Say your code requests something from a server, and you use a variable to store any returned error code:

```
var errorCode = 0
```

In the success case, you represent the lack of an error with a zero. That means 0 is a sentinel value.

Just like the empty string for occupation, this works, but it's potentially confusing for the programmer. 0 might actually be a valid error code — or could be in the future, if the server changed how it responded. Either way, you can't be completely confident that the server didn't return an error without consulting the documentation.

In these two examples, it would be much better if there were a special type that could represent the absence of a value. It would then be explicit when a value exists and when one doesn't.

Nil is the name given to the absence of a value, and you're about to see how Swift incorporates this concept directly into the language in a rather elegant way.

Some other programming languages simply use sentinel values. Some, like Objective-C, have the concept of `nil`, but it is merely a synonym for zero. It is just another sentinel value.

Swift introduces a whole new type, **optional**, that handles the possibility a value could be `nil`. If you're handling a non-optional type, then you're guaranteed to have a value and don't need to worry about the existence of a valid value. Similarly, if you are using an optional type then you know you must handle the `nil` case. It removes the ambiguity introduced by using sentinel values.

Introducing optionals

Optionals are Swift's solution to the problem of representing both a value and the absence of a value. An optional is allowed to hold either a value *or* `nil`.

Think of an optional as a box: it either contains a value, or it doesn't. When it doesn't contain a value, it's said to contain `nil`. The box itself always exists; it's always there for you to open and look inside.



Optional box
containing a
value



Optional box
containing no
value

A string or an integer, on the other hand, doesn't have this box around it. Instead there's always a value, such as "hello" or 42. Remember, non-optional types are guaranteed to have an actual value.

Note: Those of you who've studied physics may be thinking about Schroedinger's cat right now. Optionals are a little bit like that, except it's not a matter of life and death!

You declare a variable of an optional type by using the following syntax:

```
var errorCode: Int?
```

The only difference between this and a standard declaration is the question mark at the end of the type. In this case, `errorCode` is an "optional `Int`". This means the variable itself is like a box containing either an `Int` or `nil`.

Note: You can add a question mark after any type to create an optional type. This optional type is said to *wrap* the regular non-optional type. For example, optional type `String?` wraps type `String`. In other words: an optional box of type `String?` holds either a `String` or `nil`.

Also, note how an optional type must be made explicit using a type annotation (here `: Int?`). Optional types can never be inferred from initialization values, as those values are of a regular, non-optional type, or `nil`, which can be used with any optional type.

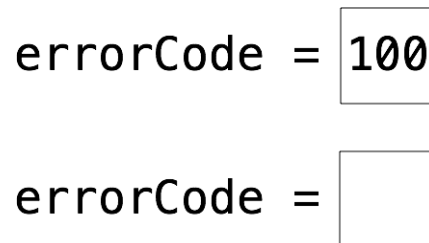
Setting the value is simple. You can either set it to an `Int`, like so:

```
errorCode = 100
```

Or you can set it to `nil`, like so:

```
errorCode = nil
```

This diagram may help you visualize what's happening:



The optional box always exists. When you assign `100` to the variable, you're filling the box with the value. When you assign `nil` to the variable, you're emptying the box.

Take a few minutes to think about this concept. The box analogy will be a big help as you go through the rest of the chapter and begin to use optionals.

Mini-exercises

1. Make an optional `String` called `myFavoriteSong`. If you have a favorite song, set it to a string representing that song. If you have more than one favorite song or no favorite, set the optional to `nil`.
2. Create a constant called `parsedInt` and set it equal to `Int("10")` which tries to parse the string `10` and convert it to an `Int`. Check the type of `parsedInt` using `OptionClick`. Why is it an optional?
3. Change the string being parsed in the above exercise to a non-integer (try `dog` for example). What does `parsedInt` equal now?

Unwrapping optionals

It's all well and good that optionals exist, but you may be wondering how you can look inside the box and manipulate the value it contains.

Take a look at what happens when you print out the value of an optional:

```
var result: Int? = 30
print(result)
```

This prints the following:

```
Optional(30)
```

Note: You will also see a warning on this line which says “Expression implicitly coerced from 'Int?' to Any”. This is because Swift warns that you're using an optional in the place of the Any type as it's something that usually means you did something wrong. To silence the warning here you can change the code to `print(result as Any)`.

That isn't really what you wanted — although if you think about it, it makes sense. Your code has printed the box. The result says, “result is an optional that contains the value 30”.

To see how an optional type is different from a non-optional type, see what happens if you try to use `result` as if it were a normal integer:

```
print(result + 1)
```

This code triggers an error:

```
Value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'?
```

It doesn't work because you're trying to add an integer to a box — not to the value inside the box, but to the box itself. That doesn't make sense!

Force unwrapping

The error message gives an indication of the solution: It tells you that the optional is not unwrapped. You need to unwrap the value from its box. It's like Christmas!

Let's see how that works. Consider the following declarations:

```
var authorName: String? = "Matt Galloway"
var authorAge: Int? = 30
```

There are two different methods you can use to unwrap these optionals. The first is known as **force unwrapping**, and you perform it like so:

```
var unwrappedAuthorName = authorName!
print("Author is \(unwrappedAuthorName)")
```

This code prints:

```
Author is Matt Galloway
```

Great! That's what you'd expect.

The exclamation mark after the variable name tells the compiler that you want to look inside the box and take out the value. The result is a value of the wrapped type. This means `unwrappedAuthorName` is of type `String`, not `String?`.

The use of the word “force” and the exclamation mark ! probably conveys a sense of danger to you, and it should. You should use force unwrapping sparingly. To see why, consider what happens when the optional doesn't contain a value:

```
authorName = nil
print("Author is \(authorName!)" )
```

This code produces the following runtime error:

```
fatal error: unexpectedly found nil while unwrapping an Optional value
```

The error occurs because the variable contains no value when you try to unwrap it. What's worse is that you get this error at runtime rather than compile time – which means you'd only notice the error if you happened to execute this code with some invalid input. Worse yet, if this code were inside an app, the runtime error would cause the app to crash!

How can you play it safe?

To stop the runtime error here, you could wrap the code that unwraps the optional in a check, like so:

```
if authorName != nil {
    print("Author is \(authorName!)" )
}
```

```
} else {  
    print("No author.")  
}
```

The `if` statement checks if the optional contains `nil`. If it doesn't, that means it contains a value you can unwrap.

The code is now safe, but it's still not perfect. If you rely on this technique, you'll have to remember to check for `nil` every time you want to unwrap an optional. That will start to become tedious, and one day you'll forget and once again end up with the possibility of a runtime error.

Back to the drawing board, then!

Optional binding

Swift includes a feature known as **optional binding**, which lets you safely access the value inside an optional. You use it like so:

```
if let unwrappedAuthorName = authorName {  
    print("Author is \(unwrappedAuthorName)")  
} else {  
    print("No author.")  
}
```

You'll immediately notice that there are no exclamation marks here. This optional binding gets rid of the optional type. If the optional contains a value, this value is unwrapped and stored in, or *bound to*, the constant `unwrappedAuthorName`. The `if` statement then executes the first block of code, within which you can safely use `unwrappedAuthorName`, as it's a regular non-optional `String`.

If the optional doesn't contain a value, then the `if` statement executes the `else` block. In that case, the `unwrappedAuthorName` variable doesn't even exist.

You can see how optional binding is much safer than force unwrapping, and you should use it whenever an optional might be `nil`. Force unwrapping is only appropriate when an optional *is guaranteed* contain a value.

Because naming things is so hard, it's common practice to give the unwrapped constant the same name as the optional (thereby *shadowing* that optional):

```
if let authorName = authorName {  
    print("Author is \(authorName)")  
} else {  
    print("No author.")  
}
```

You can even unwrap multiple values at the same time, like so:

```
if let authorName = authorName,
   let authorAge = authorAge {
    print("The author is \(authorName) who is \(authorAge) years old.")
} else {
    print("No author or no age.")
}
```

This code unwraps two values. It will only execute the `if` part of the statement when both optionals contain a value.

You can combine unwrapping multiple optionals with additional boolean checks. For example:

```
if let authorName = authorName,
   let authorAge = authorAge,
   authorAge >= 40 {
    print("The author is \(authorName) who is \(authorAge) years old.")
} else {
    print("No author or no age or age less than 40.")
}
```

Here, you unwrap name and age, and check that age is greater than or equal to 40. The expression in the `if` statement will only be `true` if name is non-`nil`, *and* age is non-`nil`, *and* age is greater than or equal to 40.

Now you know how to safely look inside an optional and extract its value, if one exists.

Mini-exercises

1. Using your `myFavoriteSong` variable from earlier, use optional binding to check if it contains a value. If it does, print out the value. If it doesn't, print "I don't have a favorite song."
2. Change `myFavoriteSong` to the opposite of what it is now. If it's `nil`, set it to a string; if it's a string, set it to `nil`. Observe how your printed result changes.

Introducing guard

Sometimes you want to check a condition and only continue executing a function if the condition is true, such as when you use optionals. Imagine a function that fetches some data from the network. That fetch might fail if the network is down. The usual way to encapsulate this behavior is using an optional, which has a value if the fetch succeeds, and `nil` otherwise.

Swift has a useful and powerful feature to help in situations like this: the **guard statement**.

Consider the following function:

```
func calculateNumberOfSides(shape: String) -> Int? {
    switch shape {
    case "Triangle":
        return 3
    case "Square":
        return 4
    case "Rectangle":
        return 4
    case "Pentagon":
        return 5
    case "Hexagon":
        return 6
    default:
        return nil
    }
}
```

This function takes a shape name and returns the number of sides that shape has. If the shape isn't known, or you pass something that isn't a shape, then it returns `nil`.

You could use this function like so:

```
func maybePrintSides(shape: String) {
    let sides = calculateNumberOfSides(shape: shape)

    if let sides = sides {
        print("A \(shape) has \(sides) sides.")
    } else {
        print("I don't know the number of sides for \(shape).")
    }
}
```

There's nothing wrong with this, and it would work.

However the same logic could be written with a guard statement like so:

```
func maybePrintSides(shape: String) {
    guard let sides = calculateNumberOfSides(shape: shape) else {
        print("I don't know the number of sides for \(shape).")
        return
    }

    print("A \(shape) has \(sides) sides.")
}
```

The guard statement comprises `guard` followed by a condition that can include both Boolean expressions and optional bindings, followed by `else`, followed by a block of code. The block of code covered by the `else` will execute if the condition is *false*.

The block of code that executes in the case of the condition being false *must* return. If you accidentally forget, the compiler will stop you - this is the true beauty of the guard statement.

You may hear programmers talking about the “happy path” through a function; this is the path you’d expect to happen most of the time. Any other path followed would be due to an error, or another reason why the function should return earlier than expected.

Guard statements ensure the happy path remains on the left hand side of the code; this is usually regarded as a good thing as it makes code more readable and understandable.

Also, because the guard statement must return in the false case, the Swift compiler knows that if the condition was true, anything checked in the guard statement’s condition *must* be true for the remainder of the function. This means the compiler can make certain optimizations. You don’t need to understand how these optimizations work, or even what they are, since Swift is designed to be user-friendly and fast.

Nil coalescing

There’s one final and rather handy way to unwrap an optional. You use it when you want to get a value out of the optional *no matter what* — and in the case of `nil`, you’ll use a default value. This is called **nil coalescing**.

Here’s how it works:

```
var optionalInt: Int? = 10
var mustHaveResult = optionalInt ?? 0
```

The nil coalescing happens on the second line, with the double question mark (??), known as the **nil coalescing operator**. This line means `mustHaveResult` will equal either the value inside `optionalInt`, or `0` if `optionalInt` contains `nil`. In this example, `mustHaveResult` contains the concrete `Int` value of `10`.

The previous code is equivalent to the following:

```
var optionalInt: Int? = 10
var mustHaveResult: Int
if let unwrapped = optionalInt {
    mustHaveResult = unwrapped
} else {
    mustHaveResult = 0
}
```


Set the `optionalInt` to `nil`, like so:

```
optionalInt = nil
mustHaveResult = optionalInt ?? 0
```

Now `mustHaveResult` equals 0.

Key points

- `nil` represents the absence of a value.
- Non-optional variables and constants must always have a non-`nil` value.
- **Optional** variables and constants are like boxes that can contain a value *or* be empty (`nil`).
- To work with the value inside an optional, you must first unwrap it from the optional.
- The safest ways to unwrap an optional's value is by using **optional binding** or **nil coalescing**. Use **forced unwrapping** only when appropriate, as it could produce a runtime error.

Where to go from here?

Optionals are a core Swift feature that helps make the language safe and easy to use. You'll find yourself using them throughout your code, making your code safe by ensuring that the absence of values is handled explicitly. This is something that you'll hopefully come to admire over the course of your Swift experience!

In particular, look forward to using them in Section II, where you'll learn about collections.

Challenges

You've learned the theory behind optionals and seen them in practice. Now it's your turn to play!

Challenge 1: You be the compiler

Which of the following are valid statements?

```
var name: String? = "Ray"
var age: Int = nil
let distance: Float = 26.7
var middleName: String? = nil
```

Challenge 2: Divide and conquer

First, create a function that returns the number of times an integer can be divided by another integer without a remainder. The function should return `nil` if the division doesn't produce a whole number. Name the function `divideIfWhole`.

Then, write code that tries to unwrap the optional result of the function. There should be two cases: upon success, print "Yep, it divides \(\answer) times", and upon failure, print "Not divisible :[".

Finally, test your function:

1. Divide 10 by 2. This should print "Yep, it divides 5 times."
2. Divide 10 by 3. This should print "Not divisible :[".

Hint 1: Use the following as the start of the function signature:

```
func divideIfWhole(_ value: Int, by divisor: Int)
```

You'll need to add the return type, which will be an optional!

Hint 2: You can use the modulo operator (%) to determine if a value is divisible by another; recall that this operation returns the remainder from the division of two numbers. For example, `10 % 2 = 0` means that 10 is divisible by 2 with no remainder, whereas `10 % 3 = 1` means that 10 is divisible by 3 with a remainder of 1.

Challenge 3: Refactor and reduce

The code you wrote in the last challenge used `if` statements. In this challenge, refactor that code to use `nil` coalescing instead. This time, make it print "It divides X times" in all cases, but if the division doesn't result in a whole number, then X should be 0.

Challenge 4: Nested optionals

Consider the following nested optional. It corresponds to a number inside a box inside a box inside a box.

```
let number: Int??? = 10
```

If you print number you get the following:

```
print(number)
// Optional(Optional(Optional(10)))

print(number!)
// Optional(Optional(10))
```

Do the following:

1. Fully force unwrap and print number.
2. Optionally bind and print number with `if let`.
3. Write a function `printNumber(_ number: Int???)` that uses `guard` to print the number only if it is bound.

Chapter 14: Arrays, Dictionaries, Sets

By Eli Ganim

As discussed in the introduction to this section, collections are flexible "containers" that let you store any number of values together. Before discussing these collections, you need to understand the concept of *mutable* vs *immutable* collections.

Mutable versus immutable collections

Just like the previous types you've read about, such as `Int` or `String`, when you create a collection you must declare it as either a constant or a variable.

If the collection doesn't need to change after you've created it, you should make it immutable by declaring it as a constant with `let`. Alternatively, if you need to add, remove or update values in the collection, then you should create a mutable collection by declaring it as a variable with `var`.

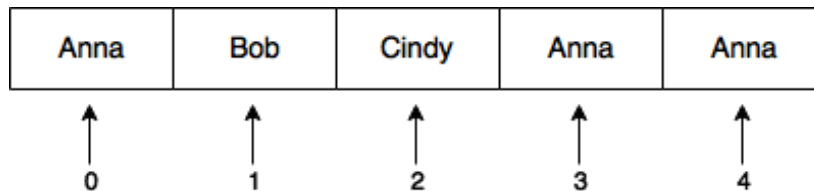
Arrays

Arrays are the most common collection type you'll run into in Swift. Arrays are typed, just like regular variables and constants, and store multiple values like a simple list.

Before you create your first array, take some time to consider in detail what an array is and why you might want to use one.

What is an array?

An array is an ordered collection of values of the same type. The elements in the array are **zero-indexed**, which means the index of the first element is 0, the index of the second element is 1, and so on. Knowing this, you can work out that the last element's index is the number of values in the array less 1.



There are five elements in this array, at indices 0–4.

All values are of type `String`, so you can't add non-string types to an array that holds strings. Notice that the same value can appear multiple times.

When are arrays useful?

Arrays are useful when you want to store your items in a particular order. You may want the elements sorted, or you may need to fetch elements by index without iterating through the entire array.

For example, if you were storing high score data, then order would matter. You would want the highest score to come first in the list (i.e. at index 0) with the next-highest score after that, and so on.

Creating arrays

The easiest way to create an array is by using an **array literal**. This is a concise way to provide array values. An array literal is a list of values separated by commas and surrounded by square brackets:

```
let evenNumbers = [2, 4, 6, 8]
```

Since the array literal only contains integers, Swift infers the type of `evenNumbers` to be an array of `Int` values. This type is written as `[Int]`. The type inside the square brackets defines the type of values the array can store, which the compiler will enforce when you add elements to the array.

If you try to add a string, for example, the compiler will return an error and your code won't compile.

An empty array can be created using the empty array literal `[]`. Because the compiler isn't able to infer a type from this, you need to use a type annotation to make the type explicit:

```
var subscribers: [String] = []
```

It's also possible to create an array with all of its values set to a default value:

```
let allZeros = Array(repeating: 0, count: 5) // [0, 0, 0, 0, 0]
```

It's good practice to declare arrays that aren't going to change as constants. For example, consider this array:

```
let vowels = ["A", "E", "I", "O", "U"]
```

`vowels` is an array of strings and its values can't be changed. But that's fine, since the list of vowels doesn't tend to change very often!

Accessing elements

Being able to create arrays is useless unless you know how to fetch values from an array. In this section, you'll learn several different ways to access elements in an array.

Using properties and methods

Imagine you're creating a game of cards, and you want to store the players' names in an array. The list will need to change as players join or leave the game, so you need to declare a mutable array:

```
var players = ["Alice", "Bob", "Cindy", "Dan"]
```

In this example, `players` is a mutable array because you assigned it to a variable.

Before the game starts, you need to make sure there are enough players. You can use the `isEmpty` property to check if there's at least one player:

```
print(players.isEmpty)  
// > false
```

Note: You'll learn all about properties in Chapter 11, "Properties". For now, just think of them as variables that are built in to values. To access a property, place a dot after the name of the constant or variable that holds the value and follow it by the name of the property you want to access.

The array isn't empty, but you need at least two players to start a game. You can get the number of players using the `count` property:

```
if players.count < 2 {  
    print("We need at least two players!")  
} else {  
    print("Let's start!")  
}  
// > Let's start!
```

It's time to start the game! You decide that the order of play is by the order of names in the array. How would you get the first player's name?

Arrays provide the `first` property to fetch the first object of an array:

```
var currentPlayer = players.first
```

Printing the value of `currentPlayer` reveals something interesting:

```
print(currentPlayer as Any)  
// > Optional("Alice")
```

The property `first` actually returns an *optional*, because if the array were empty, `first` would return `nil`. The `print()` method realizes `currentPlayer` is optional and generates a warning. To suppress the warning, simply add `as Any` to the type to be printed.

Similarly, arrays have a `last` property that returns the last value in an array, or `nil` if the array is empty:

```
print(players.last as Any)  
// > Optional("Dan")
```

Another way to get values from an array is by calling `min()`. This *method* returns the element with the lowest *value* in the array — not the lowest index! If the array contained strings, then it would return the string that's the lowest in alphabetical order, which in this case is "Alice":

```
currentPlayer = players.min()  
print(currentPlayer as Any)  
// > Optional("Alice")
```

Note: You'll learn all about methods in Chapter 12, "Methods". For now, just think of them as functions that are built in to values. To call a method, place a dot after the name of the constant or variable that holds the value and follow it by the name of the method you want to call. Just like with functions, don't forget to include the parameter list, even if it's empty, when calling a method.

Obviously, `first` and `min()` will not always return the same value. For example:

```
print([2, 3, 1].first as Any)
// > Optional(2)
print([2, 3, 1].min() as Any)
// > Optional(1)
```

As you might have guessed, arrays also have a `max()` method.

Note: The `first` and `last` properties and the `min()` and `max()` methods aren't unique to arrays. Every collection type has these properties and methods, in addition to a plethora of others. You'll learn more about this behavior when you read about protocols in Chapter 16, "Protocols".

Now that you know how to get the first player, you'll announce who that player is:

```
if let currentPlayer = currentPlayer {
    print("\(currentPlayer) will start")
}
// > Alice will start
```

You use `if let` to unwrap the optional you got back from `min()`; otherwise, the statement would print `Optional("Alice") will start`, which is not what you want.

These properties and methods are helpful if you want to get the first, last, minimum or maximum elements. But what if the element you want can't be obtained with one of these properties or methods?

Using subscripting

The most convenient way to access elements in an array is by using the subscript syntax. This syntax lets you access any value directly by using its index inside square brackets:

```
var firstPlayer = players[0]
print("First player is \(firstPlayer)")
// > First player is "Alice"
```


Because arrays are zero-indexed, you use index 0 to fetch the first object. You can use a greater index to get the next elements in the array, but if you try to access an index that's beyond the size of the array you'll get a runtime error.

```
var player = players[4]
// > fatal error: Index out of range
```

You receive this error because `players` contains only four strings. Index 4 represents the fifth element, but there is no fifth element in this array.

When you use subscripts, you don't have to worry about optionals, since trying to access a non-existing index doesn't return `nil`; it simply causes a runtime error.

Using countable ranges to make an `ArraySlice`

You can use the subscript syntax with countable ranges to fetch more than a single value from an array. For example, if you'd like to get the next two players, you could do this:

```
let upcomingPlayersSlice = players[1...2]
print(upcomingPlayersSlice[1], upcomingPlayersSlice[2])
// > "Bob Cindy\n"
```

The constant `upcomingPlayersSlice` is actually an `ArraySlice` of the original array. The reason for this type difference is to make clear that `upcomingPlayersSlice` shares storage with `players`.

The range you used is `1...2`, which represents the second and third items in the array. You can use an index here as long as the start value is smaller than or equal to the end value and both within the bounds of the array.

It is also easy to make a brand-new, zero-indexed `Array` from an `ArraySlice` like so:

```
let upcomingPlayersArray = Array(players[1...2])
print(upcomingPlayersArray[0], upcomingPlayersArray[1])
// > "Bob Cindy\n"
```

Checking for an element

You can check if there's at least one occurrence of a specific element in an array by using `contains(_:)`, which returns `true` if it finds the element in the array, and `false` otherwise.

You can use this strategy to write a function that checks if a given player is in the game:

```
func isEliminated(player: String) -> Bool {  
    return !players.contains(player)  
}
```

Now you can use this function any time you need to check if a player has been eliminated:

```
print(isEliminated(player: "Bob"))  
// > false
```

You could even test for the existence of an element in a specific range using an `ArraySlice`:

```
players[1...3].contains("Bob") // true
```

Now that you can get data *out* of your arrays, it's time to look at mutable arrays and how to change their values.

Modifying arrays

You can make all kinds of changes to mutable arrays, such as adding and removing elements, updating existing values, and moving elements around into a different order. In this section, you'll see how to work with the array to match up what's going on with your game.

Appending elements

If new players want to join the game, they need to sign up and add their names to the array. Eli is the first player to join the existing four players. You can add Eli to the end of the array using the `append(_:)` method:

```
players.append("Eli")
```

If you try to append anything other than a string, the compiler will show an error. Remember, arrays can only store values of the same type. Also, `append(_:)` only works with mutable arrays.

The next player to join the game is Gina. You can append her to the game another way, by using the `+=` operator:

```
players += ["Gina"]
```

The right-hand side of this expression is an array with a single element: the string "Gina". By using `+=`, you're appending the elements of that array to `players`. Now the array looks like this:

```
print(players)
// > ["Alice", "Bob", "Cindy", "Dan", "Eli", "Gina"]
```

Here, you added a single element to the array, but you can see how easy it would be to append *multiple* items using the `+=` operator by adding more names after Gina's.

Inserting elements

An unwritten rule of this card game is that the players' names have to be in alphabetical order. This list is missing a player that starts with the letter F. Luckily, Frank has just arrived. You want to add him to the list between Eli and Gina. To do that, you can use the `insert(_:at:)` method:

```
players.insert("Frank", at: 5)
```

The `at` argument defines where you want to add the element. Remember that the array is zero-indexed, so index 5 is Gina's index, causing her to move up as Frank takes her place.

Removing elements

During the game, the other players caught Cindy and Gina cheating. They should be removed from the game! You know that Gina is last in the `players` list, so you can remove her easily with the `removeLast()` method:

```
var removedPlayer = players.removeLast()
print("\(removedPlayer) was removed")
// > Gina was removed
```

This method does two things: It removes the last element and then returns it, in case you need to print it or store it somewhere else — like in an array of cheaters!

To remove Cindy from the game, you need to know the exact index where her name is stored. Looking at the list of `players`, you see that she's third in the list, so her index is 2.

```
removedPlayer = players.remove(at: 2)
print("\(removedPlayer) was removed")
// > Cindy was removed
```

But how would you get the index of an element if you didn't already know it? There's a method for that! `index(of:)` returns the *first index* of the element, because the array might contain multiple copies of the same value. If the method doesn't find the element, it returns `nil`.

Mini-exercise

Use `index(of:)` to determine the position of the element "Dan" in `players`.

Updating elements

Frank has decided everyone should call him Franklin from now on. You could remove the value "Frank" from the array and then add "Franklin", but that's too much work for a simple task. Instead, you should use the subscript syntax to update the name.

```
print(players)
// > ["Alice", "Bob", "Dan", "Eli", "Frank"]
players[4] = "Franklin"
print(players)
// > ["Alice", "Bob", "Dan", "Eli", "Franklin"]
```

Be careful to not use an index beyond the bounds of the array, or your code will crash.

As the game continues, some players are eliminated, and new ones come to replace them. You can also use subscripting with ranges to update multiple values in a single line of code:

```
players[0...1] = ["Donna", "Craig", "Brian", "Anna"]
print(players)
// > ["Donna", "Craig", "Brian", "Anna", "Dan", "Eli", "Franklin"]
```

This code replaces the first two players, Alice and Bob, with the four players in the new `players` array. As you can see, the size of the range doesn't have to be equal to the size of the array that holds the values you're adding.

Moving elements

Take a look at this mess! The `players` array contains names that start with A to F, but they aren't in the correct order, and that violates the rules of the game.

You can try to fix this situation by manually moving values one by one to their correct positions, like so:

```
let playerAnna = players.remove(at: 3)
players.insert(playerAnna, at: 0)
print(players)
// > ["Anna", "Donna", "Craig", "Brian", "Dan", "Eli", "Franklin"]
```

...or by swapping elements, by using `swapAt(_:_:)`:

```
players.swapAt(1, 3)
print(players)
// > ["Anna", "Brian", "Craig", "Donna", "Dan", "Eli", "Franklin"]
```

This works if you want to move a few elements, but if you want to sort the entire array, you should use `sort()`:

```
players.sort()
print(players)
// > ["Anna", "Brian", "Craig", "Dan", "Donna", "Eli", "Franklin"]
```

If you'd like to leave the original array untouched and return a sorted *copy* instead, use `sorted()` instead of `sort()`.

Iterating through an array

It's getting late, so the players decide to stop for the night and continue tomorrow. In the meantime, you'll keep their scores in a separate array. You'll investigate a better approach for this when you learn about dictionaries, but for now you can continue to use arrays:

```
let scores = [2, 2, 8, 6, 1, 2, 1]
```

Before the players leave, you want to print the names of those still in the game. You can do this using the `for-in` loop you read about in Chapter 4, "Advanced Control Flow":

```
for player in players {
    print(player)
}
// > Anna
// > Brian
// > Craig
// > Dan
// > Donna
// > Eli
// > Franklin
```

This code goes over all the elements of `players`, from index 0 up to `players.count - 1` and prints their values. In the first iteration, `player` is equal to the first element of the array; in the second iteration, it's equal to the second element of the array; and so on, until the loop has printed all the elements in the array.

If you also need the index of each element, you can iterate over the return value of the array's `enumerated()` method, which returns tuples with the index and value of each element in the array:

```
for (index, player) in players.enumerated() {  
    print("\(index + 1). \(player)")  
}  
// > 1. Anna  
// > 2. Brian  
// > 3. Craig  
// > 4. Dan  
// > 5. Donna  
// > 6. Eli  
// > 7. Franklin
```

Now you can use the technique you've just learned to write a function that takes an array of integers as its input and returns the sum of its elements:

```
func sumOfElements(in array: [Int]) -> Int {  
    var sum = 0  
    for number in array {  
        sum += number  
    }  
    return sum  
}
```

You could use this function to calculate the sum of the players' scores:

```
print(sumOfElements(in: scores))  
// > 22
```

Mini-exercise

Write a `for-in` loop that prints the players' names and scores.

Running time for array operations

Arrays are stored as a continuous block in memory. That means if you have ten elements in an array, the ten values are all stored one next to the other. With that in mind, here's the performance cost of various array operations:

Accessing elements: The cost of fetching an element is $O(1)$. Since all the values are sequential, it's easy to use *random access* and fetch a value at a particular index; all the compiler needs to know is where the array starts and what index you want to fetch.

Inserting elements: The complexity of adding an element depends on the position in which you add the new element:

- If you add to the beginning of the array, Swift requires $O(N)$ because it has to shift all of the elements over by one to make room.
- Likewise, if you add to the middle of the array, all values from that index on need to be shifted over. Doing so will require $N/2$ operations, therefore the running time is $O(N)$.
- If you add to the end of the array using `append` and there's room, it will take $O(1)$. If there isn't room, Swift will need to make space somewhere else and copy the entire array over before adding the new element, which will take $O(N)$. The average case is $O(1)$ though, because arrays are not full most of the time.

Deleting elements: Deleting an element leaves a gap where the removed element was. All elements in the array have to be sequential, so this gap needs to be closed by shifting elements forward.

The complexity is similar to inserting elements: If you're removing an element from the end, it's an $O(1)$ operation. Otherwise the complexity is $O(N)$.

Searching for an element: If the element you're searching for is the first element in the array, then the search will end after a single operation. If the element doesn't exist, you need to perform N operations until you realize that the element is not found. On average, searching for an element will take $N/2$ operations, therefore searching has a complexity of $O(N)$.

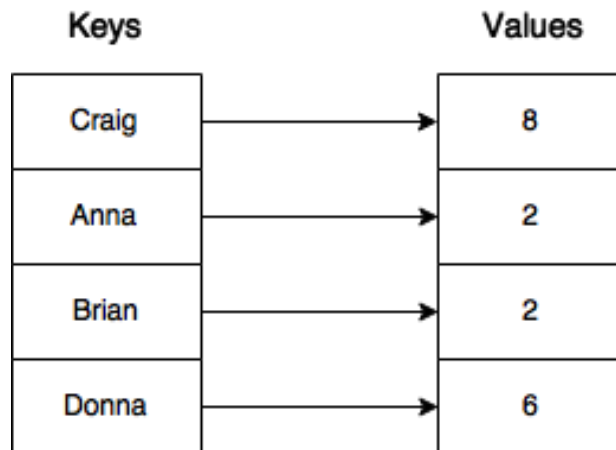
As you learn about dictionaries and sets, you'll see how their performance characteristics differ from arrays. That could give you a hint on which collection type to use for your particular case.

Key points

- **Arrays** are ordered collections of values of the same type.
- Use **subscripting** or one of the many properties and methods to access and update elements.
- Be wary of accessing an index that's out of bounds.

Dictionaries

A dictionary is an unordered collection of pairs, where each pair is comprised of a **key** and a **value**.



As shown in the diagram above, keys are unique. The same key can't appear twice in a dictionary, but different keys may point to the same value. All keys have to be of the same type, and all values have to be of the same type.

Dictionaries are useful when you want to look up values by means of an identifier. For example, the table of contents of this book maps chapter names to their page numbers, making it easy to skip to the chapter you want to read.

How is this different from an array? With an array, you can only fetch a value by its index, which has to be an integer, and all indexes have to be sequential. In a dictionary, the keys can be of any type and in no particular order.

Creating dictionaries

The easiest way to create a dictionary is by using a **dictionary literal**. This is a list of key-value pairs separated by commas, enclosed in square brackets.

For your card game from earlier, instead of using the two arrays to map players to their scores, you can use a dictionary literal:

```
var namesAndScores = ["Anna": 2, "Brian": 2, "Craig": 8, "Donna": 6]
print(namesAndScores)
// > ["Craig": 8, "Anna": 2, "Donna": 6, "Brian": 2]
```


In this example, the type of the dictionary is inferred to be `[String: Int]`. This means `namesAndScores` is a dictionary with strings as keys and integers as values.

When you print the dictionary, you see there's no particular order to the pairs. Remember that, unlike arrays, dictionaries are unordered!

The empty dictionary literal looks like this: `[:]`. You can use that to empty an existing dictionary like so:

```
namesAndScores = [:]
```

...or create a new dictionary, like so:

```
var pairs: [String: Int] = [:]
```

The type annotation is required here, as the compiler can't infer the type of the dictionary from the empty dictionary literal.

After you create a dictionary, you can define its capacity:

```
pairs.reserveCapacity(20)
```

Using `reserveCapacity(_:)` is an easy way to improve performance when you have an idea of how much data the dictionary needs to store.

Accessing values

As with arrays, there are several ways to access dictionary values.

Using subscripting

Dictionaries support subscripting to access values. Unlike arrays, you don't access a value by its index but rather by its key. For example, if you want to get Anna's score, you would type:

```
namesAndScores = ["Anna": 2, "Brian": 2, "Craig": 8, "Donna": 6]
// Restore the values

print(namesAndScores["Anna"])
// > Optional(2)
```

Notice that the return type is an optional. The dictionary will check if there's a pair with the key Anna, and if there is, return its value. If the dictionary doesn't find the key, it will return `nil`.

```
namesAndScores["Greg"] // nil
```

With arrays, out-of-bounds subscript access causes a runtime error, but dictionaries are different since their results are wrapped in an optional.

Subscript access using optionals is really powerful. You can find out if a specific player is in the game without having to iterate over all the keys, as you must do when you use an array.

Using properties and methods

Dictionaries, like arrays, conform to Swift's `Collection` protocol. Because of that, they share many of the same properties. For example, both arrays and dictionaries have `isEmpty` and `count` properties:

```
namesAndScores.isEmpty // false
namesAndScores.count    // 4
```

Note: If you just want to know whether a dictionary has elements or not, it is always better to use the `isEmpty` property. A dictionary needs to loop through all of the values to compute the count. `isEmpty`, by contrast, always runs in constant time no matter how many values there are.

Modifying dictionaries

It's easy enough to create dictionaries and access their contents — but what about modifying them?

Adding pairs

Bob wants to join the game.



Take a look at his details before you let him join:

```
var bobData = [
    "name": "Bob",
    "profession": "Card Player",
    "country": "USA"
]
```

This dictionary is of type `[String: String]`, and it's mutable because it's assigned to a variable. Imagine you received more information about Bob and you wanted to add it to the dictionary. This is how you'd do it:

```
bobData.updateValue("CA", forKey: "state")
```

There's even a shorter way to add pairs, using subscripting:

```
bobData["city"] = "San Francisco"
```

Bob's a professional card player. So far, he sounds like a good addition to your roster.

Mini-exercise

Write a function that prints a given player's city and state.

Updating values

It appears that in the past, Bob was caught cheating when playing cards. He's not just a professional — he's a card shark! He asks you to change his name and profession so no one will recognize him.

Because Bob seems eager to change his ways, you agree. First, you change his name from Bob to Bobby:

```
bobData.updateValue("Bobby", forKey: "name") // Bob
```

You saw this method above when you read about adding pairs. Why does it return the string `Bob`? `updateValue(_:forKey:)` replaces the value of the given key with the new value and returns the old value. If the key doesn't exist, this method will add a new pair and return `nil`.

As with adding, you can do this with less code by using subscripting:

```
bobData["profession"] = "Mailman"
```

Like `updateValue(_:forKey:)`, this code updates the value for this key or, if the key doesn't exist, creates a new pair.

Removing pairs

Bob — er, sorry — *Bobby*, still doesn't feel safe, and he wants you to remove all information about his whereabouts:

```
bobData.removeValue(forKey: "state")
```

This method will remove the key `state` and its associated value from the dictionary. As you might expect, there's a shorter way to do this using subscripting:

```
bobData["city"] = nil
```

Assigning `nil` as a key's associated value removes the pair from the dictionary.

Note: If you are using a dictionary that has values that are optional types, `dictionary[key] = nil` still removes the key completely. If you want keep the key and set the value to `nil` you must use the `updateValue` method.

Iterating through dictionaries

The `for-in` loop also works when you want to iterate over a dictionary. But since the items in a dictionary are pairs, you need to use a tuple:

```
for (player, score) in namesAndScores {  
    print("\(player) - \(score)")  
}  
// > Craig - 8  
// > Anna - 2  
// > Donna - 6  
// > Brian - 2
```

It's also possible to iterate over just the keys:

```
for player in namesAndScores.keys {  
    print("\(player), ", terminator: "") // no newline  
}  
print("") // print one final newline  
// > Craig, Anna, Donna, Brian,
```

You can iterate over just the values in the same manner with the `values` property of the dictionary.

Running time for dictionary operations

In order to be able to examine how dictionaries work, you need to understand what **hashing** is and how it works. Hashing is the process of transforming a value — `String`, `Int`, `Double`, `Bool`, etc — to a numeric value, known as the *hash value*. This value can then be used to quickly lookup the values in a *hash table*.

Swift dictionaries have a type requirement for keys. Keys must be **Hashable** or you will get a compiler error.

Fortunately, in Swift, all basic types are already `Hashable` and have a `hashValue` integer property. Here's an example:

```
print("some string".hashValue)
// > -497521651836391849
print(1.hashValue)
// > 1
print(false.hashValue)
// > 0
```

The hash value has to be deterministic — meaning that a given value must *always* return the same hash value. No matter how many times you calculate the hash value for some `string`, it will always give the same value. (You should never save a hash value, however, as there is no guarantee it will be the same from run-to-run of your program.)

Here's the performance of various dictionary operations. This great performance hinges on having a good hashing function that avoids value collisions. If you have a poor hashing function, all of the operations below degenerate to linear time, or $O(n)$ performance. Fortunately, the built-in types have great, general purpose `hashValue` implementations.

Accessing elements: Getting the value for a given key is a constant time operation, or $O(1)$.

Inserting elements: To insert an element, the dictionary needs to calculate the hash value of the key and then store data based on that hash. These are all $O(1)$ operations.

Deleting elements: Again, the dictionary needs to calculate the hash value to know exactly where to find the element, and then remove it. This is also an $O(1)$ operation.

Searching for an element: As mentioned above, accessing an element has constant running time, so the complexity for searching is also $O(1)$.

While all of these running times compare favorably to arrays, remember that you lose order information when using dictionaries.

Key points

- A **dictionary** is an unordered collection of key-value pairs.
- The **keys** of a dictionary are all of the same type, and the **values** are all of the same type.
- Use **subscripting** to get values and to add, update or remove pairs.
- If a key is not in a dictionary, lookup returns `nil`.
- The key of a dictionary must be a type that conforms to the **Hashable** protocol.
- Basic Swift types such as `String`, `Int`, `Double` are Hashable out of the box.

Sets

A set is an unordered collection of unique values of the same type. This can be extremely useful when you want to ensure that an item doesn't appear more than once in your collection, and when the order of your items isn't important.

Creating sets

You can declare a set explicitly by writing `Set` followed by the type inside angle brackets:

```
let setOne: Set<Int>
```

Set literals

Sets don't have their own literals. You use **array literals** to create a set with initial values. Consider this example:

```
let someArray = [1, 2, 3, 1]
```

This is an array. So how would you use array literals to create a set? Like this:

```
var someSet: Set<Int> = [1, 2, 3, 1]
```

You have to explicitly declare the variable as a `Set`.

To see the most important features of a set in action, let's print the set you just created:

```
print(someSet)
// > [2, 3, 1]
```

First, you can see there's no specific ordering. Second, although you created the set with two instances of the value 1, that value only appears once. Remember, a set's values must be unique.

Accessing elements

You can use `contains(_:)` to check for the existence of a specific element:

```
print(someSet.contains(1))
// > true
print(someSet.contains(4))
// > false
```

You can also use the `first` and `last` properties, which return one of the elements in the set. However, because sets are unordered, you won't know exactly which item you'll get.

Adding and removing elements

You can use `insert(_:)` to add elements to a set. If the element already exists, the method does nothing.

```
someSet.insert(5)
```

You can remove the element from the set like this:

```
let removedElement = someSet.remove(1)
print(removedElement!)
// > 1
```

`remove(_:)` returns the removed element if it's in the set, or `nil` otherwise.

Running time for set operations

Sets have a very similar implementations to those of dictionaries, and they also require the elements to be hashable. The running time of all the operations is identical to those of dictionaries.

Key points

- **Sets** are unordered collections of unique values of the same type.
- Sets are most useful when you need to know whether something is included in the collection or not.

Where to go from here?

Now that you've learned about the three collection types in Swift, you should have a good idea of what they can do and when you should use them. You'll see them come up as you continue on in the book.

The next chapter of the book covers **collection iteration with closures**. These let you iterate over the collection types you've learned in a less explicit and more readable manner than loops.

Before you move on, check out the following challenges to test your collection knowledge.

Challenges

1. Which of the following are valid statements?

```
1. let array1 = [Int]()
2. let array2 = []
3. let array3: [String] = []
```

For the next five statements, array4 has been declared as:

```
let array4 = [1, 2, 3]
```

```
4. print(array4[0])
5. print(array4[5])
6. array4[1...2]
7. array4[0] = 4
8. array4.append(4)
```

For the final five statements, array5 has been declared as:

```
var array5 = [1, 2, 3]
```

```
9. array5[0] = array5[1]
10. array5[0...1] = [4, 5]
```



```
11. array5[0] = "Six"
12. array5 += 6
13. for item in array5 { print(item) }
```

2. Write a function that removes the first occurrence of a given integer from an array of integers. This is the signature of the function:

```
func removingOnce(_ item: Int, from array: [Int]) -> [Int]
```

3. Write a function that removes all occurrences of a given integer from an array of integers. This is the signature of the function:

```
func removing(_ item: Int, from array: [Int]) -> [Int]
```

4. Arrays have a `reversed()` method that returns an array holding the same elements as the original array, in reverse order. Write a function that does the same thing, without using `reversed()`. This is the signature of the function:

```
func reversed(_ array: [Int]) -> [Int]
```

5. The function below returns a random number between 0 and the given argument:

```
import Foundation
func randomFromZero(to number: Int) -> Int {
    return Int(arc4random_uniform(UInt32(number)))
}
```

Use it to write a function that shuffles the elements of an array in random order. This is the signature of the function:

```
func randomized(_ array: [Int]) -> [Int]
```

6. Write a function that calculates the minimum and maximum value in an array of integers. Calculate these values yourself; don't use the methods `min` and `max`. Return `nil` if the given array is empty.

This is the signature of the function:

```
func minMax(of numbers: [Int]) -> (min: Int, max: Int)?
```

7. Which of the following are valid statements?

```
1. let dict1: [Int, Int] = [:]
2. let dict2 = [:]
3. let dict3: [Int: Int] = [:]
```

For the next four statements, use the following dictionary:

```
let dict4 = ["One": 1, "Two": 2, "Three": 3]
```

```
4. dict4[1]
5. dict4["One"]
6. dict4["Zero"] = 0
7. dict4[0] = "Zero"
```

For the next three statements, use the following dictionary:

```
var dict5 = ["NY": "New York", "CA": "California"]
```

```
8. dict5["NY"]
9. dict5["WA"] = "Washington"
10. dict5["CA"] = nil
```

8. Given a dictionary with two-letter state codes as keys, and the full state names as values, write a function that prints all the states with names longer than eight characters. For example, for the dictionary ["NY": "New York", "CA": "California"], the output would be California.
9. Write a function that combines two dictionaries into one. If a certain key appears in both dictionaries, ignore the pair from the first dictionary. This is the function's signature:

```
func merging(_ dict1: [String: String], with dict2: [String: String]) -> [String: String]
```

10. Declare a function occurrencesOfCharacters that calculates which characters occur in a string, as well as how often each of these characters occur. Return the result as a dictionary. This is the function signature:

```
func occurrencesOfCharacters(in text: String) -> [Character: Int]
```

Hint: String is a collection of characters that you can iterate over with a for statement. Bonus: To make your code shorter, dictionaries have a special subscript operator that let you add a default value if it is not found in the dictionary. For example, dictionary["a", default: 0] creates a 0 entry for the character "a" if it is not found instead of just returning nil.

11. Write a function that returns true if all of the values of a dictionary are unique. Use a set to test uniqueness. This is the function signature:

```
func isInvertible(_ dictionary: [String: Int]) -> Bool
```

12. Given the dictionary:

```
var nameTitleLookup: [String: String?] = ["Mary": "Engineer", "Patrick":  
"Intern", "Ray": "Hacker"]
```

Set the value of the key "Patrick" to nil and completely remove the key and value for "Ray".

Chapter 15: Collection Iteration with Closures

By Matt Galloway

The previous section taught you about functions. But Swift has another object you can use to break up code into reusable chunks: a **closure**. They become particularly useful when dealing with collections.

A closure is simply a function with no name; you can assign it to a variable and pass it around like any other value. This chapter shows you how convenient and useful closures can be.

Closure basics

Closures are so named because they have the ability to “close over” the variables and constants within the closure’s own scope. This simply means that a closure can access, store and manipulate the value of any variable or constant from the surrounding context. Variables and constants used within the body of a closure are said to have been **captured** by the closure.

You may ask, “If closures are functions without names, then how do you use them?” To use a closure, you first have to assign it to a variable or constant.

Here’s a declaration of a variable that can hold a closure:

```
var multiplyClosure: (Int, Int) -> Int
```

`multiplyClosure` takes two `Int` values and returns an `Int`. Notice that this is exactly the same as a variable declaration for a function. Like I said, a closure is simply a function without a name. The type of a closure is a function type.

You assign a closure to a variable like so:

```
multiplyClosure = { (a: Int, b: Int) -> Int in
    return a * b
}
```

This looks similar to a function declaration, but there's a subtle difference. There's the same parameter list, `->` symbol and return type. But in the case of closures, these elements appear inside braces, and there is an `in` keyword after the return type.

With your closure variable defined, you can use it just as if it were a function, like so:

```
let result = multiplyClosure(4, 2)
```

As you'd expect, `result` equals 8. Again, though, there's a subtle difference.

Notice how the closure has no external names for the parameters. You can't set them like you can with functions.

Shorthand syntax

Compared to functions, closures are designed to be lightweight. There are many ways to shorten their syntax. First, if the closure consists of a single return statement, you can leave out the `return` keyword, like so:

```
multiplyClosure = { (a: Int, b: Int) -> Int in
    a * b
}
```

Next, you can use Swift's type inference to shorten the syntax even more by removing the type information:

```
multiplyClosure = { (a, b) in
    a * b
}
```

Remember, you already declared `multiplyClosure` as a closure taking two `Int`s and returning an `Int`, so you can let Swift infer these types for you.

And finally, you can even omit the parameter list if you want. Swift lets you refer to each parameter by number, starting at zero, like so:

```
multiplyClosure = {
    $0 * $1
}
```

The parameter list, return type and `in` keyword are all gone, and your new closure declaration is much shorter than the original. Numbered parameters like this should really only be used when the closure is short and sweet, like the one above.

If the parameter list is much longer it can be confusing to remember what each numbered parameter refers to. In these cases you should use the named syntax.

Consider the following code:

```
func operateOnNumbers(_ a: Int, _ b: Int,
                     operation: (Int, Int) -> Int) -> Int {
    let result = operation(a, b)
    print(result)
    return result
}
```

This declares a function named `operateOnNumbers`, which takes `Int` values as its first two parameters. The third parameter is named `operation` and is of a function type. `operateOnNumbers` itself returns an `Int`.

You can then use `operateOnNumbers` with a closure, like so:

```
let addClosure = { (a: Int, b: Int) in
    a + b
}
operateOnNumbers(4, 2, operation: addClosure)
```

Remember, closures are simply functions without names. So you shouldn't be surprised to learn that you can also pass in a function as the third parameter of `operateOnNumbers`, like so:

```
func addFunction(_ a: Int, _ b: Int) -> Int {
    return a + b
}
operateOnNumbers(4, 2, operation: addFunction)
```

`operateOnNumbers` is called the same way, whether the operation is a function or a closure.

The power of the closure syntax comes in handy again. You can define the closure inline with the `operateOnNumbers` function call, like this:

```
operateOnNumbers(4, 2, operation: { (a: Int, b: Int) -> Int in
    return a + b
})
```

There's no need to define the closure and assign it to a local variable or constant. You can simply declare the closure right where you pass it into the function as a parameter!

But recall that you can simplify the closure syntax to remove a lot of the boilerplate code. You can therefore reduce the above to the following:

```
operateOnNumbers(4, 2, operation: { $0 + $1 })
```

In fact, you can even go a step further. The `+` operator is just a function that takes two arguments and returns one result so you can write:

```
operateOnNumbers(4, 2, operation: +)
```

There's one more way you can simplify the syntax, but it can only be done when the closure is the final parameter passed to a function. In this case, you can move the closure outside of the function call:

```
operateOnNumbers(4, 2) {  
    $0 + $1  
}
```

This may look strange, but it's just the same as the previous code snippet, except you've removed the `operation` label and pulled the braces outside of the function call parameter list. This is called **trailing closure syntax**.

Closures with no return value

Until now, all the closures you've seen have taken one or more parameters and have returned values. But just like functions, closures aren't required to do these things. Here's how you declare a closure that takes no parameters and returns nothing:

```
let voidClosure: () -> Void = {  
    print("Swift Apprentice is awesome!")  
}  
voidClosure()
```

The closure's type is `() -> Void`. The empty parentheses denote there are no parameters. You must declare a return type, so Swift knows you're declaring a closure. This is where `Void` comes in handy, and it means exactly what its name suggests: the closure returns nothing.

Note: `Void` is actually just a typealias for `()`. This means you could have written `() -> Void` as `() -> ()`. A function's parameter list however must always be surrounded by parentheses, so `Void -> ()` or `Void -> Void` are invalid.

Capturing from the enclosing scope

Finally, let's return to the defining characteristic of a closure: it can access the variables and constants from within its own scope.

Note: Recall that scope defines the range in which an entity (variable, constant, etc) is accessible. You saw a new scope introduced with `if`-statements. Closures also introduce a new scope and inherit all entities visible to the scope in which it is defined.

For example, take the following closure:

```
var counter = 0
let incrementCounter = {
    counter += 1
}
```

`incrementCounter` is rather simple: It increments the `counter` variable. The `counter` variable is defined outside of the closure. The closure is able to access the variable because the closure is defined in the same scope as the variable. The closure is said to **capture** the `counter` variable. Any changes it makes to the variable are visible both inside and outside the closure.

Let's say you call the closure five times, like so:

```
incrementCounter()
incrementCounter()
incrementCounter()
incrementCounter()
incrementCounter()
```

After these five calls, `counter` will equal 5.

The fact that closures can be used to capture variables from the enclosing scope can be extremely useful. For example, you could write the following function:

```
func countingClosure() -> () -> Int {
    var counter = 0
    let incrementCounter: () -> Int = {
        counter += 1
        return counter
    }
    return incrementCounter
}
```

This function takes no parameters and returns a closure. The closure it returns takes no parameters and returns an `Int`.

The closure returned from this function will increment its internal counter each time it is called. Each time you call this function you get a different counter.

For example, this could be used like so:

```
let counter1 = countingClosure()
let counter2 = countingClosure()

counter1() // 1
counter2() // 1
counter1() // 2
counter1() // 3
counter2() // 2
```

The two counters created by the function are mutually exclusive and count independently. Neat!

Custom sorting with closures

Closures come in handy when you start looking deeper at collections. In Chapter 7, you used array's `sort` method to sort an array. By specifying a closure, you can customize how things are sorted. You call `sorted()` to get a sorted version of the array as so:

```
let names = ["ZZZZZZ", "BB", "A", "CCCC", "EEEE"]
names.sorted()
// ["A", "BB", "CCCC", "EEEE", "ZZZZZZ"]
```

By specifying a custom closure, you can change the details of how the array is sorted. Specify a trailing closure like so:

```
names.sorted {
    $0.count > $1.count
}
// ["ZZZZZZ", "EEEE", "CCCC", "BB", "A"]
```

Now the array is sorted by the length of the string with longer strings coming first.

Iterating over collections with closures

In Swift, collections implement some very handy features often associated with **functional programming**. These features come in the shape of functions that you can apply to a collection to perform an operation on it.

Operations include things like transforming each element or filtering out certain elements.

All of these functions make use of closures, as you will see now.

The first of these functions lets you loop over the elements in a collection and perform an operation like so:

```
let values = [1, 2, 3, 4, 5, 6]
values.forEach {
    print("\( $0 ): \( $0 * $0 )")
}
```

This loops through each item in the collection printing the value and its square.

Another function allows you to filter out certain elements, like so:

```
var prices = [1.5, 10, 4.99, 2.30, 8.19]

let largePrices = prices.filter {
    return $0 > 5
}
```

Here, you create an array of Double to represent the prices of items in a shop. To filter out the prices which are greater than \$5, you use the filter function. This function looks like so:

```
func filter(_ isIncluded: (Element) -> Bool) -> [Element]
```

This means that filter takes a single parameter, which is a closure (or function) that takes an Element and returns a Bool. The filter function then returns an array of Elements. In this context, Element refers to the type of items in the array. In the example above, Doubles.

The closure's job is to return true or false depending on whether or not the value should be kept or not. The array returned from filter will contain all elements for which the closure returned true.

In your example, largePrices will contain:

```
(10, 8.19)
```

Note: The array that is returned from filter (and all of these functions) is a new array. The original is not modified at all.

However, there is more!

Imagine you're having a sale and want to discount all items to 90% of their original price. There's a handy function named `map` which can achieve this:

```
let salePrices = prices.map {  
    return $0 * 0.9  
}
```

The `map` function will take a closure, execute it on each item in the array and return a new array containing each result with the order maintained. In this case, `salePrices` will contain:

```
[1.35, 9, 4.491, 2.07, 7.371]
```

The `map` function can also be used to change the type. You can do that like so:

```
let userInput = ["0", "11", "haha", "42"]  
  
let numbers1 = userInput.map {  
    Int($0)  
}
```

This takes some strings that the user input and turns them into an array of `Int?`. They need to be optional because the conversion from `String` to `Int` might fail.

If you want to filter out the invalid (missing) values, you can use `flatMap()` like so:

```
let numbers2 = userInput.flatMap {Int($0)}
```

This is almost the same as `map` except it creates an array of `Int` and tosses out the missing values.

Another handy function is called `reduce`. This function takes a starting value and a closure. The closure takes two values: the current value and an element from the array. The closure returns the next value that should be passed into the closure as the current value parameter.

This could be used with the `prices` array to calculate the total, like so:

```
let sum = prices.reduce(0) {  
    return $0 + $1  
}
```

The initial value is 0. Then the closure calculates the sum of the current value plus the current iteration's value. Thus you calculate the total of all the values in the array. In this case, sum will be:

26.98

Now that you've seen `filter`, `map` and `reduce`, hopefully it's becoming clear how powerful these functions can be, especially thanks to the syntax of closures. In just a few lines of code, you have calculated quite complex values from the collection.

These functions can also be used with dictionaries. Imagine you represent the stock in your shop by a dictionary mapping the price to number of items at that price. You could use that to calculate the total value of your stock like so:

```
let stock = [1.5: 5, 10: 2, 4.99: 20, 2.30: 5, 8.19: 30]
let stockSum = stock.reduce(0) {
    return $0 + $1.key * Double($1.value)
}
```

In this case, the second parameter to the `reduce` function is a named tuple containing the key and value from the dictionary elements. A type conversion of the value is required to perform the calculation.

Here, the result is:

384.5

There's also another form of `reduce` that is useful when the result you're reducing your collection into is an array or dictionary. It's called `reduce(into: _:)`. You use it like so:

```
let farmAnimals = [{"🐎": 5, "🐕": 10, "🐑": 50, "🐼": 1}]
let allAnimals = farmAnimals.reduce(into: []) {
    (result, this: (key: String, value: Int)) in
    for _ in 0 ..< this.value {
        result.append(this.key)
    }
}
```

It works in exactly the same way as the other version, except that you don't return something from the closure. Instead, each iteration you are given a mutable value to do with what you want. In this way, there is only ever one array in this example that is created and appended to. This makes it more performant and therefore more desirable to use.

There are also a few different functions which can be helpful when you need to chop up an array. The first function is called `dropFirst`, which works like so:

```
let removeFirst = prices.dropFirst()
let removeFirstTwo = prices.dropFirst(2)
```

The `dropFirst` function takes a single parameter which defaults to 1 and returns an array with the required number of elements removed from the front. In this case the results will be as follows:

```
removeFirst = [10, 4.99, 2.30, 8.19]
removeFirstTwo = [4.99, 2.30, 8.19]
```

Just like `dropFirst`, there also exists `dropLast` which removes elements from the end of the array. It works like this:

```
let removeLast = prices.dropLast()
let removeLastTwo = prices.dropLast(2)
```

The results of these are as you would expect, as follows:

```
removeLast = [1.5, 10, 4.99, 2.30]
removeLastTwo = [1.5, 10, 4.99]
```

And finally, you can select just the first or last elements of an array as shown below:

```
let firstTwo = prices.prefix(2)
let lastTwo = prices.suffix(2)
```

Here, `prefix` returns the required number of elements from the front of the array, and `suffix` returns the required number of elements from the back of the array. The results of this function are:

```
firstTwo = [1.5, 10]
lastTwo = [2.30, 8.19]
```

That wraps up collection iteration with closures!

Mini-exercises

1. Create a constant array called `names` which contains some names as strings. Any names will do — make sure there's more than three. Now use `reduce` to create a string which is the concatenation of each name in the array.
2. Using the same `names` array, first filter the array to contain only names which have more than four characters in them, and then create the same concatenation of names as in the above exercise. (Hint: you can chain these operations together.)

3. Create a constant dictionary called `namesAndAges` which contains some names as strings mapped to ages as integers. Now use `filter` to create a dictionary containing only people under the age of 18.
4. Using the same `namesAndAges` dictionary, filter out the adults (those 18 or older) and then use `map` to convert to an array containing just the names (i.e. drop the ages).

Key points

- **Closures** are functions without names. They can be assigned to variables and passed as parameters to functions.
- Closures have **shorthand syntax** that makes them a lot easier to use than other functions.
- A closure can **capture** the variables and constants from its surrounding context.
- A closure can be used to direct how a collection is sorted.
- There exist a handy set of functions on collections which can be used to iterate over the collection and transform the collection. Transforms include mapping each element to a new value, filtering out certain values and reducing the collection down to a single value.

Where to go from here?

Closures and functions are the fundamental types for storing your code into reusable pieces. Aside from declaring them and calling them, you've also seen how useful they are when passing them around as parameters to *other* functions and closures.

Why not stick around and find out if you've really learned this stuff? Check out the challenges below to test your knowledge before moving on.

Challenges

Challenge 1: Repeating yourself

Your first challenge is to write a function that will run a given closure a given number of times.

Declare the function like so:

```
func repeatTask(times: Int, task: () -> Void)
```

The function should run the task closure, `times` number of times.

Use this function to print "Swift Apprentice is a great book!" 10 times.

Challenge 2: Closure sums

In this challenge, you're going to write a function that you can reuse to create different mathematical sums.

Declare the function like so:

```
func mathSum(length: Int, series: (Int) -> Int) -> Int
```

The first parameter, `length`, defines the number of values to sum. The second parameter, `series`, is a closure that can be used to generate a series of values. `series` should have a parameter that is the position of the value in the series and return the value at that position.

`mathSum` should calculate `length` number of values, starting at position 1, and return their sum.

Use the function to find the sum of the first 10 square numbers, which equals 385. Then use the function to find the sum of the first 10 Fibonacci numbers, which equals 143. For the Fibonacci numbers, you can use the function you wrote in the functions chapter — or grab it from the solutions if you're unsure your solution is correct.

Challenge 3: Functional ratings

In this final challenge, you will have a list of app names with associated ratings they've been given. Note — these are all fictional apps!

Create the data dictionary like so:

```
let appRatings = [  
    "Calendar Pro": [1, 5, 5, 4, 2, 1, 5, 4],  
    "The Messenger": [5, 4, 2, 5, 4, 1, 1, 2],  
    "Socialise": [2, 1, 2, 2, 1, 2, 4, 2]  
]
```

First, create a dictionary called `averageRatings` which will contain a mapping of app names to average ratings. Use `forEach` to iterate through the `appRatings` dictionary, then use `reduce` to calculate the average rating and store this rating in the `averageRatings` dictionary.

Finally, use `filter` and `map` chained together to get a list of the app names whose average rating is greater than 3.

Chapter 16: Strings

By Matt Galloway

So far you have briefly seen what the type `String` has to offer for representing text. Text is an extremely common data type: people's names; their addresses; the words of a book. All of these are examples of text that an app might need to handle. It's worth having a deeper understanding of how `String` works and what it can do.

This chapter deepens your knowledge of strings in general, and more specifically how strings work in Swift. Swift is one of the few language that handles Unicode characters correctly while maintaining maximum predictable performance.

Strings as collections

In Chapter 2 you learned the basics of what a string is, and you learned what character sets and code points are. To recap, they define the mapping numbers to the character it represents. It's now time to look further into the `String` type.

It's pretty easy to conceptualize a string as a collection of characters. Because strings are collections, you can do things like this:

```
let string = "Matt"
for char in string {
    print(char)
}
```

This will print out every character of `Matt` individually. Simple, eh?

You can also use other collection operations, such as:

```
let stringLength = string.count
```

This will give you the length of the string.

Now imagine you want to get the fourth character in the string. You may think to do something like this:

```
let fourthChar = string[3]
```

However, if you did this you would receive the following error message:

```
'subscript' is unavailable: cannot subscript String with an Int, see the documentation comment for discussion
```

Why is that? It's time to take a detour further into how strings work by introducing what a **grapheme cluster** is.

Grapheme clusters

As you know, a string is made up of a collection of Unicode characters. Until now, you have considered one code point to precisely equal one character, and vice versa. However the term "character" is fairly loose.

It may come as a surprise, but there are two ways to represent some characters. One example is the é in café, which is an e with an acute accent. You can represent this character with either a single character or with two.

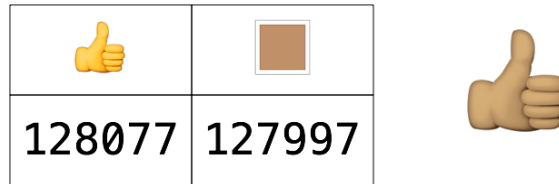
The single character to represent this is code point 233. The two-character case is an e on its own followed by an acute accent **combining character**, which is a special character that modifies the previous character.

So you can represent the e with an acute accent by either of these means:

é	e	'
233	101	769

The combination of these two characters in the second diagram forms what is known as a **grapheme cluster** defined by the Unicode standard. When you think of a character, you're actually probably thinking of a grapheme cluster. Grapheme clusters are represented by the Swift type `Character`.

Another example of combining characters are the special characters used to change the skin color of certain emojis. For example:



Here, the thumbs up emoji is followed by a skin tone combining character. On platforms that support it, including iOS and macOS, the rendered emoji is a single thumbs up character with the skin tone applied.

Let's now take a look at what this means for strings when they are used as collections. Consider the following code:

```
let cafeNormal = "café"
let cafeCombining = "cafe\u{0301}"

cafeNormal.count // 4
cafeCombining.count // 4
```

Both of these counts turn out to equal 4. This is because Swift considers a string as a collection of grapheme clusters. The astute reader will also notice that this means that finding out the length of a string takes linear time, because you need to go through all characters to determine how many grapheme clusters there are. You cannot simply know from looking at how big the string is in memory.

Note: In the code above, the acute accent combining character is written using the Unicode shorthand, which is `\u` followed by the code point in hexadecimal, in braces. You can use this shorthand to write any Unicode character. I had to use it here for the combining character because there's no way to type this character on my keyboard!

However there is a way to obtain access to the underlying Unicode code points in the string. You do this through the `unicodeScalars` **view** on the string. This view is also a collection itself. So you can do the following:

```
cafeNormal.unicodeScalars.count // 4
cafeCombining.unicodeScalars.count // 5
```

As you can see, in this case you are seeing the difference in the counts as you would expect.

You can iterate through this Unicode scalars view like so:

```
for codePoint in cafeCombining.unicodeScalars {  
    print(codePoint.value)  
}
```

This will print the following list of numbers, as expected:

```
99  
97  
102  
101  
769
```

Indexing strings

As you saw earlier, indexing into a string to get a certain character (err, I mean grapheme cluster) is not as simple as using an integer subscript. This is because Swift wants you to be aware of what's going on under the hood and so makes the syntax a little more verbose.

You have to operate on the specific string index type in order to index into strings. For example, you obtain the index that represents the start of the string like so:

```
let firstIndex = cafeCombining.startIndex
```

If you option-click on `firstIndex` in a playground you will notice that it is of type `String.Index` and is not an integer.

You can then use this value to obtain the `Character` (grapheme cluster) at that index, like so:

```
let firstChar = cafeCombining[firstIndex]
```

In this case, `firstChar` will of course be "c". The type of this value is **Character** which is a grapheme cluster.

Similarly, you can obtain the last grapheme cluster like so:

```
let lastIndex = cafeCombining.endIndex  
let lastChar = cafeCombining[lastIndex]
```

But if you do this you'll get a fatal error:

```
fatal error: Can't form a Character from an empty String
```

This is because the `endIndex` is actually 1 past the end of the string. So to obtain the last character you need to do this:

```
let lastIndex = cafeCombining.index(before: cafeCombining.endIndex)
let lastChar = cafeCombining[lastIndex]
```

Here you're obtaining the index just before the end index and then obtaining the character at that index. Alternatively you could offset from the first character like so:

```
let fourthIndex = cafeCombining.index(cafeCombining.startIndex, offsetBy: 3)
let fourthChar = cafeCombining[fourthIndex]
```

In this case, `fourthChar` is `é` as expected.

But as you know, the `é` in that case is actually made up of multiple code points. You can access these code points on the `Character` type in the same way as you can on `String`, through the `unicodeScalars` view. So you can do this:

```
fourthChar.unicodeScalars.count // 2
fourthChar.unicodeScalars.forEach { codePoint in
    print(codePoint.value)
}
```

This time you're using the `forEach` function to iterate through the Unicode scalars view. The count is 2 and as expected, the loop prints out:

```
101
769
```

Equality with combining characters

Combining characters make equality of strings a little trickier. For example, consider the word **café** written once using the single `é` character, and once using the combining character, like so:

c	a	f	é
99	97	102	233

c	a	f	e	'
99	97	102	101	769

These two strings are of course logically equal. When they are printed onscreen, they use the same **glyph** and look exactly the same. But they are represented inside the computer in different ways. Many programming languages would consider these strings to be unequal, because those languages work by comparing the code points one by one. Swift, however, considers these strings to be equal by default. Let's see that in action:

```
let equal = cafeNormal == cafeCombining
```

In this case, `equal` is `true`, because the two strings are logically the same.

String comparison in Swift uses a technique known as **canonicalization**. Say that three times fast! Before checking equality, Swift canonicalizes both strings, which means they're converted to use the same special character representation.

It doesn't matter which way Swift does the canonicalization — using the single character or using the combining character — as long as both strings get converted to the same style. Once the canonicalization is complete, Swift can compare individual characters to check for equality.

The same canonicalization comes into play when considering how many characters are in a certain string, which you saw earlier where `café` using the single `é` character and `café` using the `e` plus combining accent character had the same length.

Strings as bi-directional collections

Sometimes you want to reverse a string. Often this is so you can iterate through it backwards. Fortunately, Swift has a rather simple way to do this, through a method called `reversed()` like so:

```
let name = "Matt"
let backwardsName = name.reversed()
```

But what is the type of `backwardsName`? If you said `String`, then you would be wrong. It is actually a `ReversedCollection<String>`. This is a rather clever optimization that Swift makes. Instead of it being a concrete `String`, it is actually a **reversed collection**. This is a thin wrapper around any collection which allows you to use the collection as if it were the other way round, without incurring any more memory usage.

You can then access the Characters in the backwards string just as you would any other string, like so:

```
let secondCharIndex = backwardsName.index(backwardsName.startIndex,
offsetBy: 1)
let secondChar = backwardsName[secondCharIndex] // "t"
```

But what if you actually want a string? Well you can do that by initializing a `String` from the reversed collection, like so:

```
let backwardsNameString = String(backwardsName)
```

This will create a new `String` from the reversed collection. But when you do this, you end up making a reversed copy of the original string with its own memory storage. So if you don't actually need the whole reversed string then staying in the reversed collection domain will save memory space.

Substrings

Another thing that you often need to do when manipulating strings is to generate substrings. That is, pull out a part of the string into its own value. This can be done in Swift using a subscript that takes a range of indices.

For example, consider the following code:

```
let fullName = "Matt Galloway"
let spaceIndex = fullName.index(of: " ")!
let firstName = fullName[fullName.startIndex..
```

This code finds the index that represents the first space (using a force unwrap here because you know one exists). Then it uses a range to find the grapheme clusters between the start index and the index of the space (not including the space).

Now is a good time to introduce a new type of range that you haven't seen before: the **open-ended range**. This type of range only takes one index and assumes the other is either the start or the end of the collection.

That last line of code can be rewritten as follows using an open-ended range:

```
let firstName = fullName[..<spaceIndex] // "Matt"
```

This time we omit the `fullName.startIndex` and Swift will infer that this is what you mean.

Similarly, you can also use a one-sided range to start at a certain index and go to the end of the collection, like so:

```
let lastName = fullName[fullName.index(after: spaceIndex)...] //
"Galloway"
```

There's something interesting to point out with substrings. If you look at their type, then you will see they are of type `Substring` rather than `String`. Just like with the reversed string, you can force this `Substring` into a `String` by doing the following:

```
let lastNameString = String(lastName)
```

The reason for this extra `Substring` type is a cunning optimization. A `Substring` shares the storage with its parent `String` that it was sliced from. This means that when you're in the process of slicing a string you use no extra memory. Then, when you want the substring as a `String` you explicitly create a new string and the memory is copied into a new buffer for this new string.

The designers of Swift could have made this copy behavior by default. However, by having the separate type `Substring`, Swift makes it very explicit what is happening. The good news is that `String` and `Substring` share almost all of the same capabilities. You might not even realize which type you are using until you return or pass your `Substring` to another function that requires a `String`. In this case, you can simply initialize a new `String` from your `Substring` explicitly.

It is hopefully clear by now that Swift is very opinionated about strings, and very deliberate in the way it implements them. This is important because strings are complex beasts, but they are very frequently used. Therefore getting the API right for them is important.

Encoding

Up until now, this chapter hasn't talked about how strings are stored. You're going to look at that now.

Strings are made up of a collection of Unicode code points. These code points range from the number 0 up to 1114111 (or 0x10FFFF in hexadecimal). This means that the maximum number of bits you need to represent a code point is 21. However, if you are only ever using low code points, such as if your text contains only Latin characters, then you can get away with using only 8 bits per code point.

Numeric types in most programming languages come in sizes of addressable, powers-of-2 bits, such as 8-bits, 16-bits and 32-bits. This is because computers are made of billions of transistors that are either off or on; they just love powers of 2!

When choosing how to store strings, you could choose to store every individual code point in a 32-bit type, such as `UInt32`. So your `String` type would be backed by a `[UInt32]` (a `UInt32` array). Each of these `UInt32`s is what is known as a **code unit**.

However, you would be wasting space because not all those bits are needed, especially if the string uses only low code points.

This choice of how to store strings is known as the string's **encoding**. This particular scheme described above is known as **UTF-32**. However, because it has inefficient memory usage it is very rarely used.

UTF-8

A much more common scheme is called **UTF-8**. This uses 8-bit code units instead. One reason for UTF-8's popularity is because it is fully compatible with the venerable, English-only, 7-bit ASCII encoding. But how do you store code points that need more than 8 bits?! Herein lies the magic of the encoding.

If the code point requires up to 7 bits, it is represented by simply one code unit and is identical to **ASCII**. But for code points above 7 bits, a scheme comes into play that uses up to 4 **code units** to represent the code point.

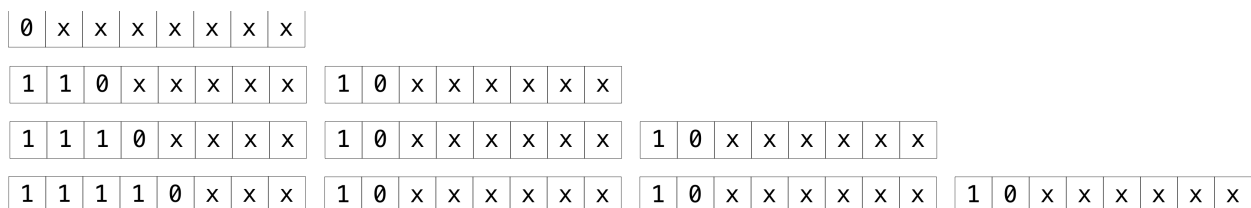
For code points of 8 to 11 bits, 2 code units are used. The first code unit's initial 3 bits are 110. The remaining 5 bits are the first 5 bits of the code point. The second code unit's initial 2 bits are 10 and the remaining 6 bits are the remaining 6 bits of the code point.

For example, the code point `0x00BD` represents the $\frac{1}{2}$ character. In binary this is `10111101`, and uses 8 bits. In UTF-8, this would comprise 2 code units of `11000010` and `10111101`.

To illustrate this, consider the following diagram:



Of course, code points higher than 11 bits are also supported. 12- to 16-bit code points use 3 UTF-8 code units, and 17- to 21-bit code points use 4 UTF-8 code units, according to the following scheme:



The “x”s are replaced with the bits from the code points.

In Swift, you can access the UTF-8 string encoding through the `utf8` view. For example, consider the following code:

```
let char = "\u{00bd}"
for i in char.utf8 {
    print(i)
}
```

The `utf8` view is a collection, just like the `unicodeScalars` view. Its values are the UTF-8 code units that make up the string. In this case, it's a single character, namely the one that we discussed above.

The above code will print the following:

```
194
189
```

If you pull out your calculator (or have a fantastic mental arithmetic mind) then you can validate that these are 11000010 and 10111101 respectively, as you expected!

Now consider a more complicated example which you'll refer back to later in this section. Take the following string:

+½🙄

And iterate through the UTF-8 code units it contains:

```
let characters = "+\u{00bd}\u{21e8}\u{1f643}"
for i in characters.utf8 {
    print("\(i) : \(String(i, radix: 2))")
}
```

This time the `print` statement will print out both the decimal number and the number in binary. It prints the following (newlines added to split grapheme clusters):

```
43 : 101011

194 : 11000010
189 : 10111101

226 : 11100010
135 : 10000111
168 : 10101000

240 : 11110000
159 : 10011111
153 : 10011001
131 : 10000011
```

Feel free to verify that these are indeed correct. Notice that the first character used 1 code unit, the second used 2 code units, and so on.

UTF-8 is therefore much more compact than UTF-32. For this string, you used 10 bytes to store the 4 code points. In UTF-32 this would be 16 bytes (4 bytes per code unit, 1 code unit per code point, 4 code points).

There is a downside to UTF-8 though. To handle certain string operations you need to inspect every byte. For example, if you wanted to jump to the `_n_th` code point, you would need to inspect every byte until you have gone past $n-1$ code points. You cannot simply jump into the buffer because you don't know how far you have to jump.

UTF-16

There is another encoding that is useful to introduce, namely **UTF-16**. Yes, you guessed it. It uses 16-bit code units!

This means that code points that are up to 16 bits use 1 code unit. But how are code points of 17 to 21 bits represented? These use a scheme known as **surrogate pairs**. These are 2 UTF-16 code units that, when next to each other, represent a code point from the range above 16-bits.

There is a space within Unicode reserved for these surrogate pair code points. They are split into low and high surrogates. The high surrogates range from `0xD800` to `0xDBFF`, and the low surrogates range from `0xDC00` to `0xDFFF`. You may think that sounds the wrong way round, but the high and low here refers to the bits from the original code point that are represented by this surrogate.

Take the upside-down face emoji from the string you saw earlier. Its code point is `0x1F643`. To find out the surrogate pairs for this code point, you apply the following algorithm:

1. Subtract `0x10000` to give `0xF643`, or `0000 1111 0110 0100 0011` in binary.
2. Split these 20 bits into two. This gives you `0000 1111 01` and `10 0100 0011`.
3. Take the first and add `0xD800` to it, to give `0xD83D`. This is your **high surrogate**.
4. Take the second and add `0xDC00` to it, to give `0xDE43`. This is your **low surrogate**.

So in UTF-16, that upside-down face emoji is represented by the code unit `0xD83D` followed by `0xDE43`. Neat!

Just as with UTF-8, Swift allows you to access the UTF-16 code units through the `utf16` view, like so:

```
for i in characters.utf16 {  
    print("\(i) : \(String(i, radix: 2))")  
}
```

In this case, the following is printed (again with newlines added to split grapheme clusters):

```
43 : 101011  
189 : 10111101  
8680 : 10000111101000  
55357 : 1101100000111101  
56899 : 1101111001000011
```

As you can see, the only code point that needs to use more than one code unit is the last one, which is your upside-down face emoji. And as expected, the values are correct!

So with UTF-16, your string this time uses 10 bytes (5 code units, 2 bytes per code unit). This turns out to be the same as UTF-8. In general, the memory usage with UTF-8 and UTF-16 will be different. For example, strings that are made up of code points of 7 bits or less will take up twice the space in UTF-16 as compared to UTF-8.

For a string made up of code points 7 bits or less, the string has to be entirely made up of those Latin characters contained in that range. Even the “£” sign is not in this range! So often the memory usage of UTF-16 and UTF-8 are comparable.

Swift string views make the `String` type encoding agnostic and is one of the only languages that does this. Internally it actually uses UTF-16. This is because it hits the sweet spot between memory usage and complexity of operations.

Converting indexes between encoding views

As you saw earlier, you use indexes to access grapheme clusters in a string. For example, using the same string from above, you can do the following:

```
let arrowIndex = characters.index(of: "\u{21e8}")!  
characters[arrowIndex] // ➡
```

Here, `arrowIndex` is of type `String.Index` and it is used to obtain the `Character` at that index.

You can convert this index into the index relating to the start of this grapheme cluster in the `unicodeScalars`, `utf8` and `utf16` views. You do that using the `samePosition(in:)` method on `String.Index`, like so:

```
if let unicodeScalarsIndex = arrowIndex.samePosition(in:
characters.unicodeScalars) {
    characters.unicodeScalars[unicodeScalarsIndex] // 8680
}

if let utf8Index = arrowIndex.samePosition(in: characters.utf8) {
    characters.utf8[utf8Index] // 226
}

if let utf16Index = arrowIndex.samePosition(in: characters.utf16) {
    characters.utf16[utf16Index] // 8680
}
```

`unicodeScalarsIndex` is of type `String.UnicodeScalarView.Index`. This grapheme cluster is represented by only one code point, so in the `unicodeScalars` view, the scalar returned is the one and only code point. If the `Character` were made up of two code points, such as the “e” and combining accent you saw earlier, the scalar returned in the code above would be just the “e”.

Likewise, `utf8Index` is of type `String.UTF8View.Index` and the value at that index is the first UTF-8 code unit used to represent this code point. The same goes for the `utf16Index`, which is of type `String.UTF16View.Index`.

Key points

- Strings are collections of `Character` types.
- A `Character` is **grapheme cluster** and is made up of one or more **code points**.
- A **combining character** is a character which alters the previous character in some way.
- You use special (non-integer) indexes to subscript into the string to a certain grapheme cluster.
- Swift’s use of **canonicalization** ensures that the comparison of strings accounts for combining characters.

- Slicing a string yields a substring with type `Substring`. This shares its storage with its parent `String`.
- You can convert from a `Substring` to a `String` by initializing a new `String` and passing the `Substring`.
- Swift `String` has a view called `unicodeScalars`, which is itself a collection of the individual Unicode code points that makes up the string.
- There are multiple ways to encode a string. UTF-8 and UTF-16 are the most popular.
- The individual parts of an encoding are called **code units**. UTF-8 uses 8-bit code units, and UTF-16 uses 16-bit code units.
- Swift's `String` has views called `utf8` and `utf16` which are collections allowing you to obtain the individual code units in the given encoding.

Where to go from here?

Strings are a fundamental part of any programming language, as text forms the bulk of the data that most programs will manipulate. In this chapter you have learned how many of the features of Swift's `String` type work.

In the coming chapters you will take a look at how to create your own types.

Challenges

Challenge 1

Write a function which take a string and prints out the count of each character in the string.

For bonus points, print them ordered by the count of each character.

For even more bonus points, print it as a nice histogram. You could use `#` characters to draw the bars.

Challenge 2

Write a function which tells you how many words there are in a string. Do it without splitting the string but rather iterating through the string yourself.

Challenge 3

Write a function which takes a string which looks like "Galloway, Matt" and returns one which looks like "Matt Galloway"; i.e. the string goes from "<LAST_NAME>, <FIRST_NAME>" to "<FIRST_NAME> <LAST_NAME>".

Challenge 4

There exists a method on string named `components(separatedBy:)` that will split the string into chunks delimited by the given string and return an array containing the results.

Your challenge is to implement this yourself.

Hint: There exists a view on `String` named `indices` which lets you iterate through all the indices (of type `String.Index`) in the string. You will need to use this.

Challenge 5

Write a function which takes a string and returns a version of it with each individual word reversed.

For example, if the string is "My dog is called Rover" then the resulting string would be "yM god si dellac revoR".

Try to do it by iterating through the `indices` of the string until you find a space and then reversing what was before it. Build up the result string by continually doing that as you go through the string.

Hint: You'll need to do a similar thing to the previous challenge, but reverse the word each time. Can you explain why this is better in terms of memory usage than using the function you created in the previous challenge?

Chapter 17: Structures

By Ben Morrow

You’ve covered some fundamental building blocks of Swift. With variables, conditionals, strings, functions and collections, you’re ready to conquer the world! Well, almost.

Most programs that perform complex tasks benefit from higher levels of abstraction. In addition to an `Int`, `String` or `Array`, most programs use new types specific to the domain of the task at hand. Keeping track of photos or contacts, for example, demands more than the simple types you’ve seen so far.

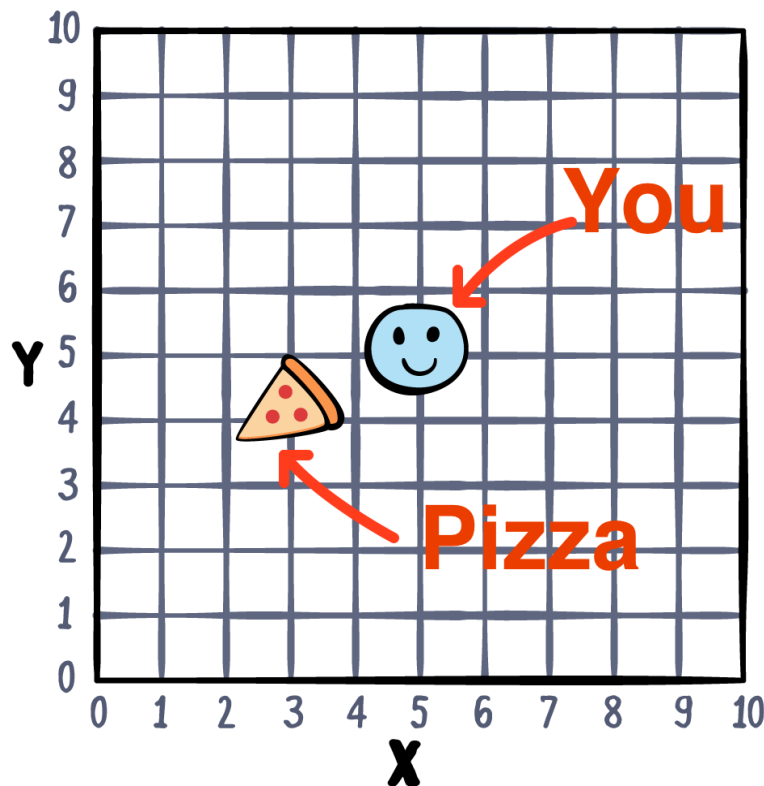
This chapter introduces **structures**, which are the first **named type** you will learn about. Structures are types that can store named properties and define their own behaviors. Like a `String`, `Int` or `Array`, you can define your own structures to create named types to use in your code. By the end of this chapter, you’ll know how to define and use your own structures.

You begin your adventure into custom types with pizza.



Introducing structures

Imagine you live in a town called Pizzaville. As you might expect, Pizzaville is known for its amazing pizza. You own the most popular (and fastest!) pizza delivery restaurant in Pizzaville — “Swift Pizza”.



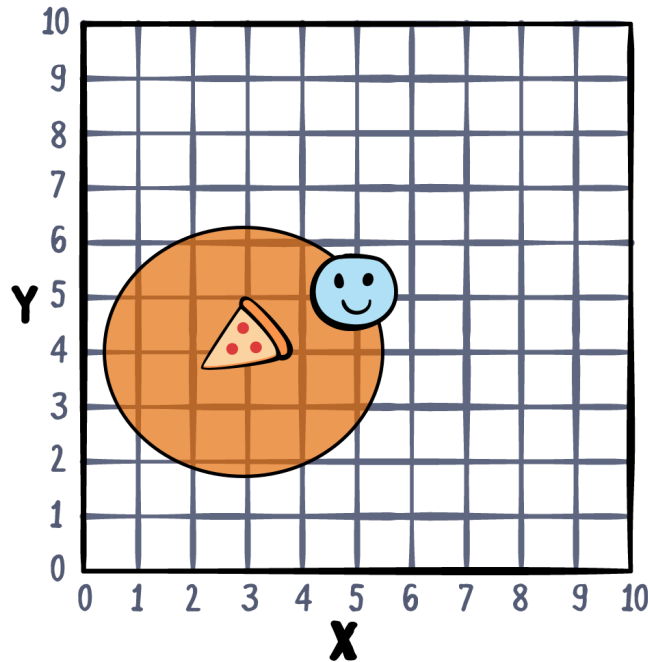
As the owner of a single restaurant, you still have a limited delivery area. You want to write a program that calculates if a potential customer is within range for your delivery drivers. The first version of your program might look something like this:

```
let restaurantLocation = (2, 4)
let restaurantRange = 2.5

// Pythagorean Theorem 🎓
func distance(from source: (x: Int, y: Int), to target: (x: Int, y: Int))
-> Double {
    let distanceX = Double(source.x - target.x)
    let distanceY = Double(source.y - target.y)
    return (distanceX * distanceX +
            distanceY * distanceY).squareRoot()
}

func isInDeliveryRange(location: (x: Int, y: Int)) -> Bool {
    let deliveryDistance = distance(from: location, to: restaurantLocation)
    return deliveryDistance < restaurantRange
}
```

Simple enough, right? `distance(from:to:)` will calculate how far away you are from your pizza. `isInDeliveryRange(location:)` will return `true` only if you're not too far away.



A successful pizza delivery business may eventually expand to include multiple locations, which adds a minor twist to the deliverable calculator. Replace your existing code with the following:

```
let restaurantLocation = (2, 4)
let restaurantRange = 2.5

let otherRestaurantLocation = (7, 8)
let otherRestaurantRange = 1.5

// Pythagorean Theorem 🎓
func distance(from source: (x: Int, y: Int), to target: (x: Int, y: Int))
-> Double {
    let distanceX = Double(source.x - target.x)
    let distanceY = Double(source.y - target.y)
    return (distanceX * distanceX +
            distanceY * distanceY).squareRoot()
}

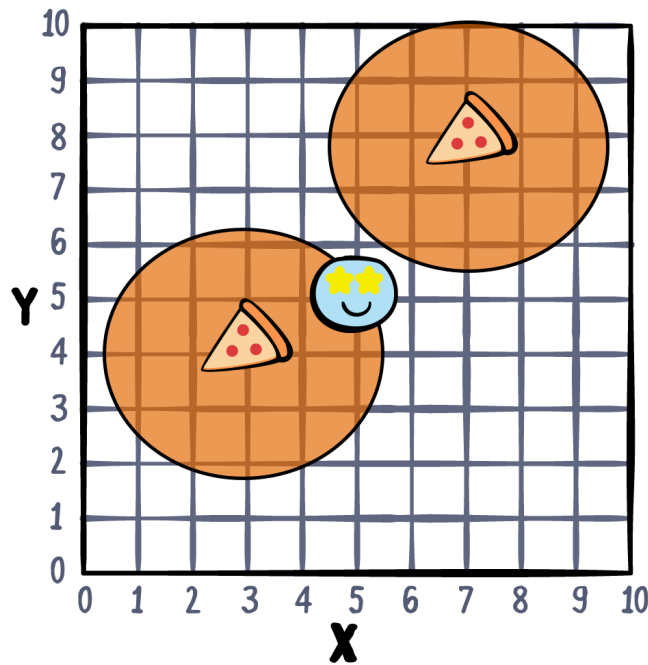
func isInDeliveryRange(location: (x: Int, y: Int)) -> Bool {
    let deliveryDistance =
        distance(from: location,
                to: restaurantLocation)

    let secondDeliveryDistance =
        distance(from: location,
                to: otherRestaurantLocation)

    return deliveryDistance < restaurantRange ||
           secondDeliveryDistance < otherRestaurantRange
}
```

`isInDeliveryRange(location:)` checks both locations to see if you can get your pizza from either one.

Eventually, the rising number of customers will force the business to expand, and soon it might grow to a total of 10 stores! Then what? Do you keep updating your function to check against all these sets of coordinates and ranges?



You might briefly consider creating an array of x/y coordinate tuples to keep track of your pizza restaurants, but that would be both difficult to read and maintain. Fortunately, Swift has additional tools to help you simplify the problem.

Your first structure

Structures are one of the named types in Swift that allow you to encapsulate related properties and behaviors. You can declare a new type, give it a name, and then use it in your code.

In the example of the pizza business, you've been using x/y coordinate tuples to represent locations.

As a first example of structures, promote locations from tuples to a structure type:

```
struct Location {  
    let x: Int  
    let y: Int  
}
```

This block of code demonstrates the basic syntax for defining a structure. In this case, the code declares a type named `Location` that combines both `x` and `y` coordinates.

The basic syntax begins with the `struct` keyword followed by the name of the type and a pair of curly braces. Everything between the curly braces is a *member* of the struct.

In `Location`, both members, `x` and `y`, are **properties**. Properties are constants or variables that are declared as part of a type. Every instance of the type will have these properties. This means that in our example, every `Location` will have both an `x` and a `y` property.

You can instantiate a structure and store it in a constant or variable just like any other type you've worked with:

```
let storeLocation = Location(x: 2, y: 4)
```

To create the `Location` value, you use the name of the type along with a parameter list in parentheses. This parameter list provides a way to specify the values for the properties `x` and `y`. This is an example of an **initializer**.

Initializers enforce that all properties are set before you start using them. This is one of the key safety features of Swift. Accidentally using uninitialized variables is a big source of bugs in other languages. Another handy Swift feature is that you didn't need to declare this initializer in the `Location` type. Swift automatically provides initializers for structures with all the properties in the parameter list.

You'll learn a lot more about initializers in Chapter 12, "Methods".

You may remember that there's also a range involved, and now that the pizza business is expanding, there may be different ranges associated with different restaurants. You can create another struct to represent the delivery area of a restaurant, like so:

```
struct DeliveryArea {  
    let center: Location  
    var radius: Double  
}  
  
var storeArea = DeliveryArea(center: storeLocation, radius: 4)
```

Now there's a new structure named `DeliveryArea` that contains a constant `center` property along with a variable `radius` property. As you can see, you can have a structure value inside a structure value; here, you use the `Location` type as the type of the `center` property of the `DeliveryArea` struct.

Mini-exercise

Write a structure that represents a pizza order. Include toppings, size and any other option you'd want for a pizza.

Accessing members

With your `DeliveryArea` defined and an instantiated value in hand, you may be wondering how you can *use* these values. Just as you have been doing with `Strings`, `Arrays`, and `Dictionaries`, use the **dot syntax** to access members:

```
print(storeArea.radius) // 4.0
```

You can even access *members of members* using dot syntax:

```
print(storeArea.center.x) // 2
```

Similar to how you can read values with dot syntax, you can also *assign* them. If the delivery radius of one pizza location becomes larger, you could assign the new value to the existing property:

```
storeArea.radius = 250
```

The semantics of constants and variables play a significant role in determining if a property can be assigned to. In this case, you can assign to `radius` because you declared it with `var`. On the other hand, you declared `center` with `let`, so you can't modify it. Your `DeliveryArea` struct allows a pizza restaurant's delivery range to be changed, but not its location!

**ONE DAY I'LL HAVE
A DELIVERY CAR!**



In addition to choosing whether your properties should be variable or constants, you must also declare the structure itself as a variable if you want to be able to modify it after it is initialized:

```
let fixedArea = DeliveryArea(center: storeLocation, radius: 4)

// Error: Cannot assign to property
fixedArea.radius = 250
```

That code causes the compiler to emit an error. Change `fixedArea` from a `let` constant to a `var` variable to make it mutable.

Now you've learned how to control the mutability of the properties in your structure.

Mini-exercise

Rewrite `isInDeliveryRange` to use `Location` and `DeliveryArea`.

Introducing methods

Using some of the capabilities of structures, you could now make a pizza delivery range calculator that looks something like this:

```
let areas = [
    DeliveryArea(center: Location(x: 2, y: 4), radius: 2.5),
    DeliveryArea(center: Location(x: 9, y: 7), radius: 4.5)
]

func isInDeliveryRange(_ location: Location) -> Bool {
    for area in areas {
        let distanceToStore =
            distance(from: (area.center.x, area.center.y),
                    to: (location.x, location.y))

        if distanceToStore < area.radius {
            return true
        }
    }
    return false
}

let customerLocation1 = Location(x: 8, y: 1)
let customerLocation2 = Location(x: 5, y: 5)

print(isInDeliveryRange(customerLocation1)) // false
print(isInDeliveryRange(customerLocation2)) // true
```

In this example, there's an array `areas` and a function that uses that array to determine if a customer's location is within any of these areas.

Being in range is something you want to know about a particular restaurant. It'd be great if `DeliveryArea` could tell you if the restaurant could deliver to a location.

Much like a structure can have constants and variables, it can also define its *own* functions. In your playground, locate the implementation of `DeliveryArea`. Just before the closing curly brace, add the following code:

```
func contains(_ location: Location) -> Bool {
    let distanceFromCenter =
        distance(from: (center.x, center.y),
                 to: (location.x, location.y))

    return distanceFromCenter < radius
}
```

This defines a function `contains`, which is now a member of `DeliveryArea`. Functions that are members of types are called **methods**. Notice how `contains` uses the `center` and `radius` properties of the current location. This implicit access to properties and other members makes methods different from regular functions. You'll learn more about methods in Chapter 12.

Just like other members of structures, you can use dot syntax to access a method:

```
let area = DeliveryArea(center: Location(x: 5, y: 5), radius: 4.5)
let customerLocation = Location(x: 2, y: 2)
area.contains(customerLocation) // true
```

Mini-exercises

- Change `distance(from:to:)` to use `Locations` as parameters instead of x-y tuples.
- Change `contains(_:)` to call the new `distance(from:to:)` with `Locations`.
- Add a method `overlaps(with:)` on `DeliveryArea` that can tell you if the area overlaps with another area.

Structures as values

The term **value** has an important meaning when it comes to structures in Swift, and that's because structures create what are known as **value types**.

A value type is a type whose instances are *copied* on assignment.

```
var a = 5
var b = a
print(a) // 5
print(b) // 5
```

```
a = 10
print(a) // 10
print(b) // 5
```

This **copy-on-assignment** behavior means that when `a` is assigned to `b`, the value of `a` is copied into `b`. That's why it's important read `=` as "assign", not "is equal to" (which is what `==` is for).

How about the same principle, except with the `DeliveryArea` struct:

```
var area1 = DeliveryArea(center: Location(x: 2, y: 4), radius: 2.5)
var area2 = area1
print(area1.radius) // 2.5
print(area2.radius) // 2.5

area1.radius = 4
print(area1.radius) // 4.0
print(area2.radius) // 2.5
```

As with the previous example, `area2.radius` didn't pick up the new value set in `area1.radius`. This demonstrates the **value semantics** of working with structures. When you assign `area2` the value of `area1`, it gets an exact copy of this value. `area1` and `area2` are still completely independent!

Structures everywhere

You saw how the `Location` struct and a simple `Int` share the same copy-on-assignment behavior. This is because they are both value types, and both have value semantics.

You know structures represent values, so what exactly is an `Int` then? If you were to look at the definition of `Int` in the Swift library, you might be a bit surprised:

```
public struct Int : FixedWidthInteger, SignedInteger {
    // ...
}
```

The `Int` type is *also* a structure. In fact, many of the standard Swift types are defined as structures, such as: `Double`, `String`, `Bool`, `Array` and `Dictionary`. As you will learn in future chapters, the value semantics of structs provide many other advantages over their reference type counterparts that make them ideal for representing core Swift types.

Conforming to a protocol

You may have noticed some unfamiliar parts to the `Int` definition from the Swift standard library above. The types `FixedWidthInteger` and `SignedInteger` appear right after the declaration of `Int`:

```
public struct Int : FixedWidthInteger, SignedInteger {  
    // ...  
}
```

These types are known as *protocols*. By putting them after a colon when `Int` is declared, you are declaring that `Int` *conforms* to these protocols.

Protocols contain a set of requirements that conforming types **must** satisfy. A simple example from the standard library is `CustomStringConvertible`:

```
public protocol CustomStringConvertible {  
    /// A textual representation of this instance.  
    public var description: String { get }  
}
```

This protocol contains one property requirement: `description`. This description is documented as “A textual representation of this instance.”

If you were to modify `DeliveryArea` to conform to `CustomStringConvertible`, you would be required to add a `description` property with a “textual representation” of the instance. Try this now. Change `DeliveryArea` to:

```
struct DeliveryArea: CustomStringConvertible {  
    let center: Location  
    var radius: Double  
  
    var description: String {  
        return ""  
        Area with center: x: \(center.x) - y: \(center.y),  
        radius: \(radius)  
        ""  
    }  
  
    func contains(_ location: Location) -> Bool {  
        let distanceFromCenter =  
            distance(from: (center.x, center.y),  
                    to: (location.x, location.y))  
        return distanceFromCenter < radius  
    }  
}
```

The value of the `description` property is based on the center and current radius. This is made possible by implementing it as a *computed* property.

You'll learn all about computed properties — and more — in the next chapter!

So what exactly does conforming to a protocol do? Because any type conforming to `CustomStringConvertible` must define `description`, you can call `description` on any instance of any type that conforms to `CustomStringConvertible`. The Swift standard library takes advantage of this with the `print()` function. That function will use `description` in the console instead of a rather noisy default description:

```
print(area1) // Area with center: x: 2 - y: 4, radius: 4.0
print(area2) // Area with center: x: 2 - y: 4, radius: 2.5
```

Any named type can use protocols to extend their behavior. In this case, you conformed your structure to a protocol defined in the Swift standard library. In Chapter 16, “Protocols”, you will learn more about defining, using, and conforming to protocols.

Key points

- Structures are named types you can define and use in your code.
- Structures are **value types**, which means their values are copied on assignment.
- You use dot syntax to access the members of named types such as structures.
- Named types can have their own variables and functions, which are called properties and methods.
- Conforming to a protocol requires implementing the properties and methods required by that protocol.

Where to go from here?

Thanks to value semantics and copying, structures are *safe*, so you'll never need to worry about values being shared and possibly being changed behind your back by another piece of code.

Structures are also very *fast* compared to their reference alternatives, which you'll learn about in Chapter 13, “Classes”.

Challenges

Challenge 1

Implement a "playable" version of Tic-Tac-Toe (OXO). The focus of this exercise is on using structures. Represent Xs with a global string constant "X" and Os with a global string constant "O". While X and O could be cleanly represented with an enumeration type, you will learn about those in Chapter 15, "Enumerations". For now, you can make the type called `BoardPiece` as an alias to `String` using `typealias`. The `BoardPiece` type can be used completely interchangeably with a `String`. While this doesn't afford much in the way of formal type safety, it does add to the readability and documentation of your code.

```
typealias BoardPiece = String
let X: BoardPiece = "X"
let O: BoardPiece = "O"
```

Challenge 2

Create a T-shirt structure that has size, color and material options. Provide methods to calculate the cost of a shirt based on its attributes.

Challenge 3

Write the engine for a Battleship-like game. If you aren't familiar with Battleship, see here: <http://bit.ly/2xJEPLM>

- Use an (x, y) coordinate system for your locations and model using a structure.
- Ships should also be modelled with structures. Record an origin, direction and length.
- Each ship should be able to report if a "shot" has resulted in a "hit".

Chapter 18: Properties

By Ben Morrow

In the last chapter, you learned that structures make you a more efficient programmer by grouping related properties and behaviors into structured types.

In the example below, the `Car` structure has two properties, both constants that store `String` values:

```
struct Car {  
    let make: String  
    let color: String  
}
```

Values like these are called **properties**. The two properties of `Car` are both **stored properties**, which means they store actual string values for each instance of `Car`.

Some properties calculate values rather than store them; that means there's no actual memory allocated for them, but they get calculated on-the-fly each time you access them. Naturally, these are called **computed properties**.

In this chapter, you'll learn about both kinds of properties. You'll also learn some other neat tricks in dealing with properties, such as how to monitor changes in a property's value and delay initialization of a stored property.

Stored properties

As you may have guessed from the example in the introduction, you're already familiar with many of the features of stored properties. To review, imagine you're building an address book. The common unit you'll need is a `Contact`:

```
struct Contact {  
    var fullName: String
```

```
} var emailAddress: String
```

You can use this structure over and over again, letting you build an array of contacts, each with a different value. The properties you want to store are an individual's full name and email address.



These are the properties of the Contact structure. You provide a data type for each, but opt not to assign a default value, because you plan to assign the value upon initialization. After all, the values will be different for each instance of Contact.

Remember that Swift automatically creates an initializer for you based on the properties you defined in your structure:

```
var person = Contact(fullName: "Grace Murray",  
                      emailAddress: "grace@navy.mil")
```

You can access the individual properties using dot notation:

```
let name = person.fullName // Grace Murray  
let email = person.emailAddress // grace@navy.mil
```

You can assign values to properties as long as they're defined as variables (and the instance is stored in a variable as well). When Grace married, she changed her last name:

```
person.fullName = "Grace Hopper"  
let grace = person.fullName // Grace Hopper
```

If you'd like to prevent a value from changing, you can define a property as a constant instead using `let`:

```
struct Contact {  
    var fullName: String  
    let emailAddress: String  
}
```

```
// Error: cannot assign to a constant
person.emailAddress = "grace@gmail.com"
```

Once you’ve initialized an instance of this structure, you can’t change `emailAddress`.

Default values

If you can make a reasonable assumption about what the value of a property should be when the type is initialized, you can give that property a default value.

It doesn’t make sense to create a default name or email address for a contact, but imagine there’s a new property type to indicate what kind of contact it is:

```
struct Contact {
    var fullName: String
    let emailAddress: String
    var type = "Friend"
}
```

By assignment a value in the definition of type, you give this property a default value. Any contact created will automatically be a friend, unless you change the value of `type` to something like “Work” or “Family”.

The downside is that the automatic initializer doesn’t notice default values, so you’ll still need to provide a value for each property unless you create your own custom initializer:

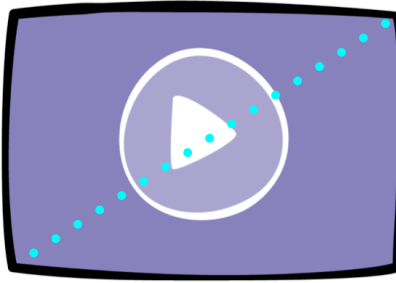
```
var person = Contact(fullName: "Grace Murray",
                     emailAddress: "grace@navy.mil",
                     type: "Friend")
```

You’ll learn more about creating initializers in the next chapter.

Computed properties

Stored properties are certainly the most common, but there are also properties that are computed, which simply means they perform a calculation before returning a value.

While a stored property can be a constant or a variable, a computed property must be defined as a variable. Computed properties must also include a type, because the compiler needs to know what to expect as a return value.



The measurement for a TV is the perfect use case for a computed property. The industry definition of the screen size of a TV isn't the screen's height or width, but its diagonal measurement:

```
struct TV {  
    var height: Double  
    var width: Double  
  
    // 1  
    var diagonal: Int {  
        // 2  
        let result = (height * height + width * width).squareRoot().rounded()  
        // 3  
        return Int(result)  
    }  
}
```

Let's go through this code one step at a time:

1. You use an `Int` type for your diagonal property. Although height and width are each a `Double`, TV sizes are usually advertised as nice, round numbers such as 50" rather than 49.52". Instead of the usual assignment operator `=` to assign a value as you would for a stored property, you use curly braces to enclose your computed property's calculation.
2. As you've seen before in this book, geometry can be handy; once you have the width and height, you can use the Pythagorean theorem to calculate the length of the diagonal. You use the `rounded` method to round the value with the standard rule: If it the decimal is 0.5 or above, it rounds up; otherwise it rounds down.
3. Now that you've got a properly rounded number, you return it as an `Int`. Had you converted `result` directly to `Int` without rounding first, the result would have been truncated, so 109.99 would have become 109.

Computed properties don't store any values; they simply return values based on calculations. From outside of the structure, a computed property can be accessed just like a stored property.

Test this with the TV size calculation:

```
var tv = TV(height: 53.93, width: 95.87)
let size = tv.diagonal // 110
```

You have a 110-inch TV. Let's say you decide you don't like the standard movie aspect ratio and would instead prefer a square screen. You cut off some of the screen width to make it equivalent to the height:

```
tv.width = tv.height
let diagonal = tv.diagonal // 76
```

Now you *only* have a 76-inch square screen. The computed property automatically provides the new value based on the new width.

**I'D BE HAPPY WITH
A TV HALF THAT SIZE**



Mini-exercise

Do you have a television or a computer monitor? Measure the height and width, plug it into a TV struct, and see if the diagonal measurement matches what you think it is.

Getter and setter

The computed property you wrote in the previous section is called a **read-only computed property**. It has a block of code to compute the value of the property, called the **getter**. It's also possible to create a **read-write computed property** with two blocks of code: a getter and a **setter**. This setter works differently than you might expect. As the computed property has no place to store a value, the setter usually sets one or more related *stored* properties indirectly:

```
var diagonal: Int {
    // 1
    get {
        // 2
        let result = (height * height + width * width).squareRoot().rounded()
        return Int(result)
    }
    set {
        // 3
        let ratioWidth = 16.0
```



```
    let ratioHeight = 9.0
    // 4
    let ratioDiagonal =
        (ratioWidth * ratioWidth + ratioHeight * ratioHeight).squareRoot()
    height = Double(newValue) * ratioHeight / ratioDiagonal
    width = height * ratioWidth / ratioHeight
  }
}
```

Here's what's happening in this code:

1. Because you want to include a setter, you now have to be explicit about which calculations comprise the getter and which the setter, so you surround each code block with curly braces and precede it with either `get` or `set`. This isn't required for read-only computed properties, as their single code block is implicitly a getter.
2. You use the same code as before to get the computed value.
3. For a setter, you usually have to make some kind of assumption. In this case, you provide a reasonable default value for the screen ratio.
4. The formulas to calculate a height and width, given a diagonal and a ratio, are a bit deep. You could work them out with a bit of time, but I've done the dirty work for you and provided them here. The important parts to focus on are:
 - The `newValue` constant lets you use whatever value was passed in during the assignment.
 - Remember, the `newValue` is an `Int`, so to use it in a calculation with a `Double`, you must first convert it to a `Double`.
 - Once you've done the calculations, you assign the height and width properties of the `TV` structure.

Now, in addition to setting the height and width directly, you can set them *indirectly* by setting the `diagonal` computed property. When you set this value, your setter will calculate and store the height and width.

Notice that there's no `return` statement in a setter — it only modifies the other stored properties. With the setter in place, you have a nice little screen size calculator:

```
tv.diagonal = 70
let height = tv.height // 34.32...
let width = tv.width // 61.01...
```

Now you can discover the biggest TV that will fit in your cabinet or on your shelf. :]

Type properties

In the previous section, you learned how to associate stored and computed properties with instances of a particular type. The properties on your instance of `TV` are separate from the properties on my instance of `TV`.

However, the type *itself* may also need properties that are common across all instances. These properties are called **type properties**.

Imagine you're building a game with many levels. Each level has a few attributes, or stored properties:

```
struct Level {
    let id: Int
    var boss: String
    var unlocked: Bool
}

let level1 = Level(id: 1, boss: "Chameleon", unlocked: true)
let level2 = Level(id: 2, boss: "Squid", unlocked: false)
let level3 = Level(id: 3, boss: "Chupacabra", unlocked: false)
let level4 = Level(id: 4, boss: "Yeti", unlocked: false)
```

You can use a type property to store the game's progress as the player unlocks each level. A type property is declared using `static`:

```
struct Level {
    static var highestLevel = 1
    let id: Int
    var boss: String
    var unlocked: Bool
}
```

Here, `highestLevel` is a property on `Level` itself rather than on the instances. That means you don't access this property on an instance:

```
// Error: you can't access a type property on an instance
let highestLevel = level3.highestLevel
```

Instead, you access it on the type itself:

```
let highestLevel = Level.highestLevel // 1
```

Using a type property means you can retrieve the same stored property value from anywhere in the code for your app or algorithm. The game's progress is accessible from any level or any other place in the game, like the main menu.

Property observers

For your `Level` implementation, it would be useful to automatically set the `highestLevel` when the player unlocks a new one. For that, you'll need a way to listen to property changes. Thankfully, there are a couple of **property observers** that get called before and after property changes.

A `willSet` observer is called when a property is about to be changed while a `didSet` observer is called after a property has been changed. Their syntax is similar to getters and setters:

```
struct Level {  
    static var highestLevel = 1  
    let id: Int  
    var boss: String  
    var unlocked: Bool {  
        didSet {  
            if unlocked && id > Level.highestLevel {  
                Level.highestLevel = id  
            }  
        }  
    }  
}
```

Now, when the player unlocks a new level, it will update the `highestLevel` type property if the level is a new high. There are a couple of things to note here:

- You *can* access the value of `unlocked` from inside the `didSet` observer. Remember that `didSet` gets called *after* the value has been set.
- Even though you're inside an instance of the type, you still have to access type properties with the type name prefix. You are required to use the full name `Level.highestLevel` rather than just `highestLevel` alone to indicate you're accessing a type property.

`willSet` and `didSet` observers are only available for stored properties. If you want to listen for changes to a computed property, simply add the relevant code to the property's setter.

Also, keep in mind that the `willSet` and `didSet` observers are *not* called when a property is set during initialization; they only get called when you assign a new value to a fully-initialized instance. That means property observers are only useful for variable properties, since constant properties are only set during initialization.

Limiting a variable

You can also use property observers to limit the value of a variable. Say you had a light bulb that could only support a maximum current flowing through its filament.

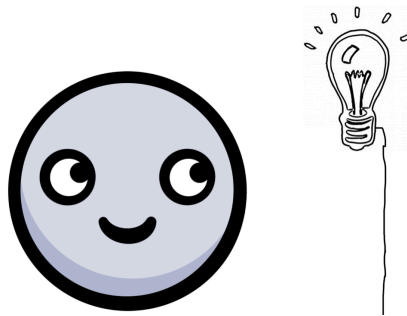
```
struct LightBulb {  
    static let maxCurrent = 40  
    var current = 0 {  
        didSet {  
            if current > LightBulb.maxCurrent {  
                print("Current too high, falling back to previous setting.")  
                current = oldValue  
            }  
        }  
    }  
}
```

In this example, if the current flowing into the bulb exceeds the maximum value, it will revert to its last successful value. Notice there's a helpful `oldValue` constant available in `didSet` so you can access the previous value.

Give it a try:

```
var light = LightBulb()  
light.current = 50  
var current = light.current // 0  
light.current = 40  
current = light.current // 40
```

You try to set the light bulb to 50 amps, but the bulb rejected that input. Pretty cool!



Note: Do not confuse property observers with getters and setters. A stored property can have a `didSet` and/or a `willSet` observer. A computed property has a getter and optionally a setter. These, even though the syntax is similar, are completely different concepts!

Mini-exercise

In the light bulb example, the bulb goes back to a successful setting if the current gets too high. In real life, that wouldn't work. The bulb would burn out! Rewrite the structure so that the bulb turns off before the current burns it out. You'll need to use the `willSet` observer for this. This observer gets called *before* the value is changed. The value that is about to be set is available in the constant `newValue`. You can't change this `newValue`, and it will still be set, so you'll have to do more than just add a `willSet` observer. :]

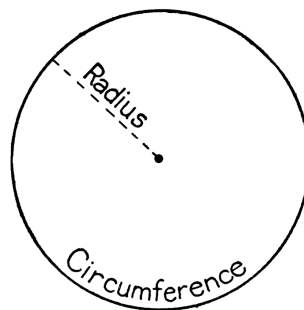
Lazy properties

If you have a property that might take some time to calculate, you don't want to slow things down until you actually need the property. Say hello to the **lazy stored property**. This could be useful for such things as downloading a user's profile picture or making a serious calculation.

Look at this example of a `Circle` structure that uses `pi` in its circumference calculation:

```
struct Circle {  
    lazy var pi = {  
        return ((4.0 * atan(1.0 / 5.0)) - atan(1.0 / 239.0)) * 4.0  
    }()  
    var radius = 0.0  
    var circumference: Double {  
        mutating get {  
            return pi * radius * 2  
        }  
    }  
    init(radius: Double) {  
        self.radius = radius  
    }  
}
```

Here, you're not trusting the value of `pi` available to you from the standard library; you want to calculate it yourself.



You can create a new `Circle` with its initializer, and the `pi` calculation won't run yet:

```
var circle = Circle(radius: 5) // got a circle, pi has not been run
```

The calculation of `pi` waits patiently until you need it. Only when you ask for the `circumference` property is `pi` calculated and assigned a value.

```
let circumference = circle.circumference // 31.42  
// also, pi now has a value
```

Since you've got eagle eyes, you've noticed that `pi` uses a `{ }()` pattern to calculate its value, even though it's a stored property. The trailing parentheses execute the code inside the closure curly braces immediately. But since `pi` is marked as `lazy`, this calculation is postponed until the first time you access the property.

For comparison, `circumference` is a computed property and therefore is calculated every time it's accessed. You expect the `circumference`'s value to change if the `radius` changes. `pi`, as a lazy stored property, is only calculated the first time. That's great, because who wants to calculate the same thing over and over again?

The lazy property must be a variable, defined with `var`, instead of a constant defined with `let`. When you first initialize the structure, the property effectively has no value. Then when some part of your code requests the property, its value will be calculated. So even though the value only changes once, you still use `var`.

Here are two more advanced features of the code:

- Since the value of `pi` changes, the `circumference` getter must be marked as `mutating`. This changes the value of the structure.
- Since `pi` is a stored property of the structure, you need a custom initializer to use only the `radius`. Remember the automatic initializer of a structure includes all of the stored properties.

Don't worry about those advanced features too much for now. You'll learn more about both the `mutating` keyword and custom initializers in the next chapter. The important part to wrap your mind around is the how the lazy stored property works. The rest of the details are window dressing which you'll get more comfortable with in time.

Mini-exercises

Of course, you should absolutely trust the value of `pi` from the standard library. It's a type property, and you can access it as `Double.pi`. Given the `Circle` example above:

1. Remove the lazy stored property `pi`. Use the value of `pi` from the Swift standard library instead.
2. Remove the initializer. Since you only have one stored property, `radius`, you can rely on the automatically included initializer.

Key points

- **Properties** are variables and constants that are part of a named type.
- **Stored properties** allocate memory to store a value.
- **Computed properties** are calculated each time your code requests them and aren't stored as a value in memory.
- The **static** keyword marks a **type property** that's universal to all instances of a particular type.
- The **lazy** keyword prevents a value of a stored property from being calculated until your code uses it for the first time. You'll want to use **lazy initialization** when a property's initial value is computationally intensive or when you won't know the initial value of a property until after you've initialized the object.

Where to go from here?

You saw the basics of properties while learning about structures, and now you've seen the more advanced features they have to offer. You've already learned a bit about methods in the previous chapter and will learn even more about them in the next one!

Challenges

Challenge 1

Rewrite the IceCream structure below to use default values and lazy initialization:

```
struct IceCream {  
    let name: String  
    let ingredients: [String]  
}
```

1. Use default values for the properties.
2. Lazily initialize the ingredients array.

Challenge 2

At the beginning of the chapter, you saw a Car structure. Dive into the inner workings of the car and rewrite the FuelTank structure below with property observer functionality:

```
struct FuelTank {  
    var level: Double // decimal percentage between 0 and 1  
}
```

1. Add a lowFuel stored property of Boolean type to the structure.
2. Flip the lowFuel Boolean when the level drops below 10%.
3. Ensure that when the tank fills back up, the lowFuel warning will turn off.
4. Set the level to a minimum of 0 or a maximum of 1 if it gets set above or below the expected values.
5. Add a FuelTank property to Car.

Chapter 19: Methods

By Ben Morrow

In the previous chapter, you learned about properties, which are constants and variables that are part of structures. **Methods**, as you’ve already seen, are merely functions that reside inside a structure.

In this chapter, you’ll take a closer look at methods and initializers. As with properties, you’ll begin to design more complex structures. The things you learn in this chapter will apply to methods across all named types, including classes and enumerations, which you’ll see in later chapters.

Method refresher

Remember `Array.removeLast()`? It pops the last item off an instance of an array:

```
var numbers = [1, 2, 3]
numbers.removeLast()
numbers // [1, 2]
```



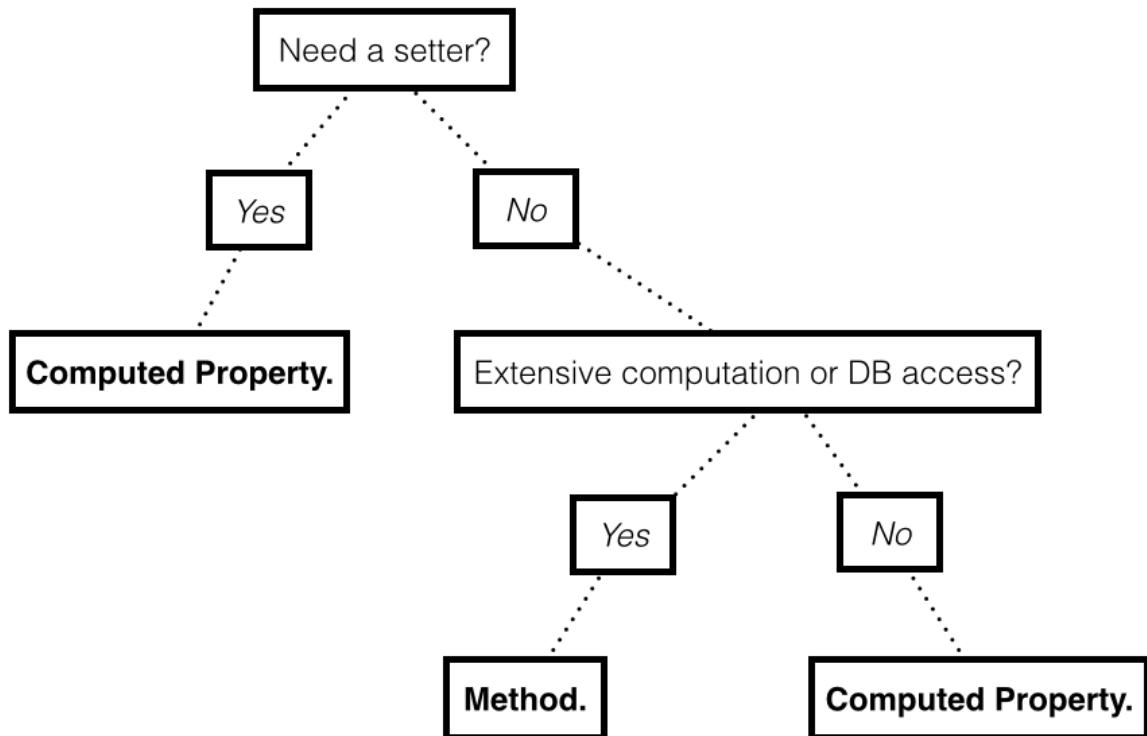
Methods like `removeLast()` help you control the data in the structure.

Comparing methods to computed properties

With computed properties, you saw in the last chapter that you could run code from inside a structure. That sounds a lot like a method. What’s the difference? It really comes down to a matter of style, but there are a few helpful thoughts to help you decide. Properties hold values that you can get and set, while methods perform work.

Sometimes this distinction gets fuzzy when a method's sole purpose is to return a single value.

Should I implement this value getter as a method or as a computed property?



Ask yourself whether you want to be able to set a value as well as get the value. A computed property can have a setter component inside to write values. Another question to consider is whether the calculation requires extensive computation or reads from a database. Even for a simple value, a method helps you indicate to future developers that the call is expensive in time and computational resources. If the call is cheap (as in constant time $O(1)$), stick with a computed property.

Turning a function into a method

To explore methods and initializers, you will create a simple model for dates called `SimpleDate`. Be aware that Apple's Foundation library contains a robust, production ready `Date` class that correctly handles all of the subtle intricacies of dealing with dates and times.

In the code below, how could you convert `monthsUntilWinterBreak(date:)` into a method?

```
let months = ["January", "February", "March",
              "April", "May", "June",
              "July", "August", "September",
              "October", "November", "December"]

struct SimpleDate {
    var month: String
}

func monthsUntilWinterBreak(from date: SimpleDate) -> Int {
    return months.index(of: "December")! - months.index(of: date.month)!
}
```

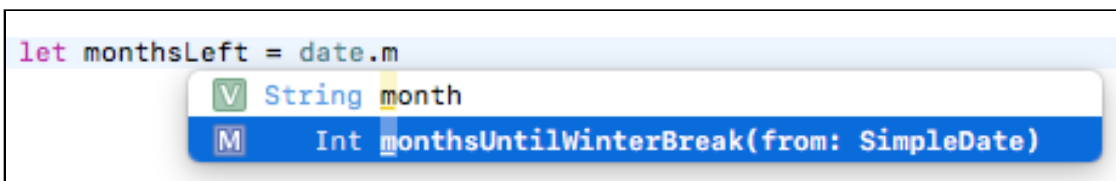
Note: If you live in the southern hemisphere, you can adjust the month for your winter break. :]

Making a method is as easy as moving the function inside the structure definition:

```
struct SimpleDate {
    var month: String

    func monthsUntilWinterBreak(from date: SimpleDate) -> Int {
        return months.index(of: "December")! - months.index(of: date.month)!
    }
}
```

There's no identifying keyword for a method; it really is just a function inside a named type. You call methods on an instance using dot syntax just as you do for properties. And just like properties, as soon as you start typing a method name, Xcode will provide suggestions. You can select one with the Up and Down arrow keys on your keyboard, and you can autocomplete the call by pressing Tab:



```
let date = SimpleDate(month: "October")
date.monthsUntilWinterBreak(from: date) // 2
```

If you think about this code for a minute, you'll realize that the method's definition is awkward. There must be a way to access the content stored by the instance instead of passing the instance itself as a parameter to the method. It would be so much nicer to call this:

```
date.monthsUntilWinterBreak() // Error!
```

Introducing self

A structure definition is like a blueprint, whereas an instance is a real object. To access the value of an instance, you use the keyword **self** inside the structure. The Swift compiler passes it in to your method as secret parameter. The method definition transforms into this:

```
// 1
func monthsUntilWinterBreak() -> Int {
    // 2
    return months.index(of: "December")! - months.index(of: self.month)!
}
```

Here's what changed:

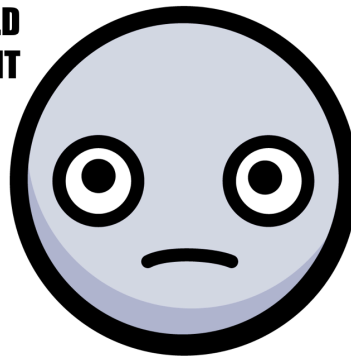
1. Now there's no parameter in the method definition.
2. In the implementation, `self` replaces the old parameter name.

You can now call the method without passing a parameter:

```
date.monthsUntilWinterBreak() // 2
```

That's looking a lot cleaner! One more thing you can do to simplify the code is to remove `self`.

**...BUT YOU JUST TOLD
ME TO ADD IT**



`self` is your reference to the instance, but most of the time you don't need to use it because Swift understands your intent if you just use a variable name. While you can always use `self` to access the properties and methods of the current instance, most of the time you don't need to. In `monthsUntilWinterBreak()`, you can just say `month` instead of `self.month`:

```
return months.index(of: "December")! - months.index(of: month)!
```

Most programmers use `self` only when it is required, for example, to disambiguate between a local variable and a property with the same name. You'll get more practice using `self` a little later.

Mini-exercise

Since `monthsUntilWinterBreak()` returns a single value and there's not much calculation involved, transform the method into a computed property with a getter component.

Introducing initializers

You learned about initializers in the previous chapters, but let's look at them again with your newfound knowledge of methods.

Initializers are special methods you call to create a new instance. They omit the `func` keyword and even a name. Instead, they use `init`. An initializer can have parameters, but it doesn't have to.

Right now, when you create a new instance of the `SimpleDate` structure, you have to specify a value for the `month` property:

```
let date = SimpleDate(month: "January")
```

You might find it more efficient to have a handy no-parameter initializer. This would create a new `SimpleDate` instance with a reasonable default value:

```
let date = SimpleDate() // Error!
```

While the compiler gives you an error now, you can provide the no-parameter initializer. By implementing `init`, you can create the simplest path to initialization with default values:

```
struct SimpleDate {  
    var month: String  
  
    init() {  
        month = "January"  
    }  
  
    func monthsUntilWinterBreak() -> Int {  
        return months.index(of: "December")! - months.index(of: month)!  
    }  
}
```

Here's what's happening in that code:

1. The `init()` definition requires neither the `func` keyword nor a name. You always use the name of the type to call an initializer.
2. Like a function, an initializer must have a parameter list, even if it is empty.
3. In the initializer, you assign values for all the stored properties of a structure.
4. An initializer never returns a value. Its task is simply to initialize a new instance.

Now you can use your simple initializer to create an instance:

```
let date = SimpleDate()  
date.month // January  
date.monthsUntilWinterBreak() // 11
```

You can test a change to the value in the initializer:

```
init() {  
    month = "March"  
}
```

The value of `monthsUntilWinterBreak()` will change accordingly:

```
let date = SimpleDate()  
date.month // March  
date.monthsUntilWinterBreak() // 9
```

As you think about the implementation here, a good user experience optimization would have the initializer use a default value based on today's date.

In the future, you'll be capable of retrieving the current date. Eventually you'll use the `Date` class from the Foundation library to work with dates. Before you get carried away with all the power that these libraries provide, let's continue implementing your own `SimpleDate` type from the ground, up.

Initializers in structures

Initializers ensure all properties are set before the instance is ready to use:

```
struct SimpleDate {  
    var month: String  
    var day: Int  
  
    init() {  
        month = "January"  
        day = 1  
    }  
  
    func monthsUntilWinterBreak() -> Int {  
        return months.index(of: "December")! - months.index(of: month)!  
    }  
}
```

If you tried to create an initializer without setting the day property, then the compiler would complain.

By creating even one custom initializer, you forgo the option to use the automatic **memberwise initializer**. Recall that the auto-generated memberwise initializer accepts all the properties in order as parameters, such as `init(month:day:)`, for the `SimpleDate` structure. When you write a custom initializer, the compiler scraps the one created automatically.

So this code won't work right now:

```
let valentinesDay = SimpleDate(month: "February", day: 14) // Error!
```

Instead, you'll have to define your own initializer with parameters:

```
init(month: String, day: Int) {  
    self.month = month  
    self.day = day  
}
```

In that code, you assign the incoming parameters to the properties of the structure. Notice how `self` is used to tell the compiler that you're referring to the property rather than the local parameter.

This wasn't necessary in the simple initializer:

```
init() {  
    month = "January"  
    day = 1  
}
```

In that code, there aren't any parameters with the same names as the properties. Therefore, `self` isn't necessary for the compiler to understand you're referring to properties.

With the complex initializer in place, you can call the new initializer the same way you used to call the automatically generated initializer:

```
let valentinesDay = SimpleDate(month: "February", day: 14)
valentinesDay.month // February
valentinesDay.day // 14
```

Introducing mutating methods

Methods in structures cannot change the values of the instance without being marked as mutating. You can imagine a method in the `SimpleDate` structure that advances to the next day:

```
mutating func advance() {
    day += 1
}
```

Note: The implementation above is a naive way of writing `advance()` because it doesn't account for what happens at the end of a month. In a challenge at the end of this chapter, you'll create a more robust version.

The `mutating` keyword marks a method that changes a structure's value. Since a structure is a value type, the system copies it each time it's passed around an app. If a method changes the value of one of the properties, then the original instance and the copied instance will no longer be equivalent.

By marking a method as `mutating`, you're also telling the Swift compiler this method must not be called on constants. This is how Swift knows which methods to allow and which to reject at compile time. If you call a mutating method on a constant instance of a structure, the compiler will flag it as an error that must be corrected before you can run your program.

For mutating methods, Swift secretly passes in `self` just like it did for normal methods. But for mutating methods, the secret `self` gets marked as an `inout` parameter.

Type methods

Like type properties, you can use **type methods** to access data across all instances. You call type methods on the type itself, instead of on an instance. To define a type method, you prefix it with the `static` modifier.

Type methods are useful for things that are *about* a type in general, rather than something about specific instances.

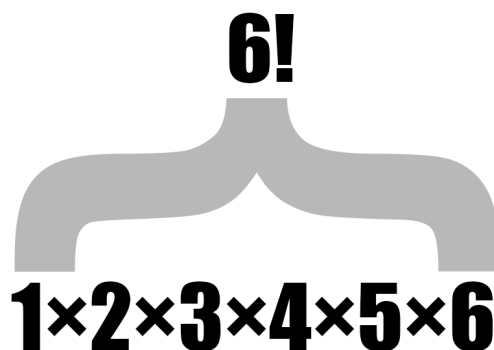
For example, you could use type methods to group similar methods into a structure:

```
struct Math {  
    // 1  
    static func factorial(of number: Int) -> Int {  
        // 2  
        return (1...number).reduce(1, *)  
    }  
}  
// 3  
Math.factorial(of: 6) // 720
```

You might have custom calculations for things such as factorial. Instead of having a bunch of free-standing functions, you can group related functions together as type methods in a structure. The structure is said to act as a **namespace**.

Here's what's happening:

1. You use `static` to declare the type method, which accepts an integer and returns an integer.
2. The implementation uses a higher-order function called `reduce(_:_:)`. It effectively follows the formula for calculating a factorial: “The product of all the whole numbers from 1 to n”. You could write this using a `for` loop, but the higher-order function expresses your intent in a cleaner way.
3. You call the type method on `Math`, rather than on an instance of the type.


$$6! \\ 1 \times 2 \times 3 \times 4 \times 5 \times 6$$

Type methods gathered into a structure will advantageously code complete in Xcode. In this example, you can see all the math utility methods available to you by typing `Math..`

A screenshot of the Xcode editor showing a code completion dropdown menu. The text 'let factorial = Math.' is entered in the code area. The dropdown menu is open, showing two suggestions: 'Int factorial(of: Int)' and 'Math init()'. Each suggestion is preceded by a small icon with the letter 'M'.

Mini-exercise

Add a type method to the `Math` structure that calculates the n -th triangle number. It will be very similar to the factorial formula, except instead of multiplying the numbers, you add them.

Adding to an existing structure with extensions

Sometimes you want to add functionality to a structure but don't want to muddy up the original definition. And sometimes you can't add the functionality because you don't have access to the source code. It is possible to *open* an existing structure (even one you do not have the source code for) and add methods, initializers and computed properties to it. This can be useful for code organization and is discussed in greater detail in Chapter 18. Doing so is as easy as using the keyword `extension`.

At the bottom of your playground, outside the definition of `Math`, add this type method named `primeFactors(of:)` using an extension:

```
extension Math {
    static func primeFactors(of value: Int) -> [Int] {
        // 1
        var remainingValue = value
        // 2
        var testFactor = 2
        var primes: [Int] = []
        // 3
        while testFactor * testFactor <= remainingValue {
            if remainingValue % testFactor == 0 {
                primes.append(testFactor)
                remainingValue /= testFactor
            }
            else {
                testFactor += 1
            }
        }
    }
}
```

```
        if remainingValue > 1 {
            primes.append(remainingValue)
        }
        return primes
    }
}
```

This method finds the prime factors for a given number. For example, 81 returns [3, 3, 3, 3]. Here's what's happening in the code:

1. The value passed in as a parameter is assigned to the mutable variable, `remainingValue` so that it can be changed as the calculation runs.
2. The `testFactor` starts as two and will be divided into `remainingValue`.
3. The logic runs a loop until the `remainingValue` is exhausted. If it evenly divides, meaning there's no remainder, that value of the `testFactor` is set aside as a prime factor. If it doesn't evenly divide, `testFactor` is incremented for the next loop.

This algorithm is brute force, but does contain one optimization: the square of the `testFactor` should never be larger than the `remainingValue`. If it is, the `remainingValue` itself must be prime and it is added to the `primes` list.

You've now added a method to `Math` without changing its original definition. Verify that the extension works with this code:

```
Math.primeFactors(of: 81) // [3, 3, 3, 3]
```

Pretty slick! You're about to see how that can be powerful in practice.

Note: In an extension, you cannot add stored properties to an existing structure because that would change the size and memory layout of the structure and break existing code.

Keeping the compiler generated initializer using extensions

With your `SimpleDate` structure, you saw that as soon as you added your own `init()` the compiler generated memberwise initializer disappeared. It turns out that you can keep both if you add your own `init()` as an extension to `SimpleDate`:

```
struct SimpleDate {
    var month: String
    var day: Int

    func monthsUntilWinterBreak() -> Int {
```

```
    }  
    return months.index(of: "December")! - months.index(of: month)!  
  }  
  
  mutating func advance() {  
    day += 1  
  }  
}  
  
extension SimpleDate {  
  init() {  
    month = "January"  
    day = 1  
  }  
}
```

`init()` gets added to `SimpleDate` without sacrificing the automatically generated memberwise initializer. Hooray!

Key points

- **Methods** are behaviors that extend the functionality of a type.
- A method is a function defined inside of a named type.
- A method can access the value of an instance by using the keyword `self`.
- **Initializers** are methods that aid in the creation of a new instance.
- A **type method** adds behavior to a type instead of the instances of that type. To define a type method, you prefix it with the `static` modifier.
- You can open an existing structure and add methods, initializers and computed properties to it using an extension.
- By adding your own initializers in extensions, you can keep the compiler's initializer for a structure.

Where to go from here?

Methods and properties are the things that make up your types. Learning about them as you have these two chapters is important since you'll find them in all the named types — structures, classes, and enumerations.

You've tackled the basics with structures. Now you're ready to learn about the the next named type, **classes**, in the next chapter.

Challenges

1. Given the Circle structure below:

```
struct Circle {  
    var radius = 0.0  
  
    var area: Double {  
        return .pi * radius * radius  
    }  
  
    init(radius: Double) {  
        self.radius = radius  
    }  
}
```

Write a method that can change an instance's area by a growth factor. For example if you call `circle.grow(byFactor: 3)`, the area of the instance will triple.

Hint: Add a setter to area.

- Below is a naïve way of writing `advance()` for the `SimpleDate` structure you saw earlier in the chapter:

```
let months = ["January", "February", "March",
              "April", "May", "June",
              "July", "August", "September",
              "October", "November", "December"]

struct SimpleDate {
    var month: String
    var day: Int

    mutating func advance() {
        day += 1
    }
}

var date = SimpleDate(month: "December", day: 31)
date.advance()
date.month // December; should be January!
date.day // 32; should be 1!
```

What happens when the function should go from the end of one month to the start of the next? Rewrite `advance()` to account for advancing from December 31st to January 1st.

- Add type methods named `isEven` and `isOdd` to your `Math` namespace that return `true` if a number is even or odd respectively.
- It turns out that `Int` is simply a struct. Add the computed properties `isEven` and `isOdd` to `Int` using an extension.

Note: Generally, you want to be careful about what functionality you add to standard library types as it can cause confusion for readers.

- Add the method `primeFactors()` to `Int`. Since this is an expensive operation, this is best left as an actual method.

Chapter 20: Classes

By Cosmin Pupăză

Structures introduced you to named types. In this chapter, you'll get acquainted with **classes**, which are much like structures — they are named types with properties and methods.

You'll learn classes are *reference* types, as opposed to *value* types, and have substantially different capabilities and benefits than their structure counterparts. While you'll often use structures in your apps to represent values, you'll generally use classes to represent *objects*.

What does *values* vs *objects* really mean, though?

Creating classes

Consider the following class definition in Swift:

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

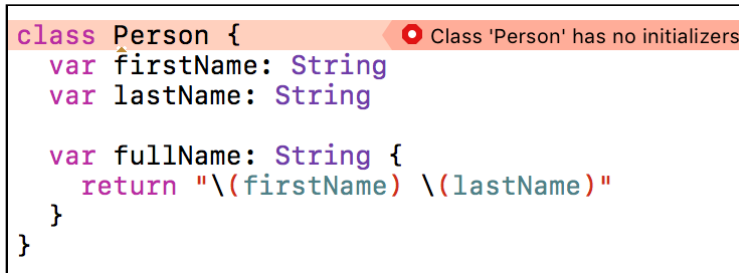
    var fullName: String {
        return "\(firstName) \(lastName)"
    }
}

let john = Person(firstName: "Johnny", lastName: "Appleseed")
```

That's simple enough! It may surprise you that the definition is almost identical to its struct counterpart. The keyword `class` is followed by the name of the class, and everything in the curly braces is a member of that class.

But you can also see some differences between a class and a struct: The class above defines an initializer that sets both `firstName` and `lastName` to initial values. Unlike a struct, a class doesn't provide a memberwise initializer automatically — which means you must provide it yourself if you need it.

If you forget to provide an initializer, the Swift compiler will flag that as an error:



```
class Person {  
    var firstName: String  
    var lastName: String  
  
    var fullName: String {  
        return "\(firstName) \(lastName)"  
    }  
}
```

Default initialization aside, the initialization rules for classes and structs are very similar. Class initializers are functions marked `init`, and all stored properties must be assigned initial values before the end of `init`.

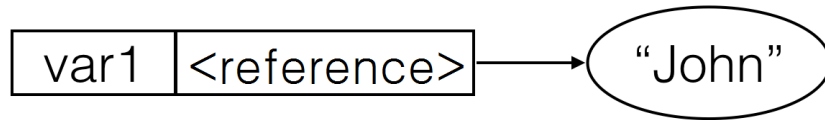
There is actually *much* more to class initialization than that, but you'll have to wait until the next chapter, "Advanced Classes", which will introduce you to the concept of **inheritance** and the effect it has on initialization rules. For now, you'll get comfortable with classes in Swift by working with basic class initializers.

Reference types

In Swift, an instance of a structure is an immutable value. An instance of a class, on the other hand, is a mutable object. Because classes are reference types, a variable of a class type does not store an actual instance, but a **reference** to a location in memory that stores the instance. If you were to create a `SimplePerson` class instance with only a name like this:

```
class SimplePerson {  
    let name: String  
    init(name: String) {  
        self.name = name  
    }  
}  
  
var var1 = SimplePerson(name: "John")
```

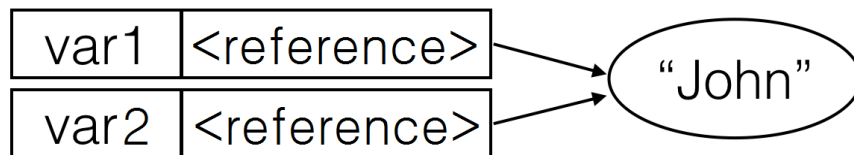

It would look something like this in memory:



If you were to create a new variable `var2` and assign to it the value of `var1`:

```
var var2 = var1
```

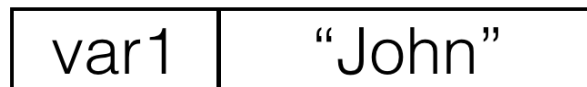
Then the references inside both `var1` and `var2` would reference the same place in memory:



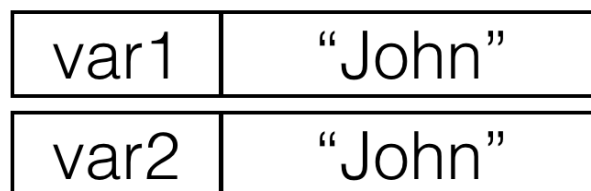
Conversely, a structure as a value type stores the actual value, providing direct access to it. Replace the `SimplePerson` class implementation with a struct like this:

```
struct SimplePerson {
    let name: String
}
```

In memory, the variable would not reference a place in memory but the value would instead belong to `var1` exclusively:



The assignment `var var2 = var1` would **copy** the *value* of `var1` in this case:



Value types and reference types each have their own distinct advantages — and disadvantages. Later in the chapter, you’ll consider the question of which type to use in a given situation. For now, let’s examine how classes and structs work under the hood. While the below description doesn’t apply to every situation (the Swift optimizer can be smart), it is a good first-order model to keep in mind.

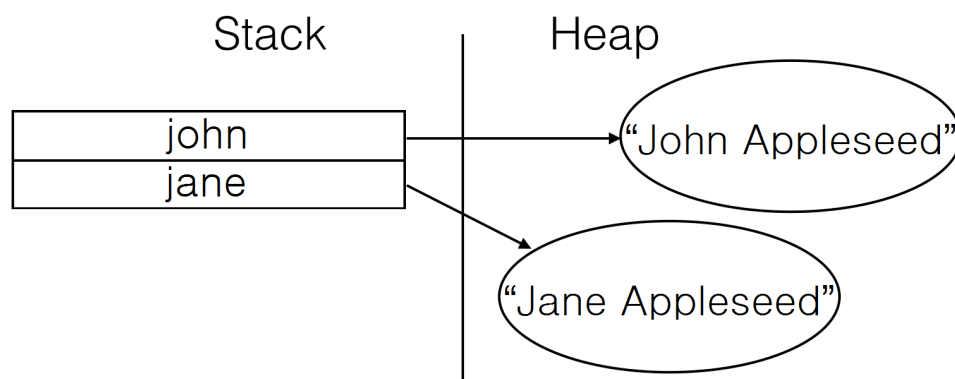
The heap vs. the stack

When you create a reference type such as class, the system stores the actual instance in a region of memory known as the **heap**. Instances of a value type such as a struct resides in a region of memory called the **stack**, unless the value is part of a class instance, in which case the value is stored on the heap with the rest of the class instance.

Both the heap and the stack have essential roles in the execution of any program. A general understanding of what they are and how they work will help you visualize the functional differences between a class and a structure:

- The system uses the **stack** to store anything on the immediate thread of execution; it is tightly managed and optimized by the CPU. When a function creates a variable, the stack stores that variable and then destroys it when the function exits. Since the stack is so well organized, it’s very efficient, and thus quite fast.
- The system uses the **heap** to store instances of reference types. The heap is generally a large pool of memory from which the system can request and dynamically allocate blocks of memory. Lifetime is flexible and dynamic. The heap doesn’t automatically destroy its data like the stack does; additional work is required to do that. This makes creating and removing data on the heap a slower process, compared to on the stack.

Maybe you’ve already figured out how this relates to structs and classes. Take a look at the diagram below:



- When you create an instance of a class, your code requests a block of memory on the heap to store the instance itself; that's the first name and last name inside the instance on the right side of the diagram. It stores the *address* of that memory in your named variable on the stack; that's the *reference* stored on the left side of the diagram.
- When you create an instance of a struct (that is not part of an instance of a class), the instance itself is stored on the stack, and the heap is never involved.

This has only been a brief introduction to the dynamics of heaps and stacks, but you know enough at this point to understand the reference semantics you'll use to work with classes.

Working with references

In Chapter 10, "Structures", you saw the copy semantics involved when working with structures and other value types. Here's a little reminder, using the `Location` and `DeliveryArea` structures from that chapter:

```
struct Location {
    let x: Int
    let y: Int
}

struct DeliveryArea {
    var range: Double
    let center: Location
}

var area1 = DeliveryArea(range: 2.5,
                        center: Location(x: 2, y: 4))
var area2 = area1
print(area1.range) // 2.5
print(area2.range) // 2.5

area1.range = 4
print(area1.range) // 4.0
print(area2.range) // 2.5
```

When you assign the value of `area1` into `area2`, `area2` receives a *copy* of the `area1` value. That way when `area1.range` receives a new value of 4, the number is only reflected in `area1` while `area2` still has the original value of 2.5.

Since a class is a reference type, when you assign to a variable of a class type, the system does *not* copy the instance; only a reference is copied.

Contrast the previous code with the following code:

```
var homeOwner = john
john.firstName = "John" // John wants to use his short name!
john.firstName // "John"
homeOwner.firstName // "John"
```

As you can see, `john` and `homeOwner` truly have the same value!

This implied sharing among class instances results in a new way of thinking when passing things around. For instance, if the `john` object changes, then anything holding a reference to `john` will automatically see the update. If you were using a structure, you would have to update each copy individually, or it would still have the old value of “Johnny”.

Mini-exercise

Change the value of `lastName` on `homeOwner`, then try reading `fullName` on both `john` and `homeOwner`. What do you observe?

Object identity

In the previous code sample, it’s easy to see that `john` and `homeOwner` are pointing to the same object. The code is short and both references are named variables. What if you want to see if the value behind a variable *is* John?

You might think to check the value of `firstName`, but how would you know it’s the John you’re looking for and not an imposter? Or worse, what if John changed his name again?

In Swift, the `===` operator lets you check if the *identity* of one object is equal to the identity of another:

```
john === homeOwner // true
```

Just as the `==` operator checks if two *values* are equal, the `===` identity operator compares the memory address of two *references*. It tells you whether the value of the references are the same; that is, they point to the same block of data on the heap.

That means this `===` operator can tell the difference between the John you’re looking for and an imposter-John.

```
let imposterJohn = Person(firstName: "Johnny", lastName: "Appleseed")

john === homeOwner // true
john === imposterJohn // false
imposterJohn === homeOwner // false
```

```
// Assignment of existing variables changes the instances the variables
reference.
homeOwner = imposterJohn
john === homeOwner // false

homeOwner = john
john === homeOwner // true
```

This can be particularly useful when you cannot rely on regular equality (==) to compare and identify objects you care about:

```
// Create fake, imposter Johns. Use === to see if any of these imposters
are our real John.
var imposters = (0...100).map { _ in
    Person(firstName: "John", lastName: "Appleseed")
}

// Equality (==) is not effective when John cannot be identified by his
name alone
imposters.contains {
    $0.firstName == john.firstName && $0.lastName == john.lastName
} // true
```

By using the identity operator, you can verify that the *references* themselves are equal, and separate our real John from the crowd:

```
// Check to ensure the real John is not found among the imposters.
imposters.contains {
    $0 === john
} // false

// Now hide the "real" John somewhere among the imposters.
imposters.insert(john, at: Int(arc4random_uniform(100)))

// John can now be found among the imposters.
imposters.contains {
    $0 === john
} // true

// Since `Person` is a reference type, you can use === to grab the real
John out of the list of imposters and modify the value.
// The original `john` variable will print the new last name!
if let indexOfJohn = imposters.index(where: { $0 === john }) {
    imposters[indexOfJohn].lastName = "Bananapeel"
}

john.fullName // John Bananapeel
```

Note: You have to import the Foundation framework in order to work with the `arc4random_uniform(_:)` function.

You may actually find that you won't use the identity operator `===` very much in your day-to-day Swift. What's important is to understand what it does, and what it demonstrates about the properties of reference types.

Mini-exercise

Write a function `memberOf(person: Person, group: [Person]) -> Bool` that will return `true` if `person` can be found inside `group`, and `false` if it can not.

Test it by creating two arrays of five `Person` objects for `group` and using `john` as the person. Put `john` in one of the arrays, but not in the other.

Methods and mutability

As you've read before, instances of classes are mutable objects whereas instances of structures are immutable values. The following example illustrates this difference:

```
struct Grade {
    let letter: String
    let points: Double
    let credits: Double
}

class Student {
    var firstName: String
    var lastName: String
    var grades: [Grade] = []

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    func recordGrade(_ grade: Grade) {
        grades.append(grade)
    }
}

let jane = Student(firstName: "Jane", lastName: "Appleseed")
let history = Grade(letter: "B", points: 9.0, credits: 3.0)
var math = Grade(letter: "A", points: 16.0, credits: 4.0)

jane.recordGrade(history)
jane.recordGrade(math)
```

Note that `recordGrade(_:)` can mutate the array `grades` by adding more values to the end. Even though this method mutates the current object, the keyword `mutating` is not required.

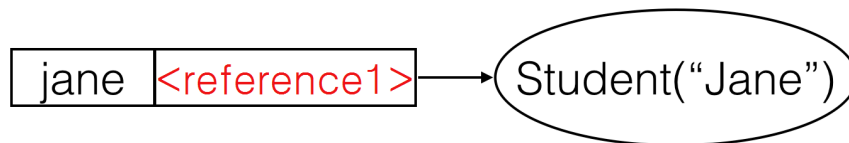
If you had tried this with a struct, you'd have wound up with a build failure, because structures are immutable. Remember, when you change the value of a struct, instead of

modifying the value, you're making a *new* value. The keyword *mutating* marks methods that replace the current value with a new one. With classes, this keyword is not used because the instance itself is mutable.

Mutability and constants

The previous example may have had you wondering how you were able to modify *jane* even though it was defined as a constant.

When you define a constant, the value of the constant cannot be changed. If you recall back to the discussion of value types vs reference types, it is important to remember that, with reference types, the value is a *reference*.



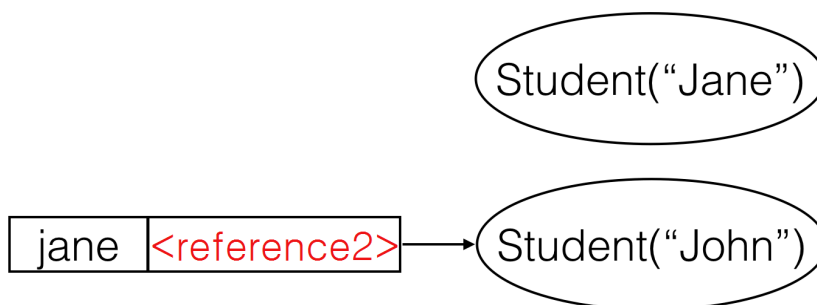
The value of "reference1" in red is the value stored in *jane*. This value is a reference and because *jane* is declared as a constant, this reference is constant. If you were to attempt to assign another student to *jane*, you would get a build error:

```
// Error: jane is a `let` constant  
jane = Student(firstName: "John", lastName: "Appleseed")
```

If you declared *jane* as a variable instead, you would be able to assign to it another instance of *Student* on the heap:

```
var jane = Student(firstName: "Jane", lastName: "Appleseed")  
jane = Student(firstName: "John", lastName: "Appleseed")
```

After the assignment of another *Student* to *jane*, the reference value behind *jane* would be updated to point to the new *Student* object.



Since nothing would be referencing the original “Jane” object, its memory would be freed to use elsewhere. You’ll learn more about this in Chapter 23, “Asynchronous Closures and Memory Management”.

Any individual member of a class can be protected from modification through the use of constants, but because reference types are not *themselves* treated as values, they are not protected as a whole from mutation.

Mini-exercise

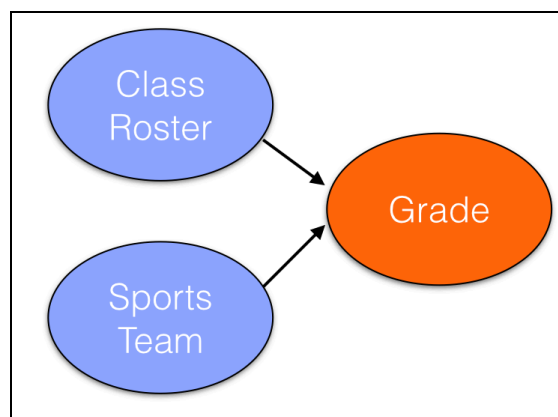
Add a computed property to `Student` that returns the student’s Grade Point Average, or GPA. A GPA is defined as the number of points earned divided by the number of credits taken. For the example above, Jane earned ($9 + 16 = 25$) points while taking ($3 + 4 = 7$) credits, making her GPA ($25 / 7 = 3.57$).

Note: Points in most American universities range from 4 per credit for an A, down to 1 point for a D (with an F being 0 points). For this exercise, you may of course use any scale that you want!

Understanding state and side effects

The referenced and mutable nature of classes leads to numerous programming possibilities, as well as many concerns. If you update a class instance with a new value, then every reference to that instance will also see the new value.

You can use this to your advantage. Perhaps you pass a `Student` instance to a sports team, a report card and a class roster. Imagine all of these entities need to know the student’s grades, and because they all point to the same instance, they’ll all see new grades as the instance records them.



The result of this sharing is that class instances have **state**. Changes in state can sometimes be obvious, but often they're not.

To illustrate this, add a `credits` property to the `Student` class:

```
var credits = 0.0
```

and update `recordGrade(_:)` to use this new property:

```
func recordGrade(_ grade: Grade) {  
    grades.append(grade)  
    credits += grade.credits  
}
```

In this slightly modified example of `Student`, `recordGrade(_:)` now adds the number of credits to the `credits` property. Calling `recordGrade(_:)` has the side effect of updating `credits`.

Now, observe how side effects can result in non-obvious behavior:

```
jane.credits // 7  
  
// The teacher made a mistake; math has 5 credits  
math = Grade(letter: "A", points: 20.0, credits: 5.0)  
jane.recordGrade(math)  
  
jane.credits // 12, not 8!
```

Whoever wrote the modified `Student` class did so somewhat naïvely by assuming that the same grade won't get recorded twice! Because class instances are mutable, you need to be careful about unexpected behavior around shared references.

While confusing in a small example such as this, mutability and state could be extremely jarring as classes grow in size and complexity. Situations like this would be much more common with a `Student` class that scales to 20 stored properties and has 10 methods.

Extending a class using an extension

Just as you saw with structs, classes can be *re-opened* using the `extension` keyword to add methods and computed properties. Add a `fullName` computed property to `Student`:

```
extension Student {  
    var fullName: String {  
        return "\(firstName) \(lastName)"  
    }  
}
```

Functionality can also be added to classes using *inheritance*. You can even add new stored properties to inheriting classes. You will explore this technique in detail in the next chapter.

When to use a class versus a struct

Now that you know the differences and similarities between a class and a struct, you may be wondering “How do I know which to use?”

Values vs. objects

While there are no hard and fast rules, one strategy is to think about value versus reference semantics and use structures as *values* and classes as *objects with identity*. An **object** is an instance of a reference type. Such instances have **identity**. This means that every object is unique and no two objects are considered equal just because they hold the same state. This is why you had to use `===` to see if objects are truly equal and not just containing the same state. This is in contrast to instances of value types, which *are* values and are by definition considered equal if they are the same value.

For example: A delivery range is a value and should be implemented as a struct, while a student is an object and should be implemented as a class. No two students should be considered equal just because they have the same name!

Speed

Speed considerations may also come into play, as structs rely on the faster stack while classes rely on the slower heap. If you will have *many* instances of a type created (hundreds to many thousands), or if these instances will only exist in memory for a very short time, then you should generally lean towards using a struct. If your instance will have a longer lifecycle in memory, or if you will create relatively few instances, then creating class instances on the heap generally won't create much overhead.

For instance, you may want to use a struct to calculate the total distance of a running route using many GPS-based waypoints, like the `Location` struct you used in Chapter 10, “Structures”. Not only will you create many waypoints, but they will also be created and destroyed quickly as you modify the route.

Conversely, you could use a class for an object to store route history as you'll only have one object for each user, and you would likely use the same history object for the lifetime of the user.

Minimalist approach

Another approach is to use only what you need. If your data will never change or you need a simple data store, then use structures. If you need to update your data and you need it to contain logic to update its own state, then use classes. Often, it's best to begin with a struct. If later you need the added capabilities of a class, then convert the struct to a class.

Structures vs. classes recap

Structures

- Useful for representing values
- Implicit copying of values
- Becomes completely immutable when declared with `let`
- Fast memory allocation (stack)

Classes

- Useful for representing objects with identity
- Implicit sharing of objects
- Internals can remain mutable even when declared with `let`
- Slower memory allocation (heap)

Key points

- Like structures, **classes** are a named type that can have properties and methods.
- Classes use **references** that are shared on assignment.
- Class instances are called **objects**.
- Objects are **mutable**.
- Mutability introduces **state**, which adds another level of complexity when managing your objects.
- Use classes when you want **reference semantics**, and structures when you want **value semantics**.

Where to go from here?

Classes should feel familiar to you after having learned about structs. However, there are enough subtle differences around copying, sharing and mutability that it's important to keep them distinct in your head!

When deciding between a struct and a class, you've seen some of the factors to take into consideration — and there will be more things to consider in the very next chapter!

Challenges

Challenge 1: Movie lists — benefits of reference types

Imagine you're writing a movie-viewing application in Swift. Users can create lists of movies and share those lists with other users.

Create a `User` and a `List` class that uses reference semantics to help maintain lists between users.

- `User`: Has a method `addList(_:)` which adds the given list to a dictionary of `List` objects (using the name as a key), and `list(forName:) -> List?` which will return the `List` for the provided name.
- `List`: Contains a name and an array of movie titles. A `print` method will print all the movies in the list.
- Create `jane` and `john` users and have them create and share lists. Have both `jane` and `john` modify the same list and call `print` from both users. Are all the changes reflected?
- What happens when you implement the same with structs? What problems do you run into?

Challenge 2: T-Shirt store — class or struct?

Your challenge here is to build a set of objects to support a T-shirt store. Decide if each object should be a class or a struct, and why.

- `TShirt`: Represents a shirt style you can buy. Each `TShirt` has a size, color, price, and an optional image on the front.
- `User`: A registered user of the t-shirt store app. A user has a name, email, and a `ShoppingCart` (see below).

- **Address:** Represents a shipping address, containing the name, street, city, and zip code.
- **ShoppingCart:** Holds a current order, which is composed of an array of `TShirt` that the User wants to buy, as well as a method to calculate the total cost. Additionally, there is an `Address` that represents where the order will be shipped.

Bonus: After you've decided on whether to use a class or struct for each object, go ahead and implement them all!

Chapter 21: Advanced Classes

By Cosmin Pupăză

The previous chapter introduced you to the basics of defining and using classes in Swift. Classes are reference types and can be used to support traditional object-oriented programming.

Classes introduce inheritance, overriding, polymorphism and composition which makes them suited for this purpose. These extra features require special consideration for initialization, class hierarchies, and understanding the class lifecycle in memory.

This chapter will introduce you to the finer points of classes in Swift, and help you understand how you can create more complex classes.

Introducing inheritance

In the previous chapter, you saw a `Grade` struct and a pair of class examples: `Person` and `Student`.

```
struct Grade {
    var letter: Character
    var points: Double
    var credits: Double
}

class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

```
class Student {  
    var firstName: String  
    var lastName: String  
    var grades: [Grade] = []  
  
    init(firstName: String, lastName: String) {  
        self.firstName = firstName  
        self.lastName = lastName  
    }  
  
    func recordGrade(_ grade: Grade) {  
        grades.append(grade)  
    }  
}
```

It's not difficult to see that there's an incredible amount of redundancy between `Person` and `Student`. Maybe you've also noticed that a *Student is a Person*!

This simple case demonstrates the idea behind class inheritance. Much like in the real world, where you can think of a student as a person, you can represent the same relationship in code by replacing the original `Student` class implementation with the following:

```
class Student: Person {  
    var grades: [Grade] = []  
  
    func recordGrade(_ grade: Grade) {  
        grades.append(grade)  
    }  
}
```

In this modified example, the `Student` class now **inherits** from `Person`, indicated by a colon after the naming of `Student`, followed by the class from which `Student` inherits, which in this case is `Person`.

Through inheritance, `Student` automatically gets the properties and methods declared in the `Person` class. In code, it would be accurate to say that a *Student is-a Person*.

With much less duplication of code, you can now create `Student` objects that have all the properties and methods of a `Person`:

```
let john = Person(firstName: "Johnny", lastName: "Appleseed")  
let jane = Student(firstName: "Jane", lastName: "Appleseed")  
  
john.firstName // "John"  
jane.firstName // "Jane"
```

Additionally, only the Student object will have all of the properties and methods defined in Student:

```
let history = Grade(letter: "B", points: 9.0, credits: 3.0)
jane.recordGrade(history)
// john.recordGrade(history) // john is not a student!
```

A class that inherits from another class is known as a **subclass** or a **derived class**, and the class from which it inherits is known as a **superclass** or a **base class**.

The rules for subclassing are fairly simple:

- A Swift class can inherit from only one other class, a concept known as **single inheritance**.
- There's no limit to the depth of subclassing, meaning you can subclass from a class that is *also* a subclass, like below:

```
class BandMember: Student {
    var minimumPracticeTime = 2
}

class OboePlayer: BandMember {
    // This is an example of an override, which we'll cover soon.
    override var minimumPracticeTime: Int {
        get {
            return super.minimumPracticeTime * 2
        }
        set {
            super.minimumPracticeTime = newValue / 2
        }
    }
}
```

A chain of subclasses is called a **class hierarchy**. In this example, the hierarchy would be OboePlayer -> BandMember -> Student -> Person. A class hierarchy is analogous to a family tree. Because of this analogy, a superclass is also called the **parent class** of its **child class**.

Polymorphism

The Student/Person relationship demonstrates a computer science concept known as **polymorphism**. In brief, polymorphism is a programming language's ability to treat an object differently based on context.

A OboePlayer is of course a OboePlayer, but it is also a Person. Because it derives from Person, you could use a OboePlayer object anywhere you'd use a Person object.

This example demonstrates how you can treat a `OboePlayer` as a `Person`:

```
func phonebookName(_ person: Person) -> String {
    return "\(person.lastName), \(person.firstName)"
}

let person = Person(firstName: "Johnny", lastName: "Appleseed")
let oboePlayer = OboePlayer(firstName: "Jane",
                             lastName: "Appleseed")

phonebookName(person) // Appleseed, Johnny
phonebookName(oboePlayer) // Appleseed, Jane
```

Because `OboePlayer` derives from `Person`, it is a valid input into the function `phonebookName(_)`. More importantly, the function has no idea that the object passed in is anything *other* than a regular `Person`. It can only observe the elements of `OboePlayer` that are defined in the `Person` base class.

With the polymorphism characteristics provided by class inheritance, Swift is treating the object pointed to by `oboePlayer` differently based on the context. This can be particularly useful to you when you have diverging class hierarchies, but want to have code that operates on a common type or base class.

Runtime hierarchy checks

Now that you are coding with polymorphism, you will likely find situations where the specific type behind a variable can be different. For instance, you could define a variable `hallMonitor` as a `Student`:

```
var hallMonitor = Student(firstName: "Jill",
                           lastName: "Bananapeel")
```

But what if `hallMonitor` were a more derived type, such as an `OboePlayer`?

```
hallMonitor = oboePlayer
```

Because `hallMonitor` is defined as a `Student`, the compiler won't allow you to attempt calling properties or methods for a more derived type.

Fortunately, Swift provides the `as` operator to treat a property or a variable as another type:

- `as`: Cast to a specific type that is known at compile time to succeed, such as casting to a supertype.
- `as?`: An optional downcast (to a subtype). If the downcast fails, the result of the expression will be `nil`.

- `as!`: A forced downcast. If the downcast fails, the program will crash. Use this rarely, and only when you are certain the cast will never fail.

These can be used in various contexts to treat the `hallMonitor` as a `BandMember`, or the `oboePlayer` as a less-derived `Student`.

```
oboePlayer as Student
(oboePlayer as Student).minimumPracticeTime // ERROR: No longer a band
member!

hallMonitor as? BandMember
(hallMonitor as? BandMember)?.minimumPracticeTime // 4 (optional)

hallMonitor as! BandMember // Careful! Failure would lead to a runtime
crash.
(hallMonitor as! BandMember).minimumPracticeTime // 4 (force unwrapped)
```

The optional downcast `as?` is particularly useful in `if let` or `guard` statements:

```
if let hallMonitor = hallMonitor as? BandMember {
    print("This hall monitor is a band member and practices
        at least \(hallMonitor.minimumPracticeTime)
        hours per week.")
}
```

You may be wondering under what contexts you would use the `as` operator by itself. Any object contains all the properties and methods of its parent class, so what use is casting it to something it already is?

Swift has a strong type system, and the interpretation of a specific type can have an effect on **static dispatch**, or the decision of which specific operation is selected at compile time.

Sound confusing? How about an example!

Assume you have two functions with identical names and parameter names for two different parameter types:

```
func afterClassActivity(for student: Student) -> String {
    return "Goes home!"
}

func afterClassActivity(for student: BandMember) -> String {
    return "Goes to practice!"
}
```

If you were to pass `oboePlayer` into `afterClassActivity(for:)`, which one of these implementations would get called? The answer lies in Swift's dispatch rules, which in this case will select the more specific version that takes in an `OboePlayer`.

If instead you were to cast `oboePlayer` to a `Student`, the `Student` version would be called:

```
afterClassActivity(for: oboePlayer) // Goes to practice!  
afterClassActivity(for: oboePlayer as Student) // Goes home!
```

Inheritance, methods and overrides

Subclasses receive all properties and methods defined in their superclass, plus any additional properties and methods the subclass defines for itself. In that sense, subclasses are *additive*; for example, you've already seen that the `Student` class can add additional properties and methods for handling a student's grades. These properties and methods wouldn't be available to any `Person` class instances, but they *would* be available to `Student` subclasses.

Besides creating their own methods, subclasses can *override* methods defined in their superclass. Assume that student athletes become ineligible for the athletics program if they're failing three or more classes. That means you need to keep track of failing grades somehow.

```
class StudentAthlete: Student {  
    var failedClasses: [Grade] = []  
  
    override func recordGrade(_ grade: Grade) {  
        super.recordGrade(grade)  
  
        if grade.letter == "F" {  
            failedClasses.append(grade)  
        }  
    }  
  
    var isEligible: Bool {  
        return failedClasses.count < 3  
    }  
}
```

In this example, the `StudentAthlete` class overrides `recordGrade(_:)` so it can keep track of any courses the student has failed. The `StudentAthlete` class then has its own computed property, `isEligible`, that uses this information to determine the athlete's eligibility.

When overriding a method, use the `override` keyword before the method declaration.

If your subclass were to have an identical method declaration as its superclass, but you omitted the `override` keyword, Swift would indicate a build error:

```
class StudentAthlete: Student {  
    var failedClasses: [Grade] = []  
  
    func recordGrade(_ grade: Grade) {  
        super.recordGrade(grade)  
  
        if grade.letter == "F" {  
            failedClasses.append(grade)  
        }  
    }  
  
    var isEligible: Bool {  
        return failedClasses.count < 3  
    }  
}
```

Overriding declaration requires an 'override' keyword

This makes it very clear whether a method is an override of an existing one or not.

Introducing super

You may have also noticed the line `super.recordGrade(grade)` in the overridden method. The `super` keyword is similar to `self`, except it will invoke the method in the nearest implementing superclass. In the example of `recordGrade(_:)` in `StudentAthlete`, calling `super.recordGrade(grade)` will execute the method as defined in the `Student` class.

Remember how inheritance let you define `Person` with first name and last name properties and avoid repeating those properties in subclasses? Similarly, being able to call the superclass methods means you can write the code to record the grade once in `Student` and then call “up” to it as needed in subclasses.

Although it isn’t always required, it’s often important to call `super` when overriding a method in Swift. The `super` call is what will record the grade itself in the grades array, because that behavior isn’t duplicated in `StudentAthlete`. Calling `super` is also a way of avoiding the need for duplicate code in `StudentAthlete` and `Student`.

When to call super

As you may notice, exactly *when* you call `super` can have an important effect on your overridden method.

Suppose you replace the overridden `recordGrade(_:)` method in the `StudentAthlete` class with the following version that recalculates the `failedClasses` each time a grade is recorded:

```
override func recordGrade(_ grade: Grade) {
    var newFailedClasses: [Grade] = []
    for grade in grades {
        if grade.letter == "F" {
            newFailedClasses.append(grade)
        }
    }
    failedClasses = newFailedClasses

    super.recordGrade(grade)
}
```

This version of `recordGrade(_:)` uses the `grades` array to find the current list of failed classes. If you've spotted a bug in the code above, good job! Since you call `super` last, if the new `grade.letter` is an `F`, the code won't update `failedClasses` properly.

While it's not a hard rule, it's generally best practice to call the `super` version of a method first when overriding. That way, the superclass won't experience any side effects introduced by its subclass, and the subclass won't need to know the superclass's implementation details.

Preventing inheritance

Sometimes you'll want to disallow subclasses of a particular class. Swift provides the `final` keyword for you to guarantee a class will never get a subclass:

```
final class FinalStudent: Person {}
class FinalStudentAthlete: FinalStudent {} // Build error!
```

By marking the `FinalStudent` class `final`, you tell the compiler to prevent any classes from inheriting from `FinalStudent`. This can remind you — or others on your team! — that a class wasn't designed to have subclasses.

Additionally, you can mark individual *methods* as `final`, if you want to allow a class to have subclasses, but protect individual methods from being overridden:

```
class AnotherStudent: Person {
    final func recordGrade(_ grade: Grade) {}
}

class AnotherStudentAthlete: AnotherStudent {
    override func recordGrade(_ grade: Grade) {} // Build error!
}
```

There are benefits to initially marking any new class you write as `final`. This tells the compiler it doesn't need to look for any more subclasses, which can shorten compile time, and it also requires you to be very explicit when deciding to subclass a class previously marked `final`. You will learn more about controlling who can override a class in Chapter 18, "Access Control and Code Organization".

Inheritance and class initialization

The previous chapter briefly introduced you to class initializers, which are similar to their struct counterparts. With subclasses, there are a few more considerations with regard to how you set up instances.

Note: In the chapter's playground I have renamed `Student` and `StudentAthlete` to `NewStudent` and `NewStudentAthlete` in order to keep both versions working side-by-side.

Modify the `StudentAthlete` class to add a list of sports an athlete plays:

```
class StudentAthlete: Student {  
    var sports: [String]  
    // original code  
}
```

Because `sports` doesn't have an initial value, `StudentAthlete` must provide one in its own initializer:

```
class StudentAthlete: Student {  
    var sports: [String]  
  
    init(sports: [String]) {  
        self.sports = sports  
        // Build error - super.init isn't called before  
        // returning from initializer  
    }  
    // original code  
}
```

Uh-oh! The compiler complains that you didn't call `super.init` by the end of the initializer:

```
27
28 class StudentAthlete: Student {
29     var sports: [String]
30
31     init(sports: [String]) {
32         self.sports = sports
33         // Build error – super.init isn't called before
34         // returning from initializer
35     }
36 }
```

! Super.init isn't called before returning from initializer

Initializers in subclasses are *required* to call `super.init` because without it, the superclass won't be able to provide initial states for all its stored properties — in this case, `firstName` and `lastName`.

Let's make the compiler happy:

```
class StudentAthlete: Student {
    var sports: [String]

    init(firstName: String, lastName: String, sports: [String]) {
        self.sports = sports
        super.init(firstName: firstName, lastName: lastName)
    }
    // original code
}
```

The initializer now calls the initializer of its superclass, and the build error is gone.

Notice that the initializer now takes in a `firstName` and a `lastName` to satisfy the requirements for calling the `Person` initializer.

You also call `super.init` *after* you initialize the `sports` property, which is an enforced rule.

Two-phase initialization

Because of Swift's requirement that all stored properties have initial values, initializers in subclasses must adhere to Swift's convention of **two-phase initialization**.

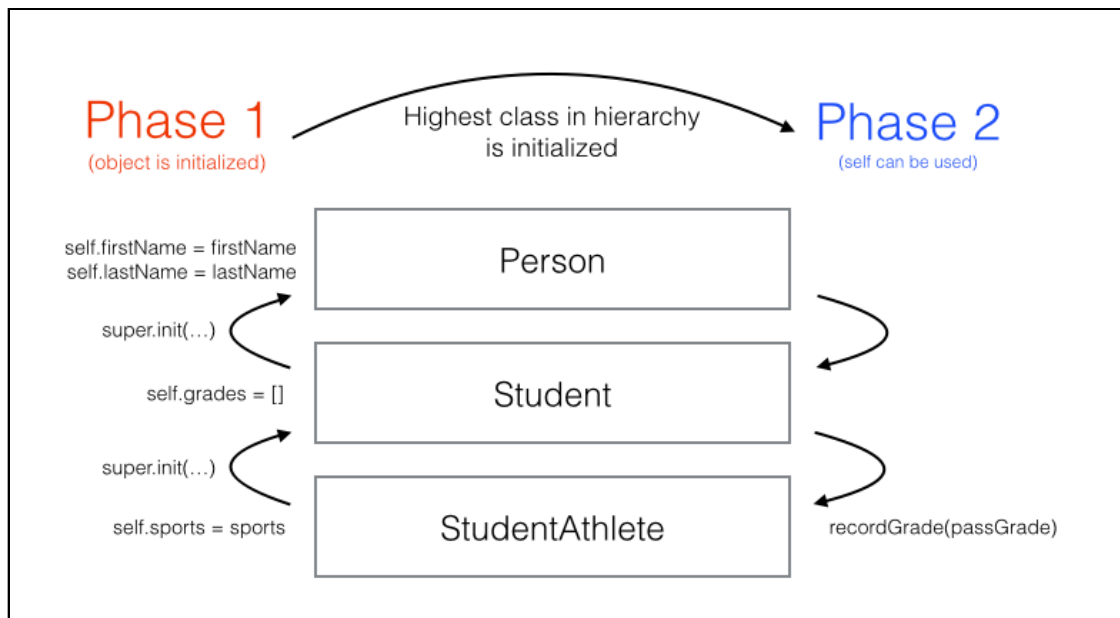
- **Phase one:** Initialize all of the stored properties in the class instance, from the bottom to the top of the class hierarchy. You can't use properties and methods until phase one is complete.

- **Phase two:** You can now use properties and methods, as well as initializations that require the use of `self`.

Without two-phase initialization, methods and operations on the class might interact with properties before they've been initialized.

The transition from phase one to phase two happens after you've initialized all stored properties in the base class of a class hierarchy.

In the scope of a subclass initializer, you can think of this as coming after the call to `super.init`.



Here's the `StudentAthlete` class again, with athletes automatically getting a starter grade:

```
class StudentAthlete: Student {
    var sports: [String]

    init(firstName: String, lastName: String, sports: [String]) {
        // 1
        self.sports = sports
        // 2
        let passGrade = Grade(letter: "P", points: 0.0,
                               credits: 0.0)
        // 3
        super.init(firstName: firstName, lastName: lastName)
        // 4
        recordGrade(passGrade)
    }
    // original code
}
```


The above initializer shows two-phase initialization in action.

1. First, you initialize the `sports` property of `StudentAthlete`. This is part of the first phase of initialization and has to be done early, before you call the superclass initializer.
2. Although you can create local variables for things like grades, you can't call `recordGrade(_:)` yet because the object is still in the first phase.
3. You call `super.init`. When this returns, you know that you've also initialized every class in the hierarchy, because the same rules are applied at every level.
4. After `super.init` returns, the initializer is in phase 2, so you call `recordGrade(_:)`.

Mini-exercise

What's different in the two-phase initialization in the base class `Person`, as compared to the others?

Required and convenience initializers

You already know it's possible to have multiple initializers in a class, which means you could potentially call *any* of those initializers from a subclass.

Often, you'll find that your classes have various initializers that simply provide a "convenient" way to initialize an object:

```
class Student {
    let firstName: String
    let lastName: String
    var grades: [Grade] = []

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    init(transfer: Student) {
        self.firstName = transfer.firstName
        self.lastName = transfer.lastName
    }

    func recordGrade(_ grade: Grade) {
        grades.append(grade)
    }
}
```

In this example, the `Student` class can be built with another `Student` object. Perhaps the student switched majors? Both initializers fully set the first and last names.

Subclasses of `Student` could potentially rely on the `Student`-based initializer when they make their call to `super.init`. Additionally, the subclasses might not even provide a method to initialize with first and last names. You might decide the first and last name-based initializer is important enough that you want it to be available to *all* subclasses.

Swift supports this through the language feature known as **required initializers**:

```
class Student {
    let firstName: String
    let lastName: String
    var grades: [Grade] = []

    required init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
    // original code
}
```

In the modified version of `Student` above, the first and last name-based initializer has been marked with the keyword `required`. This keyword will force all subclasses of `Student` to implement this initializer.

Now that there's a required initializer on `Student`, `StudentAthlete` *must* override and implement it too.

```
class StudentAthlete: Student {
    // Now required by the compiler!
    required init(firstName: String, lastName: String) {
        self.sports = []
        super.init(firstName: firstName, lastName: lastName)
    }
    // original code
}
```

Notice how the `override` keyword isn't required with required initializers. In its place, the `required` keyword must be used to make sure that any subclass of `StudentAthlete` still implements this required initializer.

You can also mark an initializer as a **convenience** initializer:

```
class Student {
    convenience init(transfer: Student) {
        self.init(firstName: transfer.firstName,
                  lastName: transfer.lastName)
    }
    // original code
}
```

The compiler forces a convenience initializer to call a non-convenience initializer (directly or indirectly), instead of handling the initialization of stored properties itself.

A non-convenience initializer is called a **designated** initializer and is subject to the rules of two-phase initialization. All initializers you’ve written in previous examples were in fact designated initializers.

You might want to mark an initializer as convenience if you only use that initializer as an easy way to initialize an object, but you still want it to leverage one of your designated initializers.

Here’s a summary of the compiler rules for using designated and convenience initializers:

1. A designated initializer must call a designated initializer from its immediate superclass.
2. A convenience initializer must call another initializer from the same class.
3. A convenience initializer must ultimately call a designated initializer.

Mini-exercise

Create two more convenience initializers on `Student`. Which other initializers are you able to call?

When and why to subclass

This chapter has introduced you to class inheritance, along with the numerous programming techniques that subclassing enables.

But you might be asking, “When should I subclass?”

Rarely is there a right or wrong answer to that important question. Understanding the trade-offs can help you make the best decision for any particular case.

Using the `Student` and `StudentAthlete` classes as an example, you might decide you can simply put all of the characteristics of `StudentAthlete` into `Student`:

```
class Student: Person {  
    var grades: [Grade]  
    var sports: [Sport]  
    // original code  
}
```

In reality, this *could* solve all of the use cases for your needs. A `Student` that doesn’t play sports would simply have an empty sports array, and you would avoid some of the added complexities of subclassing.

Single responsibility

In software development, the guideline known as the **single responsibility principle** states that any class should have a single concern. In `Student/StudentAthlete`, you might argue that it shouldn't be the `Student` class's job to encapsulate responsibilities that only make sense to student athletes.

Strong types

Subclassing creates an additional type. With Swift's type system, you can declare properties or behavior based on objects that are student athletes, not regular students:

```
class Team {
    var players: [StudentAthlete] = []

    var isEligible: Bool {
        for player in players {
            if !player.isEligible {
                return false
            }
        }
        return true
    }
}
```

A team has players who are student athletes. If you tried to add a regular `Student` object to the array of players, the type system wouldn't allow it. This can be useful as the compiler can help you enforce the logic and requirement of your system.

Shared base classes

You can subclass a shared base class multiple times by classes that have mutually exclusive behavior:

```
// A button that can be pressed.
class Button {
    func press() {}
}

// An image that can be rendered on a button
class Image {}

// A button that is composed entirely of an image.
class ImageButton: Button {
    var image: Image

    init(image: Image) {
        self.image = image
    }
}
```

```
// A button that renders as text.  
class TextButton: Button {  
    var text: String  
  
    init(text: String) {  
        self.text = text  
    }  
}
```

In this example, you can imagine numerous `Button` subclasses that share only the fact that they can be pressed. The `ImageButton` and `TextButton` classes likely have entirely different mechanisms to render the appearance of a button, so they might have to implement their own behavior when the button is pressed.

You can see here how storing image and text in the `Button` class — not to mention any other kind of button there might be — would quickly become impractical. It makes sense for `Button` to be concerned with the press behavior, and the subclasses to handle the actual look and feel of the button.

Extensibility

Sometimes you simply must subclass if you're extending the behavior of code you don't own. In the example above, it's possible `Button` is part of a framework you're using, and there's no way you can modify or extend the source code to fit your needs.

In that case, subclass `Button` so you can add your custom subclass and use it with code that's expecting an object of type `Button`. Using access control, discussed in detail in Chapter 18, "Access Control and Code Organization", the author of a class can designate if any of the members of a class can be overridden or not.

Identity

Finally, it's important to understand that classes and class hierarchies model what objects *are*. If your goal is to share behavior (what objects *can do*) between types, more often than not you should prefer protocols over subclassing. You'll learn about protocols in Chapter 16, "Protocols".

Understanding the class lifecycle

In the previous chapter, you learned that objects are created in memory and that they're stored on the heap. Objects on the heap are *not* automatically destroyed, because the heap is simply a giant pool of memory. Without the utility of the call stack, there's no automatic way for a process to know that a piece of memory will no longer be in use.

In Swift, the mechanism for deciding when to clean up unused objects on the heap is known as **reference counting**. In short, each object has a reference count that's incremented for each constant or variable with a reference to that object, and decremented each time a reference is removed.

Note: You might see the reference count called a “retain count” in other books and online resources. They refer to the same thing!

When a reference count reaches zero, that means the object is now abandoned since nothing in the system holds a reference to it. When that happens, Swift will clean up the object.

Here's a demonstration of how the reference count changes for an object. Note that there's only one actual object created in this example; the one object just has many references to it.

```
var someone = Person(firstName: "Johnny", lastName: "Appleseed")
// Person object has a reference count of 1 (someone variable)

var anotherSomeone: Person? = someone
// Reference count 2 (someone, anotherSomeone)

var lotsOfPeople = [someone, someone, anotherSomeone, someone]
// Reference count 6 (someone, anotherSomeone, 4 references in
// lotsOfPeople)

anotherSomeone = nil
// Reference count 5 (someone, 4 references in lotsOfPeople)

lotsOfPeople = []
// Reference count 1 (someone)

someone = Person(firstName: "Johnny", lastName: "Appleseed")
// Reference count 0 for the original Person object!
// Variable someone now references a new object
```

In this example, you don't have to do any work yourself to increase or decrease the object's reference count. That's because Swift has a feature known as **automatic reference counting** or **ARC**.

While some older languages require you to increment and decrement reference counts in *your* code, the Swift compiler adds these calls automatically at compile time.

Note: If you use a low-level language like C, you're required to manually free memory you're no longer using yourself. Higher-level languages like Java and C# use something called **garbage collection**. In that case, the runtime of the

language will search your process for references to objects, before cleaning up those that are no longer in use. Garbage collection, while more powerful than ARC, comes with a memory utilization and performance cost that Apple decided wasn't acceptable for mobile devices or a general systems language.

Deinitialization

When an object's reference count reaches zero, Swift removes the object from memory and marks that memory as free.

A **deinitializer** is a special method on classes that runs when an object's reference count reaches zero, but before Swift removes the object from memory.

Modify class `Person` as follows:

```
class Person {  
    // original code  
    deinit {  
        print("\(firstName) \(lastName) is being removed  
              from memory!")  
    }  
}
```

Much like `init` is a special method in class initialization, `deinit` is a special method that handles deinitialization. Unlike `init`, `deinit` isn't required and is automatically invoked by Swift. You also aren't required to override it or call `super` within it. Swift will make sure to call each class deinitializer.

If you add this deinitializer, you'll see the message `Johnny Appleseed is being removed from memory!` in the debug area after running the previous example.

What you do in an deinitializer is up to you. Often you'll use it to clean up other resources, save state to a disk or execute any other logic you might want when an object goes out of scope.

Mini-exercises

Modify the `Student` class to have the ability to record the student's name to a list of graduates. Add the name of the student to the list when the object is deallocated.

Retain cycles and weak references

Because classes in Swift rely on reference counting to remove them from memory, it's important to understand the concept of a **retain cycle**.

Add a field representing a classmate — for example, a lab partner — and a deinitializer to class `Student` like this:

```
class Student: Person {
    var partner: Student?
    // original code
    deinit {
        print("\(firstName) is being deallocated!")
    }
}

var alice: Student? = Student(firstName: "Alice",
                               lastName: "Appleseed")
var bob: Student? = Student(firstName: "Bob",
                             lastName: "Appleseed")

alice?.partner = bob
bob?.partner = alice
```

Now suppose both `alice` and `bob` drop out of school:

```
alice = nil
bob = nil
```

If you run this in your playground, you'll notice that you don't see the message `Alice/Bob is being deallocated!`, and Swift doesn't call `deinit`. Why is that?

`Alice` and `Bob` each have a reference to *each other*, so the reference count never reaches zero! To make things worse, by assigning `nil` to `alice` and `bob`, there are no more references to the initial objects. This is a classic case of a retain cycle, which leads to a software bug known as a **memory leak**.

With a memory leak, memory isn't freed up even though its practical lifecycle has ended. Retain cycles are the most common cause of memory leaks.

Fortunately, there's a way that the `Student` object can reference another `Student` without being prone to retain cycles, and that's by making the reference **weak**:

```
class Student: Person {
    weak var partner: Student?
    // original code
}
```

This simple modification marks the `partner` variable as `weak`, which means the reference in this variable will not take part in reference counting. When a reference isn't weak, it's called a **strong reference**, which is the default in Swift. Weak references must be declared as optional types so that when the object that they are referencing is released, it automatically becomes `nil`.

Key points

- **Class inheritance** is one of the most important features of classes and enables **polymorphism**.
- Swift classes use **two-phase initialization** as a safety measure to ensure all stored properties are initialized before they are used.
- **Subclassing** is a powerful tool, but it's good to know when to subclass. Subclass when you want to extend an object and could benefit from an "is-a" relationship between subclass and superclass, but be mindful of the inherited state and deep class hierarchies.
- Class instances have their own lifecycles which are controlled by their **reference counts**.
- **Automatic reference counting**, or **ARC**, handles reference counting for you automatically, but it's important to watch out for **retain cycles**.

Where to go from here?

Classes and structs are the most common types you'll use to model things in your apps, from students to grades to people to teams. As you've seen, classes are more complex, and support things such as reference semantics and inheritance.

In the next chapter, you'll learn about the third type available to you: enumerations.

Challenges

1. Create three simple classes called A, B, and C where C inherits from B and B inherits from A. In each class initializer, call `print("I'm <X>!")` both before and after `super.init()`. Create an instance of C called c. What order do you see each `print()` called in?
2. Implement `deinit` for each class. Create your instance c inside of a `do { }` scope which will cause the reference count to go to zero when it exits the scope. Which order are the classes deinitialized in?
3. Cast the instance of type C to an instance of type A. Which casting operation do you use and why?

4. Create a subclass of `StudentAthlete` called `StudentBaseballPlayer` and include properties for position, number, and battingAverage. What are the benefits and drawbacks of subclassing `StudentAthlete` in this scenario?
5. Fix the following classes so there isn't a memory leak when you add an order:

```
class Customer {  
    let name: String  
    var orders: [Order] = []  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func add(_ order: Order) {  
        order.customer = self  
        orders.append(order)  
    }  
}  
  
class Order {  
    let product: String  
    var customer: Customer?  
  
    init(product: String) {  
        self.product = product  
    }  
}
```

Section III: Your Second Swift 4 and iOS 11 App

This third section of this book corresponds to the **Your Second Swift 4 and iOS 11 App** sections of the course. In this section, you'll create your second app: an iPhone to-do list app called "checklists".

Chapter 22: Table Views

Chapter 23: The Data Model

Chapter 24: Navigation Controllers

Chapter 25: The Add Item Screen

Chapter 26: Delegates and Protocols

Chapter 27: Edit Items

Chapter 28: Saving and Loading

Chapter 29: Lists

Chapter 30: Improved Data Model

Chapter 31: User Defaults

Chapter 32: UI Improvements

Chapter 33: Local Notifications

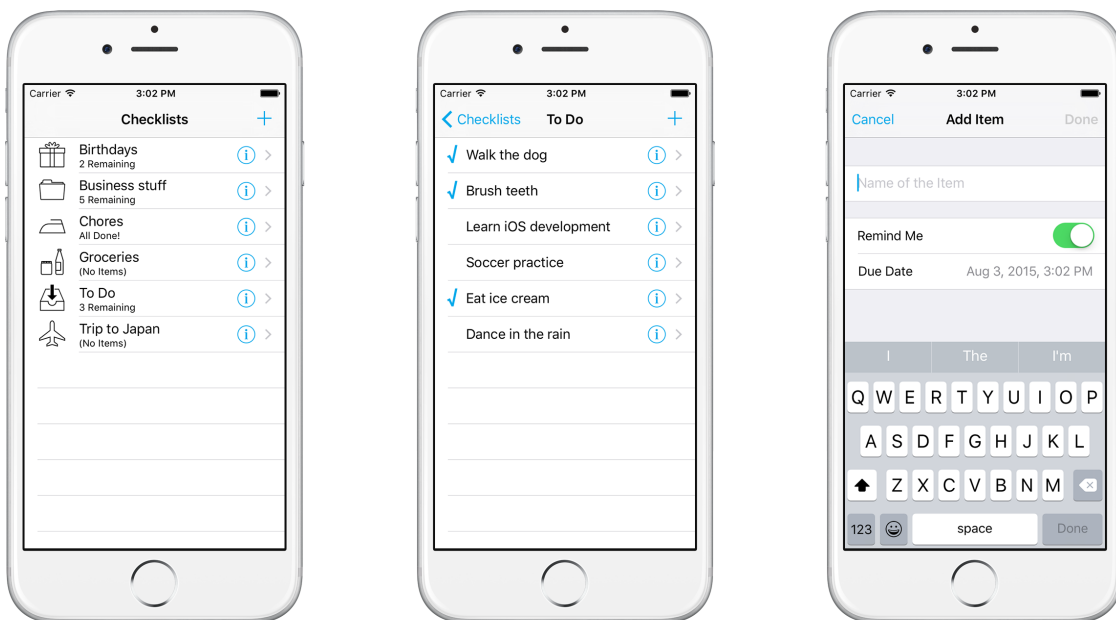
Chapter 22: Table Views

By Fahim Farook and Matthijs Hollemans

Ready to get started on your next app? Let's go!

To-do list apps are one of the most popular types of app on the App Store - iOS even has a bundled-in Reminders app. Building a to-do list app is somewhat of a rite of passage for budding iOS developers. So, it makes sense that you create one as well.

Your own to-do list app, *Checklists*, will look like this when you're finished:



The finished Checklists app

The app lets you organize to-do items into lists and then check off these items once you've completed them. You can also set a reminder on a to-do item that will make the iPhone pop up an alert on the due date, even when the app isn't running.

As far as to-do list apps go, *Checklists* is very basic, but don't let that fool you. Even a simple app such as this already has five different screens and a lot of complexity behind the scenes.

This chapter covers the following:

- **Table views and navigation controllers:** A basic introduction to navigation controllers and table views.
- **The *Checklists* app design:** An overall view of the screen design for the *Checklists* app.
- **Add a table view:** Create your first 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

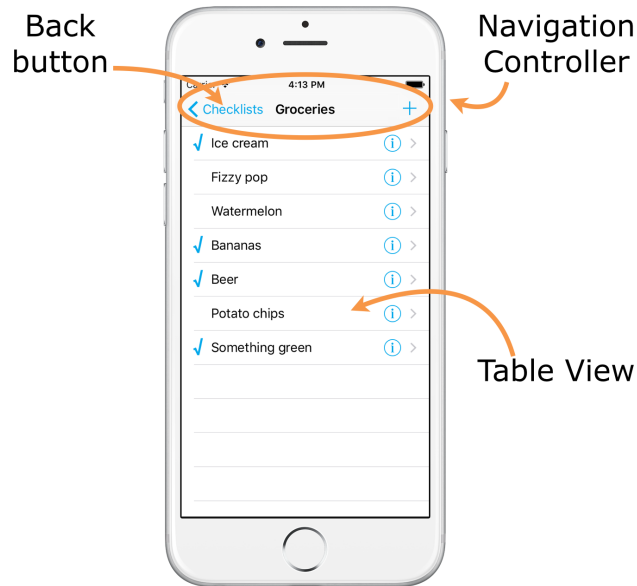
Checklists will introduce you to two of the most commonly used UI (user interface) elements in iOS apps: the table view and the navigation controller.

A **table view** shows a list of things. The three screens above all use a table view. In fact, all of this app's screens use table views. 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 app, tapping the name of a list – “Groceries”, for example – slides in the screen containing the to-do items from that list. The button in the upper-left corner takes you back to the previous screen with a smooth animation. Moving between those screens is the job of the navigation controller.

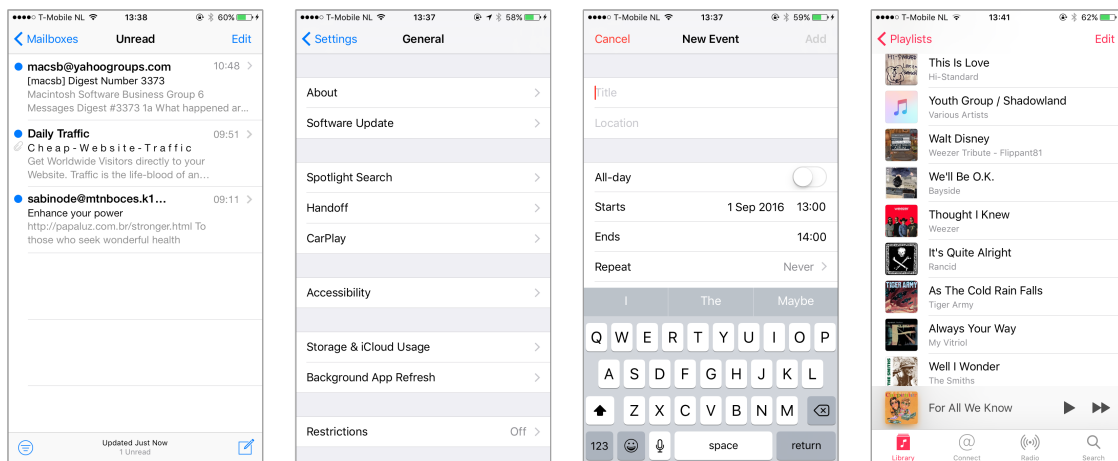
Navigation controllers and table views are often used together.



The grey bar at the top is the navigation bar. The list of items is the table view.

Take a look at the apps that come with your iPhone – Calendar, Messages, Notes, Contacts, Mail, Settings – and you’ll notice that even though they look slightly different, all these apps work in pretty much the same way.

That’s because they all use table views and navigation controllers:



These are all table views inside navigation controllers

(The Music app also has a *tab bar* at the bottom, something you’ll learn about later on.)

If you want to learn how to program iOS apps, you need to master these two components 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.

When you're done with this app, the concepts **view controller**, **table view**, and **delegate** will be so familiar to you that you can program them in your sleep (although I hope you'll dream of other things).

This is a very long read with a lot of source code, so take your time to let it all sink in. I encourage you to experiment with the code that you'll be writing. Change stuff and see what it does, even if it breaks the app.

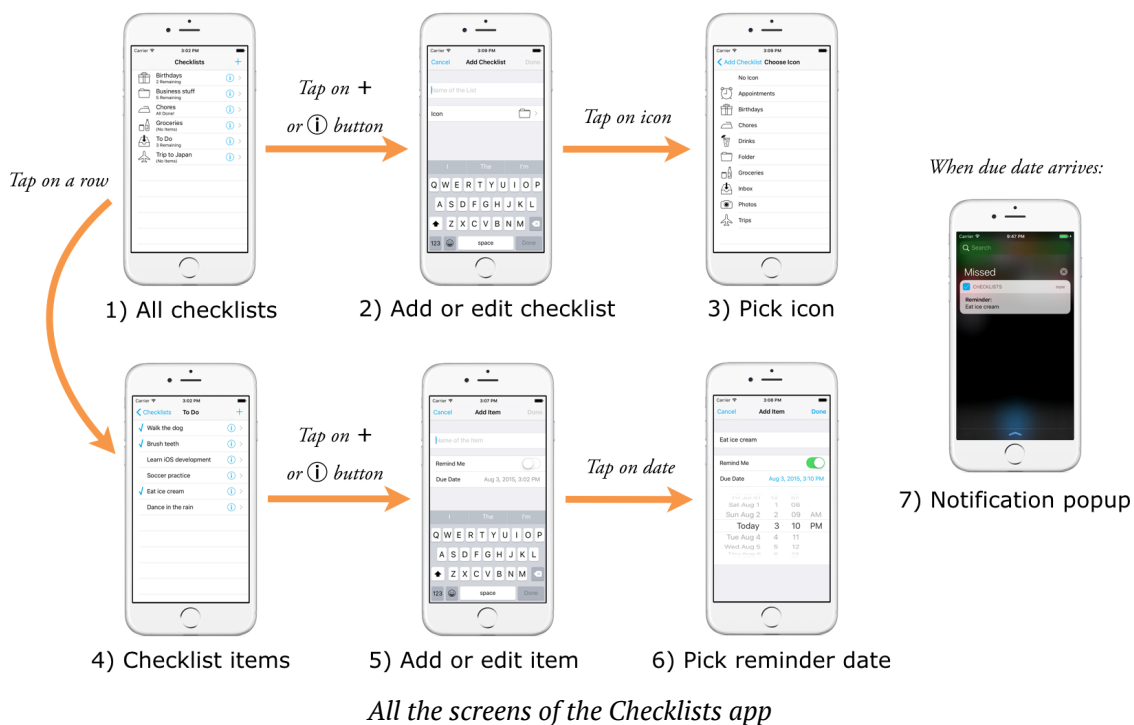
Making mistakes that result in bugs, tearing your hair out in frustration, the light bulb moment when you realize what's wrong, the satisfaction of fixing the bug – they're all essential parts of the developer learning process :]

There's no doubt: playing with code is the quickest way to learn!

By the way, if something is unclear to you – for example, you may wonder why method names in Swift look so funny – then don't panic! Have some faith and keep going... everything will be explained in due course.

The Checklists app design

Just so you know what you're in for, here is an overview of how the *Checklists* app will work:



The main screen of the app shows all your “checklists” (1). You can create multiple lists to organize your to-do items.

A checklist has a name, an icon, and zero or more to-do items. You can edit the name and icon of a checklist in the Add/Edit Checklist screen (2) and (3).

You tap on the checklist’s name to view its to-do items (4).

A to-do item has a description, a checkmark to indicate that the item is done, and an optional due date. You can edit the item in the Add/Edit Item screen (5).

iOS will automatically notify the user of checklist items that have their “remind me” option set (6), even if the app isn’t running (7). That’s a pretty advanced feature, but I think you’ll be up for the task.

You can find the full source code of this app in the Source Code folder, so have a play with it to get a feel for how it works.

Done playing? Then let’s get started!

Important: The *iOS Apprentice* projects are for **Xcode 9.0** and better only. If you’re still using an older version of Xcode, please update to the latest version of Xcode from the Mac App Store.

But don’t get carried away either – often Apple makes beta versions available of upcoming Xcode releases. Please do *not* use an Xcode beta to follow along. Often, the beta versions break things in unexpected ways and you’ll only end up confused. Stick to the official versions for now!

Add a table view

Seeing as table views are so important, you will start out by examining how they work. Making lists has never been this much fun!

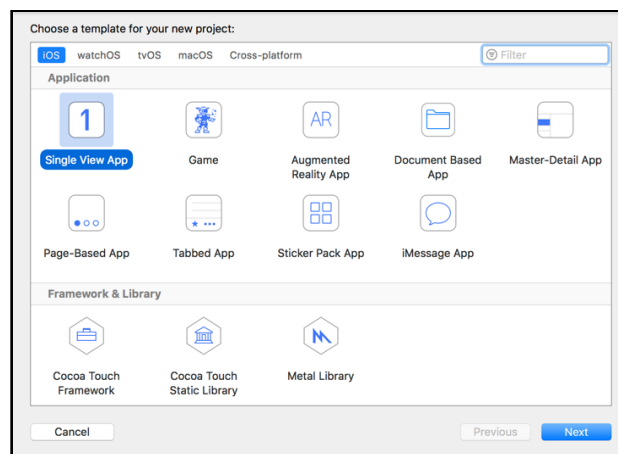
Because smart developers split up the workload into small, simple steps, this is what you’re going to do in this chapter:

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 toggle a checkmark on and off.

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 full-blown app.

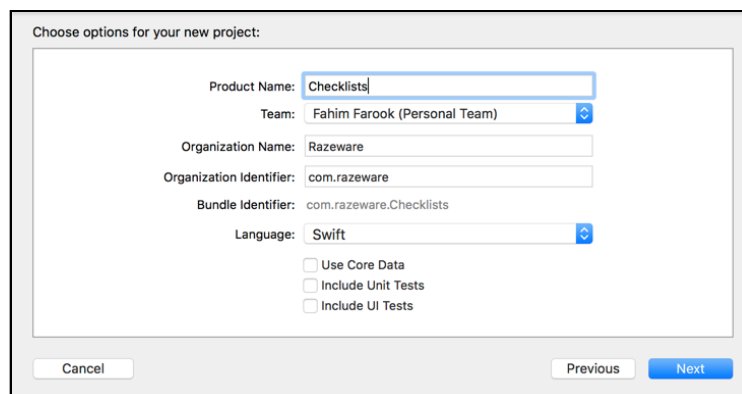
Create the project

► Launch Xcode and start a new project. Choose the **Single View Application** template:



Choosing the Xcode template

Xcode will ask you to fill out a few options:



Choosing the template options

► Fill out these options as follows:

- Product Name: **Checklists**
- Team: Since you already set up your developer account for the previous app (you did, didn't you?) you can select your team here - or, you can just leave this at the default setting.
- Organization Name: Your name or the name of your company

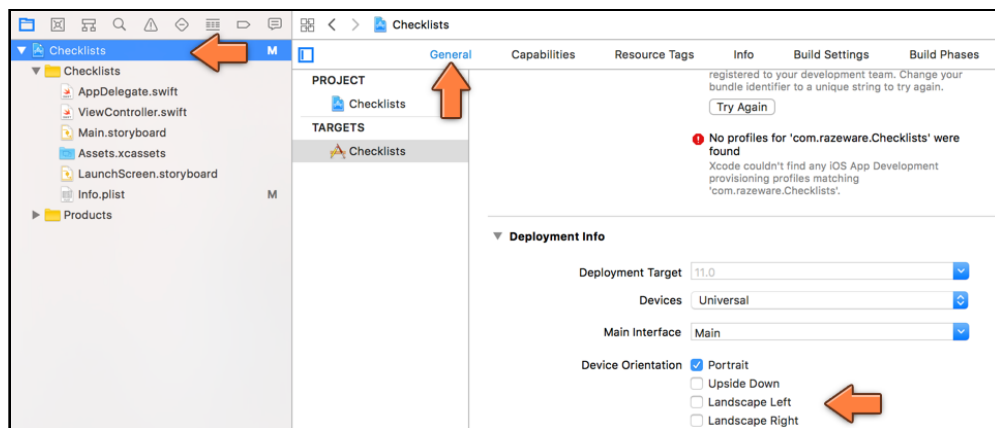
- Organization Identifier: Use your own identifier here, using reverse domain name notation
 - Language: **Swift**
 - Use Core Data, Include Unit Tests, Include UI Tests: these should be off.
- Press **Next** and choose a location for the project.

You can run the app if you want, but as you might remember from the *Bull's Eye* app, at this point it is just a white screen.

Set the app orientation

Checklists will run in portrait orientation only. However, the default project that Xcode just generated also includes landscape support.

- Click on the Checklists project item at the top of the project navigator and go to the **General** tab. Under **Deployment Info**, **Device Orientation**, make sure that only **Portrait** is selected.



The Device Orientation setting

With the landscape options disabled, rotating the device will no longer have any effect. The app always stays in portrait orientation.

Upside down

There is also an Upside Down orientation but you typically won't use it.

If your app supports Upside Down, users are able to rotate their iPhone so that the home button is at the top of the screen instead of at the bottom.

That may be confusing, especially when the user receives a phone call: the microphone is at the wrong end with the phone upside down.

iPad apps, on the other hand, are supposed to support all four orientations including upside-down.

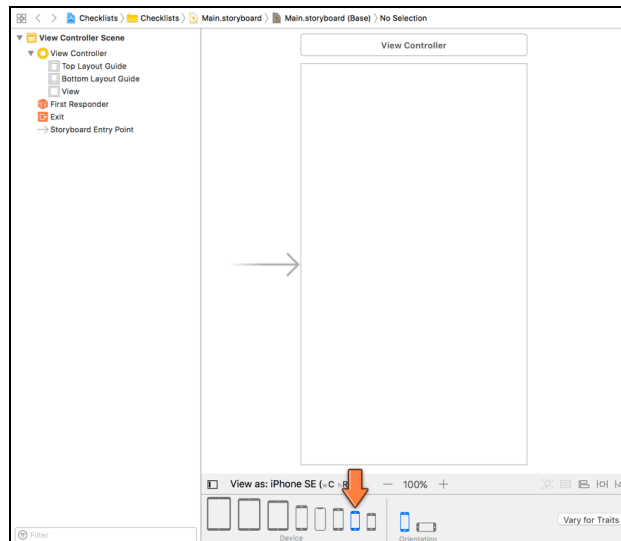
Edit the storyboard

Xcode created a basic app that consists of a single view controller. Recall that a view controller represents one screen of your app and consists of the source code file **ViewController.swift** and a user interface design in **Main.storyboard**.

The storyboard contains the designs of all your app's view controllers inside a single document, with arrows showing the flow between them. In storyboard terminology, each view controller is named a *scene*.

You already used a storyboard in *Bull's Eye* but in this app you will unlock the full power of storyboarding.

► Click on **Main.storyboard** to open Interface Builder.

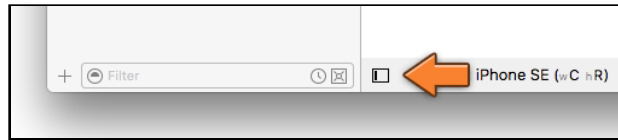


The storyboard editor with the app's only scene

By default, the scene will have the dimensions of a 5.5" iPhone. I used the **View as:** panel at the bottom to switch to the slightly smaller **iPhone SE** because that takes up less room in the book. However, it does not matter which device size you choose to edit the storyboard: the app will automatically resize to fit all iPhone models.

► Select **View Controller** in the Document Outline on the left.

Tip: Recall that the Document Outline shows the view hierarchy of all the scenes in the storyboard. If you cannot see the Document Outline, then click the small square button at the bottom of the Interface Builder window to toggle its visibility.



This button shows and hides the Document Outline

► Press **delete** on your keyboard to remove the **View Controller Scene** from the storyboard. The canvas should be empty and the Document Outline say “No Scenes”.

You do this because you don’t want a regular view controller but a **table view controller**. This is a special type of view controller that makes working with table views a little easier.

The view controller code

But remember, the scene on the storyboard is just half the equation - there's also the Swift code file. And the type specified in code has to match the scene's type. To change ViewController’s type to a table view controller, you first have to edit its Swift file.

► Click on **ViewController.swift** to open it in the source code editor. Change the following line from this:

```
class ViewController: UIViewController {
```

To this:

```
class ChecklistViewController: UITableViewController {
```

With this change you tell the Swift compiler that your own view controller is now a UITableViewController object instead of a regular UIViewController.

Remember that everything starting with “UI” is part of UIKit. These pre-fabricated components serve as the building blocks for your own app.

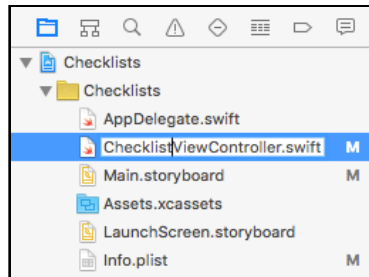
When Xcode made the project, it assumed you wanted the ViewController object to be built on top of a basic UIViewController, but here you’re changing it to use the UITableViewController building block instead.

You also renamed ViewController to ChecklistViewController to give it a more descriptive name. This is your own object – you can tell because its name *doesn’t* start with UI.

Over the course of this app, you will add data and functionality to the `ChecklistViewController` object to make the app actually do things. You'll also add several new view controllers to the app.

► In the Project navigator on the left, click once to select **ViewController.swift**, and then click again to edit its name. (Don't double-click too fast or you'll open the Swift file inside a new source code editor window.)

Change the filename to **ChecklistViewController.swift**:

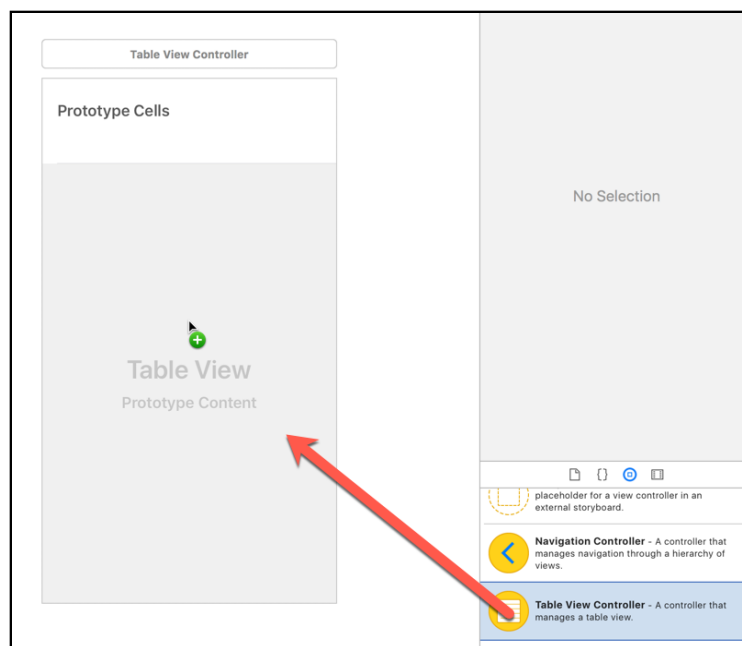


Renaming the Swift file

You might get a warning: “The document could not be saved. The file has been changed by another application.” Click **Save Anyway** to make it go away.

Set the view controller class in the storyboard

► Go back to the storyboard and drag a **Table View Controller** from the Object Library (bottom-right corner) on to the canvas:

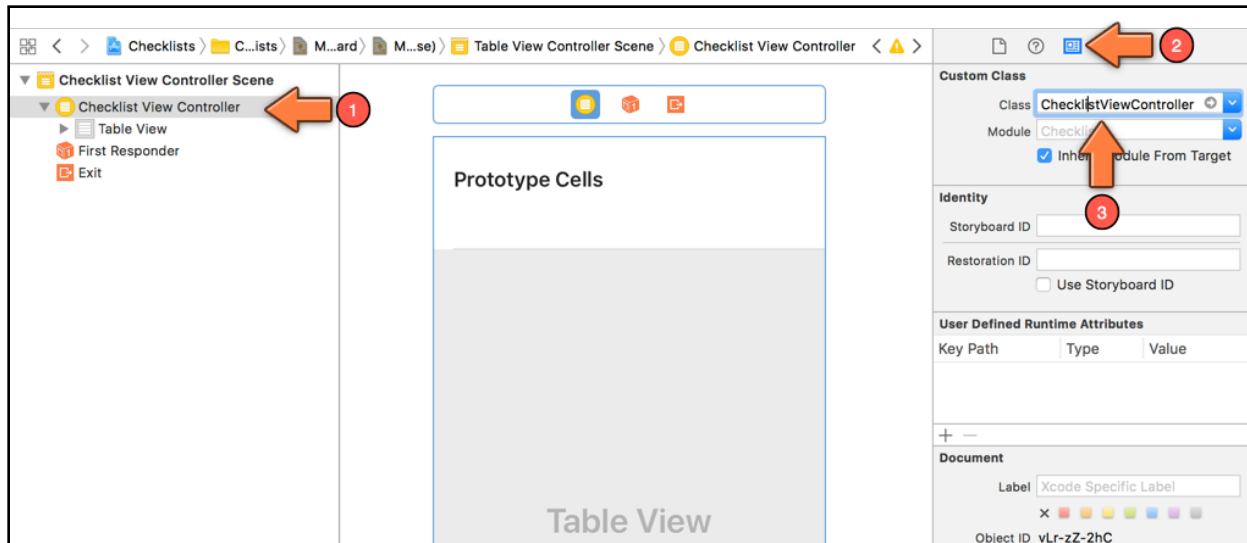


Dragging a Table View Controller into the storyboard

This adds a new Table View Controller scene to the storyboard.

► Go to the **Identity inspector** (the third tab in the inspectors pane on the right of the Xcode window) and under **Custom Class** type **ChecklistViewController** (or choose it using the dropdown list).

Tip: When you do this, make sure the actual Table View Controller is selected, not the Table View inside it. There should be a thin blue border around the scene.



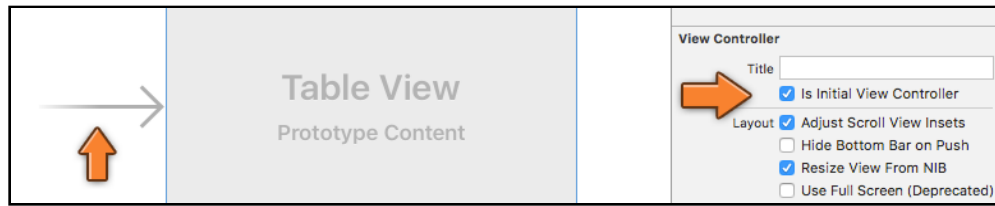
Changing the Custom Class of the Table View Controller

The name of the scene in the Document Outline on the left should change to “Checklist View Controller Scene”. You have successfully changed `ChecklistViewController` from a regular view controller object into a table view controller.

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.

Set the initial view controller

If there is no big arrow pointing towards your new table view controller, then go to the **Attributes inspector** and check **Is Initial View Controller**.



The arrow points at the initial view controller

The initial view controller is the first screen that your users will see. Without one of these, iOS won't know which view controller to load from your storyboard when the app starts up and you'll end up staring at a black screen.

► Run the app on the Simulator.

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.



The app now uses a table view controller

By the way, it doesn't really matter which Simulator you use. Table views resize themselves to the dimensions of the device, and the app will work equally well on the small iPhone SE or the huge iPhone X.

Personally, I use the iPhone SE Simulator because it's compact, but remember that you can open any of the simulators and then simply resize the simulator window by dragging on the corners, just like you resize any macOS window.

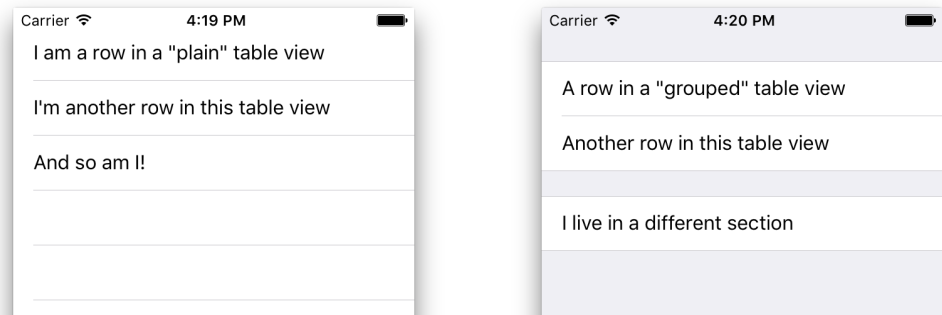
Note: When you build the app, Xcode gives the warning "Prototype table cells must have reuse identifiers". Don't worry about this for now, we'll fix it soon.

The anatomy of a table view

First, let's talk a bit more about table views. A `UITableView` object displays a list of items.

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.

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)

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 *Checklists* app.

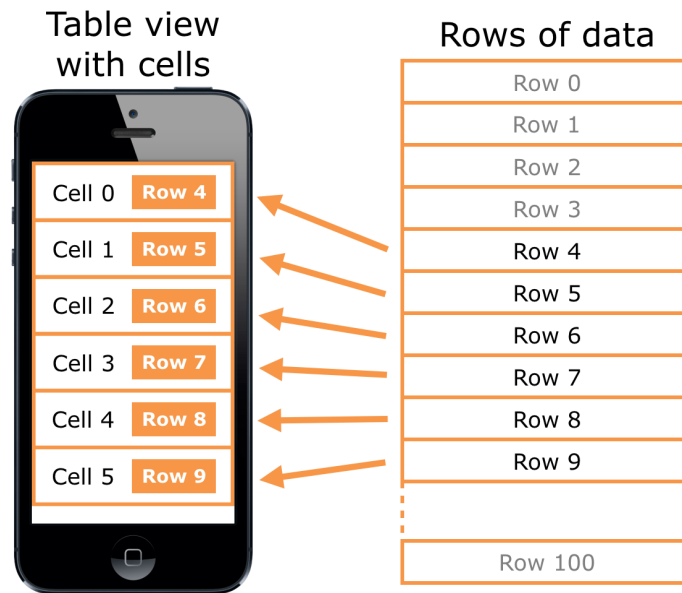
The data for a table comes in the form of **rows**. In the first version of *Checklists*, each row will correspond to a to-do item that you can check off when you're done with it.

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.

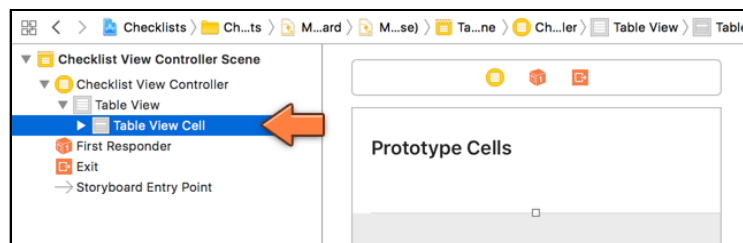


Cells display the contents of rows

Add a prototype cell

In the past, you had to put in quite a bit of effort to create cells for your tables. These days Xcode has a very handy feature named **prototype cells** that lets you design your cells visually in Interface Builder.

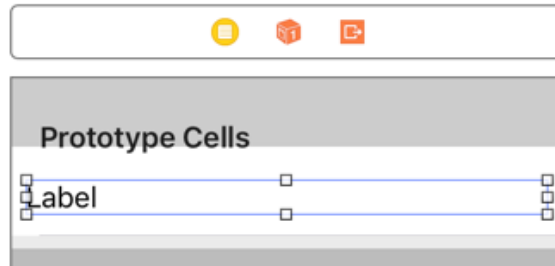
► Open the storyboard and click the empty cell (the white row below the Prototype Cells label) to select it.



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.

► Drag a **Label** from the Object Library on to the white area in the table view representing the cell. Make sure the label spans the entire width of the cell (but leave a small margin on the sides).

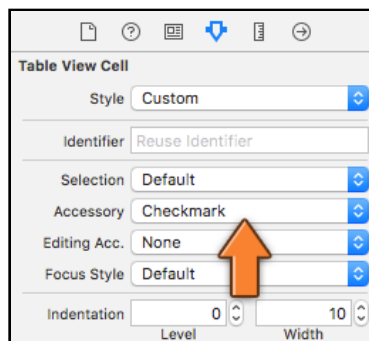


Adding the label to the prototype cell

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 on the Document Outline. It has to be inside the Content View for the table view cell.

Besides the label you will also add a checkmark to the cell's design. The checkmark is provided by something called the **accessory**, a built-in view that appears on the right side of the cell. You can choose from a few standard accessory controls or provide your own.

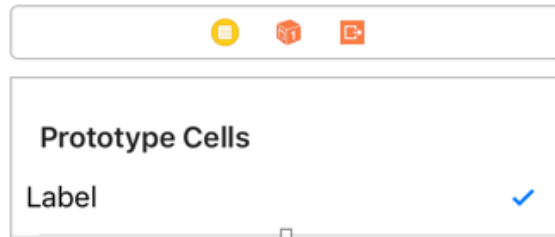
► Select the **Table View Cell** again. In the **Attributes inspector**, set the **Accessory** field to **Checkmark**:



Changing the accessory to get a checkmark

(If you don't see this option, then make sure you selected the Table View Cell, not the Content View or Label below it.)

Your design should now look something like this:



The design of the prototype cell: a label and a checkmark

Note: You may want to resize the label a bit so that it doesn't overlap the checkmark.

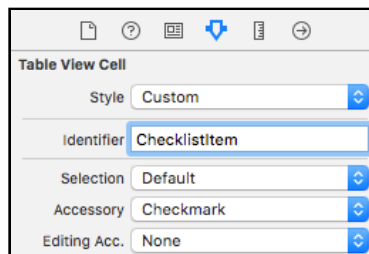
You also need to set a **reuse identifier** on the cell. This is an internal name that the table view uses to find free cells to reuse when rows scroll off the screen and new rows must become visible.

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.

Checklists has only one type of cell but you still need to give it an identifier.

► Type **ChecklistItem** into the Table View Cell's **Identifier** field (you can find this in the **Attributes inspector**).



Giving the table view cell a reuse identifier

► Run the app and you'll see... zip, zilch, nada - 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 **ChecklistViewController.swift** and add the following methods just before the closing bracket at the bottom of the file:

```
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: "CheckListItem",
                                for: indexPath)

    return cell
}
```

These methods look a bit more complicated than the ones you've seen in *Bull's Eye*, 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.

What's a protocol, you ask? Well, it's a standard set of methods that a class must adhere to - a protocol to be followed, so to speak. It allows code to be written in such a way that you know that a given class would implement certain methods (with specific parameters of a given type) but where you don't need to know all the implementation details of the class - such as all its methods. A protocol usually allows you to add functionality for a certain type of operation to a class - for example, handling data for a table view.

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.)

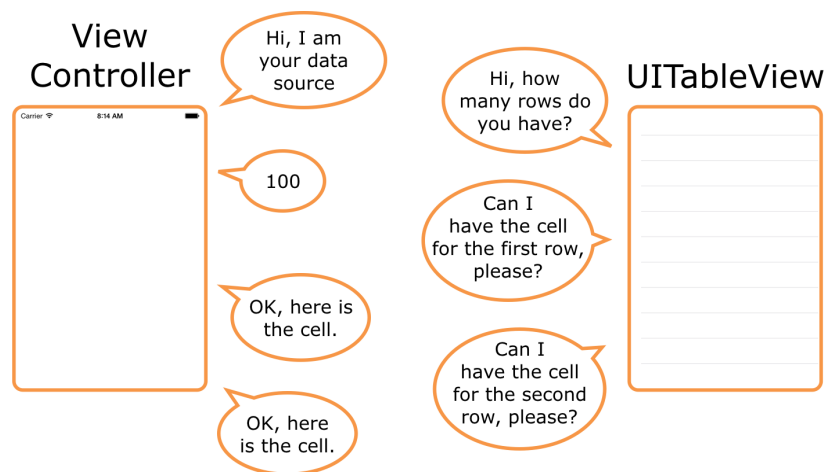
The table view needs to know how many rows of data it has and how it should display each of those rows. But you can't simply dump that data into the table view's lap and be done with it. You don't say: "Dear table view, here are my 100 rows, now go show them on the screen."

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 `cellForRowAt` 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 `ChecklistViewController` 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.

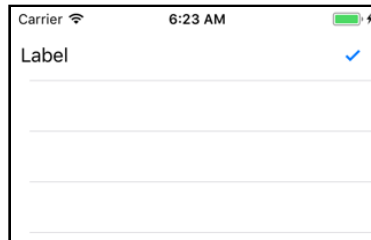
The statements inside a method usually perform some kind of computation using instance variables and any data received through the method’s parameters. When the method is done, `return` says, “Hey, I’m done. Here is the answer I came up with.” The return value is often called the *result* of the method.

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:t)` – 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 you'll see there is a single cell in the table:



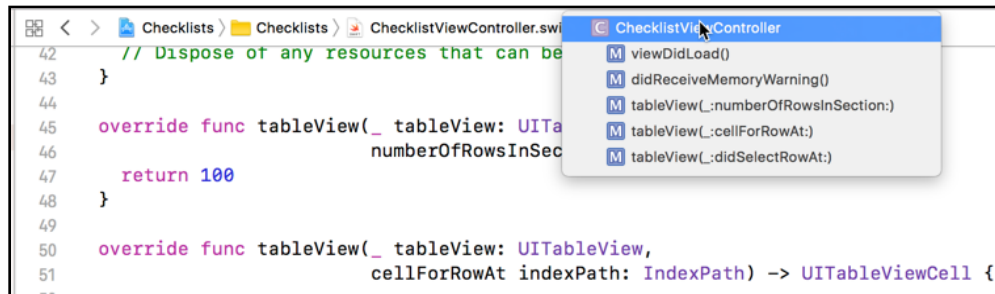
The table now has one row

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 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 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 tableView - the complete signature uniquely identifies the method, if that makes sense?

Exercise: Modify the app so that it shows five rows.

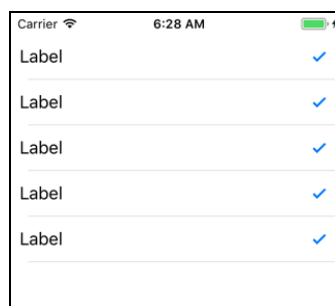
That shouldn't have been too hard:

```
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:

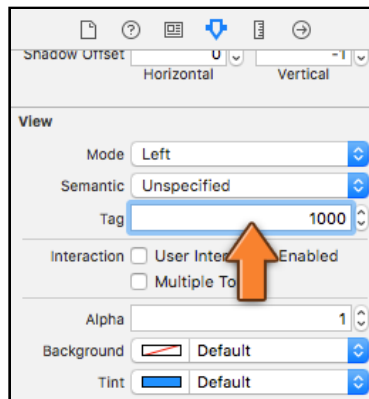
1. Add a prototype cell to the table view in the storyboard.
2. Set a reuse identifier on the prototype cell.
3. Call `tableView.dequeueReusableCell(withIdentifier:for:)`. This makes a new copy of the prototype cell if necessary, or, recycles an existing cell that is no longer in use.

Once you have a cell, you should fill 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 **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.

Double-check to make sure you set the tag on the *Label*, 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 **ChecklistViewController.swift**, change `tableView(_:cellForRowAt:)` to the following:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
                        -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(
                        withIdentifier: "CheckListItem",
                        for: indexPath)

    // Add the following code
    let label = cell.viewWithTag(1000) as! UILabel

    if indexPath.row == 0 {
        label.text = "Walk the dog"
    } else if indexPath.row == 1 {
        label.text = "Brush my teeth"
    } else if indexPath.row == 2 {
        label.text = "Learn iOS development"
    } else if indexPath.row == 3 {
        label.text = "Soccer practice"
    } else if indexPath.row == 4 {
        label.text = "Eat ice cream"
    }
    // End of new code block

    return cell
}
```

You’ve already seen the first line. This gets a copy of the prototype cell – either a new one or a recycled one – and puts it into a local constant named `cell`:

```
let cell = tableView.dequeueReusableCell(
                        withIdentifier: "CheckListItem",
                        for: indexPath)
```

(Recall that this is a constant because it’s declared with `let`, not `var`. It is local because it’s defined inside a method.)

But 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 I 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. The *Checklists* app has no need for this kind of grouping, so you'll ignore the section property of `IndexPath` for now.

The first new line that you've just added is:

```
let label = 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` object.

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 *Bull's Eye*... 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. The next bit shouldn't give you too much trouble:

```
if indexPath.row == 0 {  
    label.text = "Walk the dog"  
} else if indexPath.row == 1 {  
    label.text = "Brush my teeth"  
} else if indexPath.row == 2 {  
    label.text = "Learn iOS development"  
} else if indexPath.row == 3 {  
    label.text = "Soccer practice"  
} else if indexPath.row == 4 {  
    label.text = "Eat ice cream"  
}
```

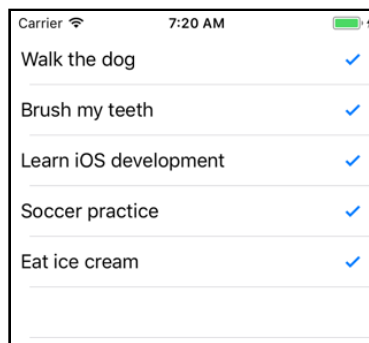
You have seen this `if - else if - else` structure before. It simply looks at the value of `indexPath.row`, which contains the row number, and changes the label's text accordingly. The cell for the first row gets the text "Walk the dog", the cell for the second row gets the text "Brush my teeth", and so on.

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 text:



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`.

Just for the heck of it, let's put 100 rows into the table.

➤ Make `tableView(_:numberOfRowsInSection:)` return 100.

➤ Also, change the code you added earlier to the following:

```
if indexPath.row % 5 == 0 {
    label.text = "Walk the dog"
} else if indexPath.row % 5 == 1 {
    label.text = "Brush my teeth"
} else if indexPath.row % 5 == 2 {
    label.text = "Learn iOS development"
} else if indexPath.row % 5 == 3 {
    label.text = "Soccer practice"
} else if indexPath.row % 5 == 4 {
    label.text = "Eat ice cream"
}
```

This uses the **remainder operator** (also known as the **modulo operator**), represented by the `%` sign, to determine what row you're on.

The `%` operator returns the remainder of a division. You may remember this from doing math in school. For example $13 \% 4 = 1$, because four goes into thirteen 3 times with a remainder of 1. However, $12 \% 4$ is 0 because there is no remainder.

The first row, as well as the sixth, eleventh, sixteenth and so on, will show the text “Walk the dog”. The second, seventh and twelfth row will show “Brush my teeth”. The third, eighth and thirteenth row will show “Learn iOS development”. And so on...

I think you get the picture: every five rows these lines repeat. Rather than typing in all the possibilities all the way up to a hundred, you let the computer calculate this for you (afterall, that is what they are good at):

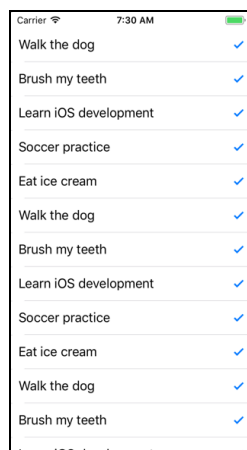
```
First row:    0 % 5 = 0
Second row:   1 % 5 = 1
Third row:    2 % 5 = 2
Fourth row:   3 % 5 = 3
Fifth row:    4 % 5 = 4

Sixth row:    5 % 5 = 0 (same as first row) *** The sequence
Seventh row:  6 % 5 = 1 (same as second row) repeats here
Eighth row:   7 % 5 = 2 (same as third row)
Ninth row:    8 % 5 = 3 (same as fourth row)
Tenth row:    9 % 5 = 4 (same as fifth row)

Eleventh row: 10 % 5 = 0 (same as first row) *** The sequence
Twelfth row:  11 % 5 = 1 (same as second row) repeats again
and so on...
```

If this makes no sense to you at all, then feel free to ignore it. You’re just using this trick to quickly fill up a large table with data.

► Run the app and you should see this:



The table now has 100 rows

Note: To scroll through this table view on the Simulator, you have to pretend you're using an actual iPhone. Click the mouse to “grab” the table view and then drag up or down. Simply swiping without clicking first – the way you'd normally scroll things on the Mac – doesn't work.

Exercise: How many cells do you think this table view uses?

Answer: There are a 100 rows, but only about 14 (or more, depending on the device screen height) fit on the screen at a time. If you count the number of visible rows in the screenshot above you'll get up to 13, but it's possible to scroll the table in such a way that the top cell is still visible while a new cell is pulled in from below. So that makes at least 14 cells.

If you scroll really fast, then I guess it is possible that the table view needs to make a few more temporary cells, but I'm not sure about that. Is this important to know? Not really. You should let the table view take care of juggling the cells behind the scenes. All you have to do is give the table view a cell when it asks for it and fill it up with the data for the corresponding row.

You'll usually have fewer cells than rows. If the app always made a cell for each row, iOS would run out of memory really fast, especially on large tables. Because not all rows can be visible at once, that would be very wasteful and slow. iOS is a good citizen and recycles cells whenever it can.

Now you know why `UITableView` makes the distinction between rows – the data, of which you'll usually have lots – and cells – the visible representation of that data on the screen, of which there are only about a dozen.

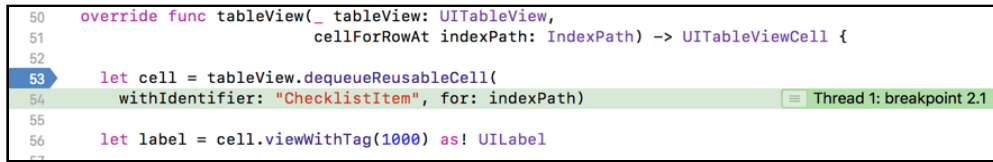
As the song goes, “Rows and cells, rows and cells, tables all the way. Oh what fun it is to learn about new things every day!”

Strange crashes?

A common question on the *iOS Apprentice* forums is, “I'm just following along with the book and suddenly my app crashes... What went wrong?”

If that happens to you, then make sure you haven't set a *breakpoint* on your code by accident. A breakpoint is a debugging tool that stops your program at a specific line and jumps into the Xcode debugger. It may appear like a crash, but your program simply paused.

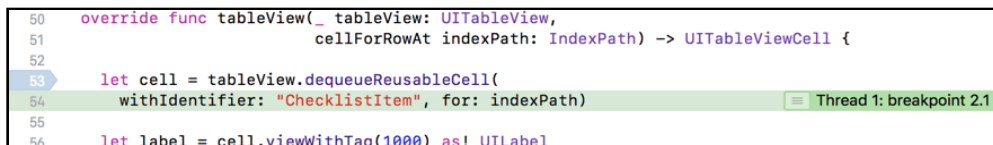
A breakpoint looks like a blue arrow in the left-hand margin (also known as the **gutter**) of the source editor:



The blue arrow sets a breakpoint

If your app suddenly pauses and the source editor shows a blue arrow on a particular line, then you simply hit a breakpoint. Sometimes people click in the margin by mistake and set a breakpoint without even realizing it (I've certainly done that!).

To remove the breakpoint, drag it out of the Xcode window. Or, you can deactivate a breakpoint by simply clicking on it - it will still be there, ready to be activated again by a click, but will not pause code execution. A deactivated breakpoint is indicated by a faded blue arrow.

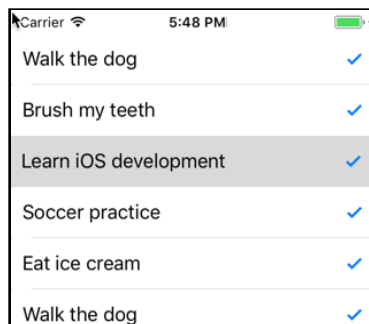


A deactivated breakpoint

By the way, the forums for this book are at forums.raywenderlich.com, so drop by if you have any questions.

Tap 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 tapping the row will toggle the checkmark on and off.



A tapped row stays gray

Taps on rows are handled by the table view's **delegate**. Remember I said 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.

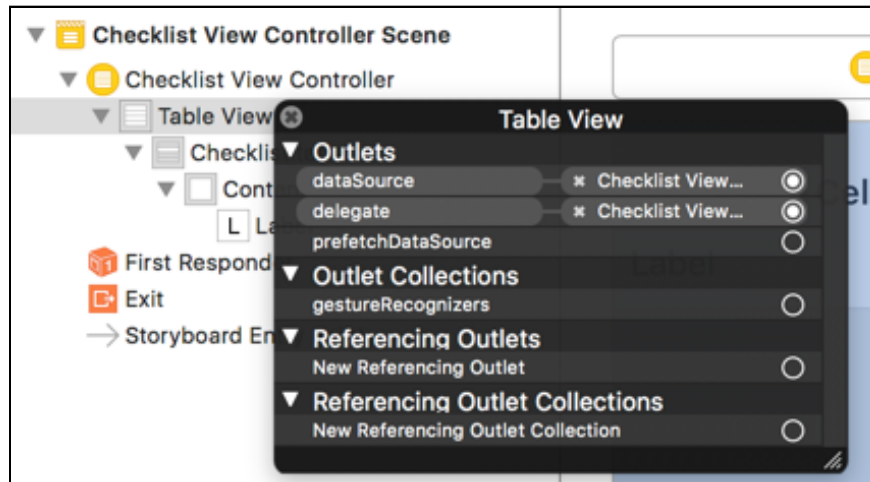
The table view doesn't really care who its data source is or what kind of data your app deals with, just that it can send the `cellForRowAt` message and receive a cell in return. This keeps the table view component simple and moves the responsibility for handling the data to where it belongs: in your 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 make it toggle the checkmark; 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 **ChecklistViewController.swift**:

```
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.

► Let's make `tableView(_:didSelectRowAt:)` toggle the checkmark. Change the method to the following:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    if let cell = tableView.cellForRow(at: indexPath) {
        if cell.accessoryType == .none {
            cell.accessoryType = .checkmark
        } else {
            cell.accessoryType = .none
        }
    }
    tableView.deselectRow(at: indexPath, animated: true)
}
```


The checkmark is part of the cell (the accessory, remember?). So, you first need to find the `UITableViewCell` object for the tapped row. You simply ask the table view: what is the cell at this `indexPath` you've given me?

It is theoretically possible that there is no cell at the specified index-path, for example if that row isn't visible. So, you need to use the special `if let` statement.

The `if let` tells Swift that you only want to perform the code inside the `if` condition only if there really is a `UITableViewCell` object. In this app there always will be one – after all, that's what the user just tapped – but Swift doesn't know that.

Once you have the `UITableViewCell` object, you look at the cell's accessory type, which you can access via the `accessoryType` property. If it is “none”, then you change the accessory to a checkmark; if it is already a checkmark, you change it back to none.

Note: In the above code, to find the cell you call `tableView.cellForRow(at:)`.

It's important to realize this is not the same method as the data source method `tableView(_:cellForRowAt:)` that you added earlier.

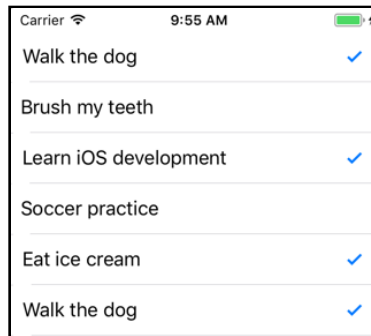
Despite the similar names they are different methods in different objects, performing different tasks. Tricky, eh?

The purpose of your data source method is to deliver a new (or recycled) cell object to the table view when a row becomes visible. You never call this method yourself; only the `UITableView` may call its data source methods.

The purpose of `tableView.cellForRow(at:)` is also to return a cell object, but this is an existing cell for a row that is currently being displayed. It won't create any new cells. If there is no cell for that row yet, it will return the special value `nil`, meaning that no cell could be found. (You use the `if let` statement to “catch” such `nil` values.)

Remember how I said methods should have clear, descriptive names? UIKit is generally pretty good with its names, but this is a case where a very similar name used in two different places can lead to confusion and despair. Beware this pitfall!

► Run the app and try it out. You should be able to toggle the checkmarks on the rows. Sweet!



You can now tap on a row to toggle the checkmark

Note: If the checkmark does not appear or disappear right away but only after you select *another* row, then make sure the method name is not `tableView(_:didDeselectRowAt:)`! You want `didSelect`, not `didDeselect`. Xcode's autocompletion may have fooled you into picking the wrong method name.

Unfortunately, the app has a bug. Here's how to reproduce it:

► Tap a row to remove the checkmark. Scroll that row off the screen and scroll back again (try scrolling really fast). The checkmark has reappeared!

In addition, the checkmark seems to spontaneously disappear from other rows. What is going on here?

Again, it's a matter of cells vs. rows: you have toggled the checkmark on the cell but the cell may be reused for another row when you're scrolling. Whether a checkmark is set or not should be a property of a given row (or rather, the data underlying that row), not the cell.

Instead of using the cell's accessory to remember to show a checkmark or not, you need some way to keep track of the checked status for each row. 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 *Bull's Eye* 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,           // parameter 1  
    numberOfRowsInSection section: Int) // parameter 2  
-> Int {                               // return value
```

```

    }
    override func tableView(
        _ tableView: UITableView,           // parameter 1
        cellForRowAt indexPath: IndexPath) // parameter 2
        -> UITableViewCell {               // return value
    }
    override func tableView(
        _ tableView: UITableView,           // parameter 1
        didSelectRowAt indexPath: IndexPath) { // parameter 2
    }
}

```

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 it a little bit 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`.

The first name, `numberOfRowsInSection`, is used when calling the method. This is known as 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 `ChecklistViewController.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 **22 - Table Views** in the Source Code folder.

Chapter 23: The Data Model

By Fahim Farook and Matthijs Hollemans

In the previous chapter, you created a table view for *Checklists*, got it to display rows of items, and added the ability to mark items as completed (or not completed). However, this was all done using hardcoded, fake data. This would not do for a real to-do app since your users want to store their own custom to-do items.

In order to store, manage, and display to-do information efficiently, you need a data model that allows you to store (and access) to-do information easily. And 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 which are central to iOS programming.
- **The data model:** Creating a data model to hold the data for *Checklists*.
- **Clean up the code:** Simplify your code so that it is easier to understand and maintain.

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.

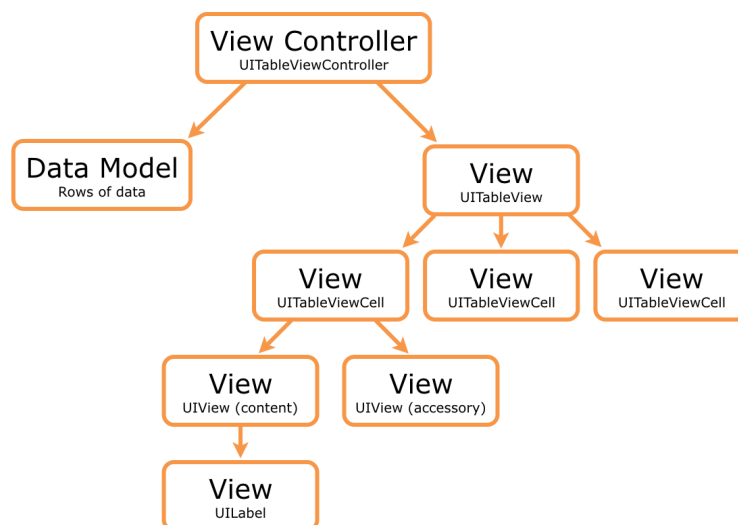
The operations that the data model objects perform are sometimes called the *business rules* or the *domain logic*. For *Checklists*, the checklists and their to-do items form the data model.

- **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 *Checklists*, 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 label and the accessory.

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, but 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 email, and the tweet sheet for sending Twitter messages.

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.

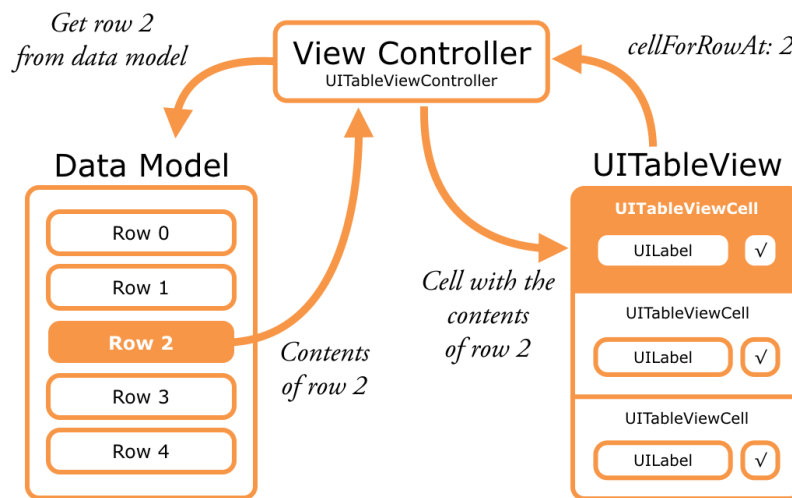
The data model

So far, you've put a bunch of fake data into the table view. The data consists of a text string and a checkmark that can be on or off.

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 to-do items. Each of these items will get its own row in the table.

For each to-do item you need to store two pieces of information: the text (“Walk the dog”, “Brush my teeth”, “Eat ice cream”) and whether the checkmark is set or not.

That is two pieces of information per row, so you need two variables for each row.

The first iteration

First, I’ll show you the cumbersome way to program this. It will work but it isn’t very smart. Even though this is not the best approach, I’d still like you to 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 **ChecklistViewController.swift**, add the following constants right after the `class ChecklistViewController` line.


```
class ChecklistViewController: UITableViewController {  
    let row0text = "Walk the dog"  
    let row1text = "Brush teeth"  
    let row2text = "Learn iOS development"  
    let row3text = "Soccer practice"  
    let row4text = "Eat ice cream"  
    . . .  
}
```

These constants are defined outside of any method (they are not “local”), so they can be used by all of the methods in `ChecklistViewController`.

► 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: "ChecklistItem",  
        for: indexPath)  
    let label = cell.viewWithTag(1000) as! UILabel  
  
    if indexPath.row == 0 {  
        label.text = row0text  
    } else if indexPath.row == 1 {  
        label.text = row1text  
    } else if indexPath.row == 2 {  
        label.text = row2text  
    } else if indexPath.row == 3 {  
        label.text = row3text  
    } else if indexPath.row == 4 {  
        label.text = row4text  
    }  
    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 a constant with the text for that row. Together, those five 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 you're supposed to draw, and put the text from the corresponding constant into the cell.

Handle checkmarks

Now, let's fix the checkmark toggling logic. You no longer want to toggle the checkmark on the cell but at the row (or data) level. To do this, you add five new instance variables to keep track of the “checked” state of each of the rows. (This time the values have to be variables instead of constants since you will be changing the checked/unchecked state for each row.) These new variables are also part of your data model.

► Add the following instance variables:

```
var row0checked = false
var row1checked = false
var row2checked = false
var row3checked = false
var row4checked = false
```

These variables have the data type `Bool`. You've seen the data types `Int` (whole numbers), `Float` (decimal/fractional numbers), and `String` (text) before. A `Bool` variable can hold only two possible values: `true` or `false`.

`Bool` is short for “boolean”, after Englishman George Boole who long ago invented a kind of logic that forms the basis of all modern computing. The fact that computers talk in ones and zeros is largely due to him.

You use `Bool` variables to remember whether something is true (1) or not (0). As a convention, the names of boolean variables often start with the verb “is” or “has”, as in `isHungry` or `hasIceCream`.

The instance variable `row0checked` is `true` if the first row has its checkmark set and `false` if it doesn't. Likewise, `row1checked` reflects whether the second row has a checkmark or not. The same thing goes for the instance variables for the other rows.

Note: How does the compiler know that the type of these variables is `Bool`? You never specified that anywhere.

Remember *type inference* from your code in *Bulls's Eye*? Because you said `var row0checked = false`, Swift assumes that you intended to make this a `Bool`, as `false` is valid only for `Bool` values.

The delegate method that handles taps on table cells will now use these new instance variables to determine whether the checkmark for a row needs to be toggled on or off.

The code in `tableView(_:didSelectRowAt:)` should be something like the following. *Don't make these changes just yet!* Just try to understand what happens first.

```

override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {

    if let cell = tableView.cellForRow(at: indexPath) {
        if indexPath.row == 0 {
            row0checked = !row0checked
            if row0checked {
                cell.accessoryType = .checkmark
            } else {
                cell.accessoryType = .none
            }
        } else if indexPath.row == 1 {
            row1checked = !row1checked
            if row1checked {
                cell.accessoryType = .checkmark
            } else {
                cell.accessoryType = .none
            }
        } else if indexPath.row == 2 {
            row2checked = !row2checked
            if row2checked {
                cell.accessoryType = .checkmark
            } else {
                cell.accessoryType = .none
            }
        } else if indexPath.row == 3 {
            row3checked = !row3checked
            if row2checked {
                cell.accessoryType = .checkmark
            } else {
                cell.accessoryType = .none
            }
        } else if indexPath.row == 4 {
            row4checked = !row4checked
            if row4checked {
                cell.accessoryType = .checkmark
            } else {
                cell.accessoryType = .none
            }
        }
    }
    tableView.deselectRow(at: indexPath, animated: true)
}

```

It should be clear that the code looks at `indexPath.row` to find the row that was tapped, and then performs some logic with the corresponding “row checked” instance variable. But there’s also some new stuff you may not have seen before.

Let’s look at the first `if indexPath.row` statement in detail:

```

if indexPath.row == 0 {
    row0checked = !row0checked
    if row0checked {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }
}

```

```
    }  
  } . . .
```

If `indexPath.row` is 0, the user tapped on the very first row and the corresponding instance variable is `row0checked`.

You do the following to flip that boolean value around:

```
row0checked = !row0checked
```

The `!` symbol is the **logical not** operator. There are a few other logical operators that work on `Bool` values, such as **and** and **or**, which you'll encounter soon enough.

What `!` does is simple: it reverses the meaning of the value. If `row0checked` is `true`, then `!` makes it `false`. Conversely, `!false` is `true`.

Think of `!` as “not”: not yes is no and not no is yes. Yes?

Once you have the new value of `row0checked`, you can use it to show or hide the checkmark:

```
if row0checked {  
    cell.accessoryType = .checkmark  
} else {  
    cell.accessoryType = .none  
}
```

The same logic is used for the other four rows.

In fact, the other rows use the *exact* same logic. The only thing that is different between each of these code blocks is the name of the “row checked” instance variable.

Because the code looks so familiar from one `if` statement to the next, we can improve upon it.

► Replace the current `tableView(_:didSelectRowAt:)` implementation with the following:

```
override func tableView(_ tableView: UITableView,  
                        didSelectRowAt indexPath: IndexPath) {  
  
    if let cell = tableView.cellForRow(at: indexPath) {  
        var isChecked = false  
  
        if indexPath.row == 0 {  
            row0checked = !row0checked  
            isChecked = row0checked  
        } else if indexPath.row == 1 {  
            row1checked = !row1checked  
            isChecked = row1checked  
        } else if indexPath.row == 2 {
```

```
        row2checked = !row2checked
        isChecked = row2checked
    } else if indexPath.row == 3 {
        row3checked = !row3checked
        isChecked = row3checked
    } else if indexPath.row == 4 {
        row4checked = !row4checked
        isChecked = row4checked
    }

    if isChecked {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }
}
tableView.deselectRow(at: indexPath, animated: true)
}
```

Now isn't that a lot shorter than the previous iteration (that you weren't supposed to type in)?

Notice how the logic that sets the checkmark on the cell has moved to the bottom of the method. There is now only one place where this happens.

To make this possible, you store the value of the “row checked” instance variable into the `isChecked` local variable. This temporary variable is just used to remember whether the selected row needs a checkmark or not.

By using a local variable you were able to remove a lot of duplicated code, which is a good thing. You've taken the logic that all rows had in common and moved it out of the `if` statements into a single place.

Note: Code duplication makes programs a lot harder to read. Worse, it invites subtle mistakes that cause hard-to-find bugs. Always be on the lookout for opportunities to remove duplicate code!

Exercise: There was actually a bug in the previous, longer version of this method – did you spot it? That's what happens when you use copy-paste to create duplicate code, like I did when I wrote that method.

► Run the app and observe... that it still doesn't work very well. Initially, you have to tap a couple of times on a row to actually make the checkmark go away.

What's wrong here? Simple: when you declared the `rowXchecked` variables you set their values to `false`.

So `row0checked` and the others indicate that there is no checkmark on their row, but the table draws one anyway. That's because you enabled the checkmark accessory on the prototype cell.

In other words: the data model (the “row checked” variables) and the views (the checkmarks inside the cells) are out-of-sync.

There are a few ways you could try to fix this: you could set the `Bool` variables to `true` to begin with, or you could remove the checkmark from the prototype cell in the storyboard.

Neither is a foolproof solution. What goes wrong here isn't so much that you initialized the “row checked” values wrong or designed the prototype cell wrong, but that you didn't set the cell's `accessoryType` property to the right value in `tableView(_:cellForRowAt:)`.

When you are asked for a new cell, you always should configure all of its properties. The call to `tableView.dequeueReusableCell(withIdentifier:)` could return a cell that was previously used for a row with a checkmark. If the new row shouldn't have a checkmark, then you have to remove it from the cell at this point (and vice versa).

Let's fix that.

► Add the following method to **ChecklistViewController.swift**:

```
func configureCheckmark(for cell: UITableViewCell,
                        at indexPath: IndexPath) {
    var isChecked = false

    if indexPath.row == 0 {
        isChecked = row0checked
    } else if indexPath.row == 1 {
        isChecked = row1checked
    } else if indexPath.row == 2 {
        isChecked = row2checked
    } else if indexPath.row == 3 {
        isChecked = row3checked
    } else if indexPath.row == 4 {
        isChecked = row4checked
    }

    if isChecked {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }
}
```

This new method looks at the cell for a certain row, specified as usual by `indexPath`, and makes the checkmark visible if the corresponding “row checked” variable is `true`, or hides the checkmark if the variable is `false`.

This logic should look very familiar! The only difference with before is that here you don’t toggle the state of the “row checked” variable. You only read it and then set the cell’s accessory.

You’ll call this method from `tableView(_:cellForRowAt:)`, just before you return the cell.

► Change `tableView(_:cellForRowAt:)` to the following (recall that `. . .` means that the existing code at that spot doesn’t change):

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    . . .
    configureCheckmark(for: cell, at: indexPath)
    return cell
}
```

► Run the app again.

Now the app works just fine. Initially all the rows are unchecked. Tapping a row checks it, tapping it again unchecks it. The rows and cells are now always in sync. This code guarantees that each cell always has the value that corresponds to its underlying data row.

External and internal parameter names

The new `configureCheckmark` method has two parameters, `for` and `at`. Its full name is therefore `configureCheckmark(for:at:)`.

`for` and `at` are the *external* names of these parameters.

Adding short prepositions such as “at”, “with”, or “for” is very common in Swift. It makes the name of the method sound like a proper English phrase: “configure checkmark for this cell at that index-path”. Doesn’t it just roll off your tongue?

When you call the method, you always have to include those external parameter names:

```
configureCheckmark(for: someCell, at: someIndexPath)
```

Here, `someCell` is a variable that refers to a `UITableViewCell` object. Likewise, `someIndexPath` is a variable of type `IndexPath`.

You can't write the following:

```
configureCheckmark(someCell, someIndexPath)
```

This won't compile. The app doesn't have a `configureCheckmark` method that doesn't take parameter names, only `configureCheckmark(for:at:)`. The `for` and `at` are an integral part of the method name!

Inside the method you use the *internal* labels `cell` and `indexPath` to refer to the parameters.

```
func configureCheckmark(for cell: UITableViewCell,
                        at indexPath: IndexPath) {
    if indexPath.row == 0 {
        . . .
    }

    cell.accessoryType = .checkmark
    . . .
}
```

You can't write `if at.row == 0` or `for.accessoryType = .checkmark`. That also sounds a little odd, doesn't it?

This split between external and internal labels is unique to Swift and Objective-C and takes some getting used to if you're familiar with other languages.

This naming convention primarily exists so that Swift can talk to older Objective-C code, and this is a good thing since most of the iOS frameworks are still written in Objective-C.

Simplify the code

Why was `configureCheckmark(for:at:)` set up as a method of its own anyway? Well, because you can use it to simplify `tableView(_:didSelectRowAt:)`.

Notice how similar these two methods currently are. That's another case of code duplication that you can get rid of!

You can simplify `didSelectRowAt` by letting `configureCheckmark(for:at:)` do some of the work.

► Replace `tableView(_:didSelectRowAt:)` with the following:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {

    if let cell = tableView.cellForRow(at: indexPath) {
        if indexPath.row == 0 {
```



```

        row0checked = !row0checked
    } else if indexPath.row == 1 {
        row1checked = !row1checked
    } else if indexPath.row == 2 {
        row2checked = !row2checked
    } else if indexPath.row == 3 {
        row3checked = !row3checked
    } else if indexPath.row == 4 {
        row4checked = !row4checked
    }

    configureCheckmark(for: cell, at: indexPath)
}
tableView.deselectRow(at: indexPath, animated: true)
}

```

This method no longer sets or clears the checkmark from the cell, but only toggles the “checked” state in the data model and then calls `configureCheckmark(for:at:)` to update the view.

► Run the app again and it should still work.

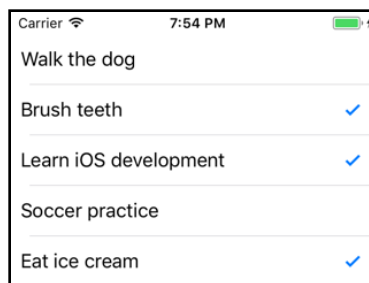
► Change the declarations of the instance variables to the following and run the app again:

```

var row0checked = false
var row1checked = true
var row2checked = true
var row3checked = false
var row4checked = true

```

Now rows 1, 2 and 4 (the second, third and fifth rows) initially have a checkmark while the others don't.



The data model and the table view cells are now always in-sync

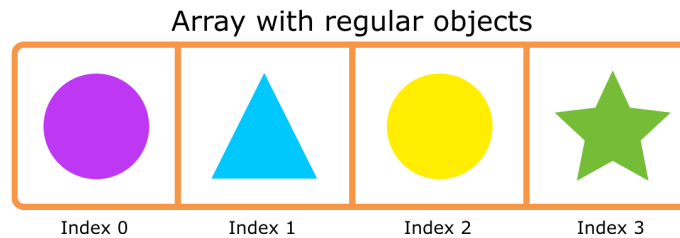
The approach that we've taken here to remember which rows are checked or not works just fine... when there's five rows of data.

But what if you have 100 rows and they all need to be unique? Should you add another 95 “row text” and “row checked” variables to the view controller, as well as that many additional `if` statements? I hope not!

There is a better way: arrays.

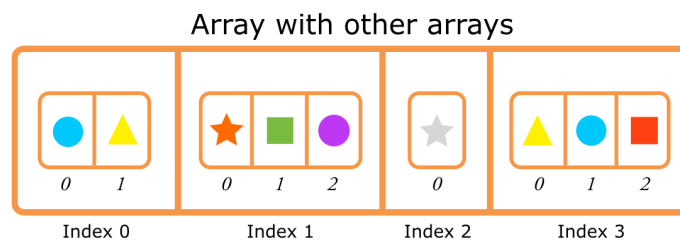
Arrays

An **array** is an ordered list of objects. If you think of a variable as a container of one value (or one object) then an array is a container for multiple objects.



Arrays are ordered lists containing multiple objects

Of course, the array itself is also an object (named `Array`) that you can put into a variable. And because arrays are objects, arrays can contain other arrays.



Arrays can also include other arrays

The objects inside an array are indexed by numbers, starting at 0 as usual. To ask the array for the first object, you write `array[0]`. The second object is at `array[1]`, and so on.

The array is *ordered*, meaning that the order of the objects it contains matters. The object at index 0 always comes before the object at index 1.

Note: An array is a *collection* object. There are several other collection objects and they all organize their objects in a different fashion. Dictionary, for example, contains *key-value pairs*, just like a real dictionary contains a list of words and a description for each of those words. You'll use some of these other collection types in later chapters.

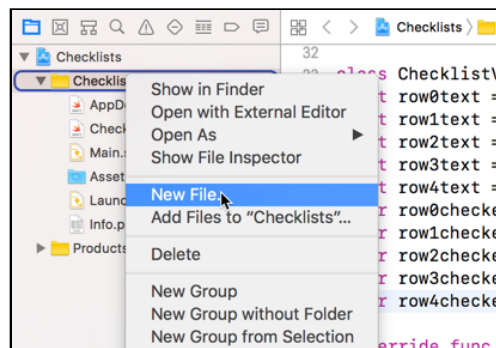
The organization of an array is very similar to the rows for a table – they are both lists of objects in a particular order – so it makes sense to put your data model’s rows into an array.

Arrays store one object per index, but your rows currently consist of two separate pieces of data: the text and the checked state. It would be easier if you made a single object for each row, because then the row number from the table simply becomes the index in the array.

The second iteration

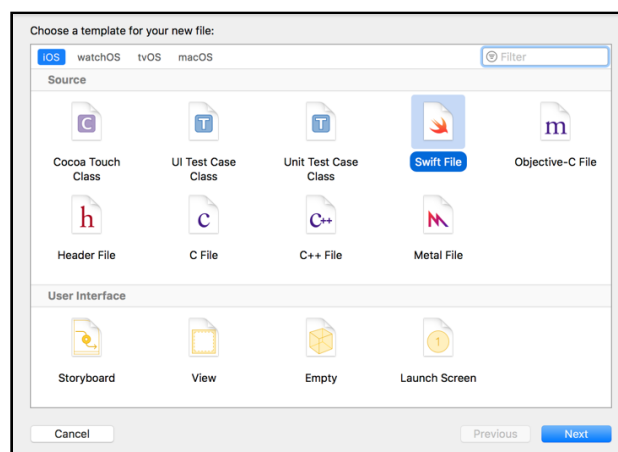
Let’s combine the text and checkmark state into a new object of your own!

► Select the **Checklists** group in the project navigator and right click. Choose **New File...** from the popup menu:



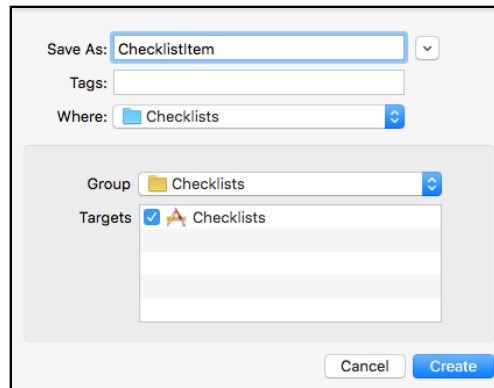
Adding a new file to the project

Under the **Source** section choose **Swift File**:



Choosing the Swift File class template

Click **Next** to continue. Save the new file as **ChecklistItem** (you don't really need to add the **.swift** file extension since it will be automatically added for you).



Saving the new Swift file

Press **Create** to add the new file to the project.

► Add the following to the new **ChecklistItem.swift** file, below the `import` line:

```
class ChecklistItem {  
    var text = ""  
    var checked = false  
}
```

What you see here is the absolute minimum amount of code you need in order to make a new object. The `class` keyword names the object and the two lines with `var` add data items (instance variables) to it.

The `text` property will store the description of the checklist item (the text that will appear in the table view cell's label) and the `checked` property determines whether the cell gets a checkmark or not.

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 **ChecklistItem.swift** for now. The `CheckListItem` object currently only serves to combine the `text` and the `checked` variables into one object. Later you'll do more with it.

Before you try using an array, let's simply replace the `String` and `Bool` instance variables in the view controller with these new `CheckListItem` objects to see how that approach would work.

► In **ChecklistViewController.swift**, remove the old properties (both the `let` and `var` values) and replace them with `CheckListItem` objects:

```
class ChecklistViewController: UITableViewController {  
    var row0item: CheckListItem  
    var row1item: CheckListItem  
    var row2item: CheckListItem  
    var row3item: CheckListItem  
    var row4item: CheckListItem
```

These replace the `row0text`, `row0checked`, etc. instance variables.

Because some methods in the view controller still refer to these 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 that into Xcode.

Of course, if you're reading the print edition of this book, copying & 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 {  
    label.text = row0item.text  
} else if indexPath.row == 1 {  
    label.text = row1item.text  
} else if indexPath.row == 2 {  
    label.text = row2item.text  
} else if indexPath.row == 3 {  
    label.text = row3item.text  
} else if indexPath.row == 4 {  
    label.text = row4item.text  
}
```

► In `tableView(_:didSelectRowAt:)`, again change the if statement block to:

```
if indexPath.row == 0 {
    row0item.checked = !row0item.checked
} else if indexPath.row == 1 {
    row1item.checked = !row1item.checked
} else if indexPath.row == 2 {
    row2item.checked = !row2item.checked
} else if indexPath.row == 3 {
    row3item.checked = !row3item.checked
} else if indexPath.row == 4 {
    row4item.checked = !row4item.checked
}
```

► And finally, in `configureCheckmark(for:at:)`, change the if block to:

```
if indexPath.row == 0 {
    isChecked = row0item.checked
} else if indexPath.row == 1 {
    isChecked = row1item.checked
} else if indexPath.row == 2 {
    isChecked = row2item.checked
} else if indexPath.row == 3 {
    isChecked = row3item.checked
} else if indexPath.row == 4 {
    isChecked = row4item.checked
}
```

Basically, all of the above changes do one thing - instead of using the separate `row0text` and `row0checked` variables, you now use `row0item.text` and `row0item.checked`.

That takes care of all of the errors except for one. Xcode complains that “Class `ChecklistViewController` has no initializers.” This was not a problem before, so what has gone wrong?

Initialize objects

Previously, you gave the “row text” and “row checked” variables a value when you declared them, like so:

```
let row0text = "Walk the dog"
var row0checked = false
```

With the new `CheckListItem` object you can't do that because a `CheckListItem` consists of more than one value.

Instead you used what's known as a *type annotation* to tell Swift that `row0Item` is an object of type `CheckListItem`:

```
var row0item: CheckListItem
```

But at this point `row0item` doesn't have a value yet, it's just an empty container for a `CheckListItem` object.

And that's a problem: in Swift programs, all variables should always have an explicit value – the containers can never be undefined.

If you can't give the variable a value right away when you declare it, then you have to give it a value inside an *initializer* method - as the name implies, an initializer method, initializes (or sets up) the object when it first comes into existence.

► Add the following to **ChecklistViewController.swift**. The initializer is a special type of method (which is why it doesn't start with the word `func`). It is customary to place it near the top of the file, just below the instance variables.

```
required init?(coder aDecoder: NSCoder) {
    row0item = CheckListItem()
    row0item.text = "Walk the dog"
    row0item.checked = false

    row1item = CheckListItem()
    row1item.text = "Brush my teeth"
    row1item.checked = true

    row2item = CheckListItem()
    row2item.text = "Learn iOS development"
    row2item.checked = true

    row3item = CheckListItem()
    row3item.text = "Soccer practice"
    row3item.checked = false

    row4item = CheckListItem()
    row4item.text = "Eat ice cream"
    row4item.checked = true

    super.init(coder: aDecoder)
}
```

Every object in Swift has an `init` method, or initializer. Some objects even have more than one initializer.

The `init` method is called by Swift when the object comes into existence.

For the view controller, that happens when it is loaded from the storyboard during app startup. At that point, its `init?(coder)` method is called.

That makes `init?(coder)` a great place for putting values into any variables that still need them (soon you'll learn more about what the "coder" parameter is for).

Inside `init?(coder)`, you first create a new `CheckListItem` object:

```
row0item = CheckListItem()
```

And then set its properties:

```
row0item.text = "Walk the dog"  
row0item.checked = false
```

You repeat this for the other four rows. Each row gets its own `CheckListItem` object that you store in its own instance variable.

This is essentially doing the same thing as before, except that this time the text and checked variables are not separate instance variables of the view controller but properties of a `CheckListItem` object.

► Run the app just to make sure that everything still works.

Putting the text and checked properties into their own `CheckListItem` object already improved the code, but it is still a bit unwieldy.

With the current approach, you need to keep around a `CheckListItem` 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 **ChecklistViewController.swift**, remove all the instance variables and replace them with a single array variable named `items`:

```
class ChecklistViewController: UITableViewController {  
    var items: [CheckListItem]
```

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 `CheckListItem`. Those square brackets indicate that the variable is going to be an array containing `CheckListItem` objects.

► Make the following changes to `init?(coder:)`:

```
required init?(coder aDecoder: NSCoder) {  
    items = [CheckListItem]()           // add this line  
  
    let row0item = CheckListItem()      // let  
    row0item.text = "Walk the dog"  
    row0item.checked = false  
    items.append(row0item)              // add this line
```



```
let row1item = ChecklistItem()           // let
row1item.text = "Brush my teeth"
row1item.checked = true
items.append(row1item)                   // add this line

let row2item = ChecklistItem()           // let
row2item.text = "Learn iOS development"
row2item.checked = true
items.append(row2item)                   // add this line

let row3item = ChecklistItem()           // let
row3item.text = "Soccer practice"
row3item.checked = false
items.append(row3item)                   // add this line

let row4item = ChecklistItem()           // let
row4item.text = "Eat ice cream"
row4item.checked = true
items.append(row4item)                   // add this line

super.init(coder: aDecoder)
}
```

This is not so different from before, except that you first create – or *instantiate* – the array object:

```
items = [ChecklistItem]()
```

You’ve seen that the notation `[ChecklistItem]` means an array of `ChecklistItem` objects. But that is just the data type of the `items` variable; it is not the actual array object yet.

To get the array object you have to construct it first. That is what the parentheses `()` are for: they tell Swift to make the new array object.

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 array-with-ChecklistItems.”

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.

So until you order an actual array-of-`ChecklistItems` object from the factory and put that into `items`, the variable is empty. And empty variables are a big no-no in Swift.

Just to drive this point home:

```
// This declares that items will hold an array of ChecklistItem
// objects but it does not actually create that array.
// At this point, items does not have a value yet.
var items: [CheckListItem]

// This instantiates the array. Now items contains a valid array
// object, but the array has no objects inside it yet.
items = [CheckListItem]()
```

Note: You can simplify the above two lines by combining them. When you declare the items variable, you can also instantiate it by having the declaration be: `var items: [CheckListItem]()`. That is perfectly acceptable. I have separated the two steps out above for the sake of clarity. Feel free to instantiate variables when they are declared if you like - I generally do.

Each time you make a ChecklistItem object, you also add it to the array:

```
// This instantiates a new ChecklistItem object. Notice the ().
let row0item = ChecklistItem()

// Set values for the data items inside the new object.
row0item.text = "Walk the dog"
row0item.checked = false

// This adds the ChecklistItem object to the items array.
items.append(row0item)
```

Notice that you're also using the parentheses here to create each of the individual ChecklistItem objects.

It's also important that `row0item` and the others are now local to the `init` method. They are no longer valid instance variable names (because you removed those earlier). That's why you need to use the `let` keyword; without it, the app won't compile.

At the end of `init?(coder)`, the `items` array contains five ChecklistItem objects. This is your new data model.

Simplify 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 these methods:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
```

```

-> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "CheckListItem",
        for: indexPath)

    let item = items[indexPath.row] // Add this

    let label = cell.viewWithTag(1000) as! UILabel
    // Replace everything after the above line with the following
    label.text = item.text
    configureCheckmark(for: cell, at: indexPath)
    return cell
}

```

```

override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {

    if let cell = tableView.cellForRow(at: indexPath) {
        // Replace everything inside this `if` condition
        // with the following
        let item = items[indexPath.row]
        item.checked = !item.checked

        configureCheckmark(for: cell, at: indexPath)
    }
    tableView.deselectRow(at: indexPath, animated: true)
}

```

```

func configureCheckmark(for cell: UITableViewCell,
    at indexPath: IndexPath) {
    // Replace full method implementation
    let item = items[indexPath.row]

    if item.checked {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }
}

```

That's a lot simpler than what you had before! Each method is now only a handful of lines long.

In each method, you do:

```
let item = items[indexPath.row]
```

This asks the array for the `CheckListItem` object at the index that corresponds to the row number. Once you have that object, you can simply look at its `text` and `checked` properties and do whatever you need to do.

If the user were to add 100 to-do 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 do exactly the same as before but internal structure of the code is way better.

Exercise: Add a few more rows to the table. You should only have to change `init?` (coder) for this to work.

Clean up the code

There are a few more things you can do to improve the source code.

➤ Replace `configureCheckmark(for:at:)` with this one:

```
func configureCheckmark(for cell: UITableViewCell,
    with item: ChecklistItem) {
    if item.checked {
        cell.accessoryType = .checkmark
    } else {
        cell.accessoryType = .none
    }
}
```

Instead of an index-path, you now directly pass the `ChecklistItem` object to the method.

Note that now the full name of the method becomes `configureCheckmark(for:with:)` and that's how you will call it from other places in the app.

Why did you change this method? Previously it received an index-path and then did the following to find the corresponding `ChecklistItem`:

```
let item = items[indexPath.row]
```

But in both `cellForRowAt` and `didSelectRowAt` you already do that. So, it's simpler to pass that `CheckListItem` object directly to `configureCheckmark` instead of making it do the same work twice. Anything that simplifies the code is good.

➤ Also add this new method:

```
func configureText(for cell: UITableViewCell,
                  with item: CheckListItem) {
    let label = cell.viewWithTag(1000) as! UILabel
    label.text = item.text
}
```

This sets the checklist item's text on the cell's label. Previously you did that in `cellForRowAt` but it's clearer to put that in its own method.

➤ Update `tableView(_:cellForRowAt:)` so that it calls these new methods:

```
override func tableView(_ tableView: UITableView,
                       cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "CheckListItem",
        for: indexPath)

    let item = items[indexPath.row]

    configureText(for: cell, with: item)
    configureCheckmark(for: cell, with: item)
    return cell
}
```

➤ Also update `tableView(_:didSelectRowAt:)`:

```
override func tableView(_ tableView: UITableView,
                       didSelectRowAt indexPath: IndexPath) {

    if let cell = tableView.cellForRow(at: indexPath) {
        let item = items[indexPath.row]
        item.toggleChecked()
        configureCheckmark(for: cell, with: item)
    }
    tableView.deselectRow(at: indexPath, animated: true)
}
```

The above calls a new method named `toggleChecked()` on the item object longer instead of modifying the `CheckListItem`'s `checked` property directly.

You will need to add this new method to the `CheckListItem` object since Xcode should already be complaining about the method not being there.

► Open **ChecklistItem.swift** and add the following method (just below the property declarations and before the closing curly bracket):

```
func toggleChecked() {  
    checked = !checked  
}
```

Naturally, your own objects can also have methods. As you can see, this method does exactly what `didSelectRowAt` used to do, except that you’ve added this bit of functionality to `CheckListItem` instead.

A good object-oriented design principle is that you should let objects change their own state as much as possible. Previously, the view controller implemented this toggling behavior but now `CheckListItem` knows how to toggle itself on or off.

► Run the app. It should still work exactly the same as before, but the code is a lot better. You can now have lists with thousands of to-do items, for those especially industrious users.

Clean up that mess!

So what’s the point of making all of these changes if the app still works exactly the same? For one, the code is much cleaner and that helps with avoiding bugs. By using an array you’ve also made the code more flexible. The table view can now handle any number of rows.

You’ll find that when you are programming you are constantly restructuring your code to make it better. It’s impossible to do the whole thing 100% perfect from the get go.

So you write code until it becomes messy and then you clean it up. After a little while it becomes a big mess again and you clean it up again. The process for cleaning up code is called *refactoring* and it’s a cycle that never ends.

There are a lot of programmers who never refactor their code. The result is what we call “spaghetti code” and it’s a horrible mess to maintain.

If you haven’t looked at your code for several months but need to add a new feature or fix a bug, you may need some time to read it through to understand again how everything fits together. This task becomes that much harder when you have spaghetti code.

So, it’s in your own best interest to write code that is as clean as possible.

If you want to check your work, you can find the project files for the current version of the app in the folder **23-The Data Model** in the Source Code folder.

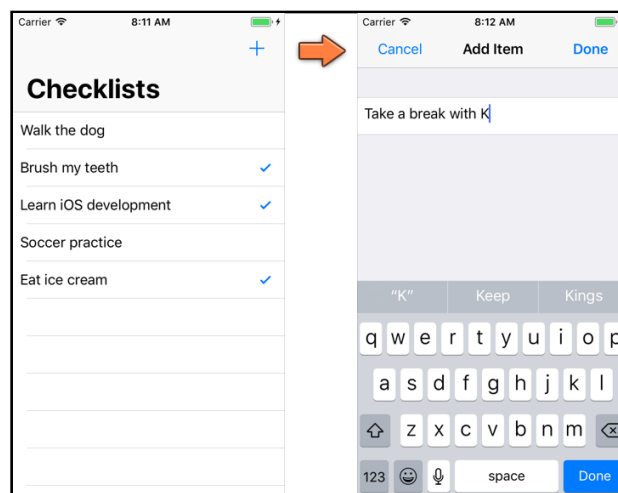
Chapter 24: Navigation Controllers

By Fahim Farook and Matthijs Hollemans

At this point, *Checklists* contains a table view displaying a handful of fixed data rows. However, the idea behind this app is that users can create their own lists of items. Therefore, you need to give the user the ability to add to-do items.

In this chapter you'll expand the app to have a **navigation bar** at the top. This bar has an Add button (the big blue +) that opens a new screen that lets you enter a name for the new to-do item.

When you tap Done, the new item will be added to the list.



The + button in the navigation bar opens the Add Item screen

Presenting a new screen to add items is a common pattern in a lot of apps. Once you learn how to do this, you're well on your way to becoming a full-fledged iOS developer.

This chapter covers the following:

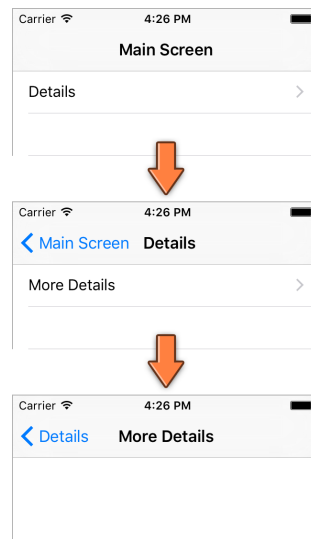
- **Navigation controller:** Add a navigation controller to *Checklists* to allow navigation between screens and add a button to the navigation bar to allow adding new items.
- **Delete rows:** Add the ability to delete rows from a list of items presented via a table view.
- **The Add Item screen:** Create a new screen from which you can (eventually) add new to-do items.

Navigation controller

First, let's add the navigation bar. You may have seen in the Object 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 inside 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 page to another:



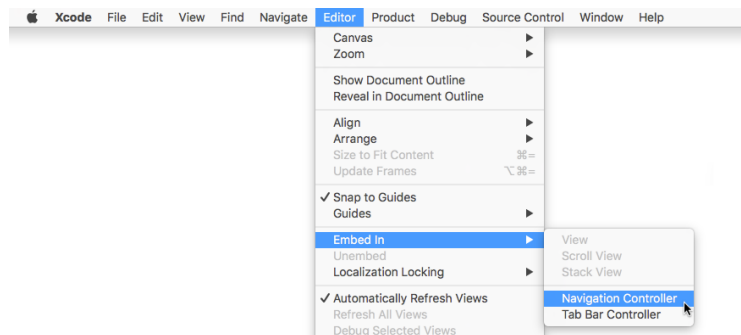
A navigation controller in action

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 of your own on the right.

Add a navigation controller

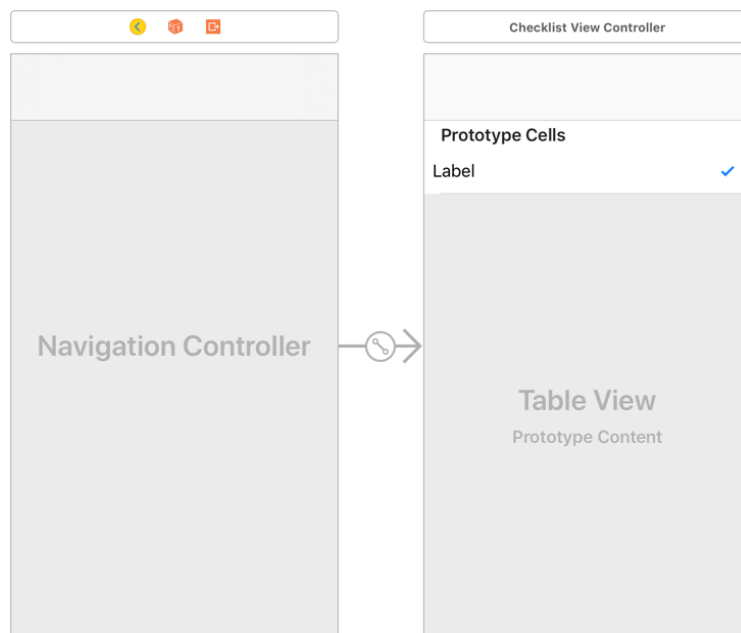
Adding a navigation controller is really easy.

- Open **Main.storyboard** and select the **Checklist View Controller**.
- From the menu bar at the top of the screen, choose **Editor** → **Embed In** → **Navigation Controller**.



Putting the view controller inside a navigation controller

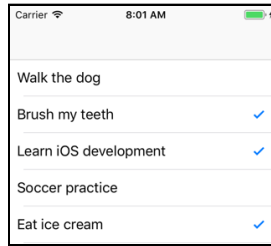
That's it. Interface Builder has now added a new Navigation Controller scene and made a relationship between it and your view controller.



The navigation controller is now linked with your view controller

When the app starts up, the Checklist View Controller is automatically put inside a navigation controller.

- Run the app and try it out.

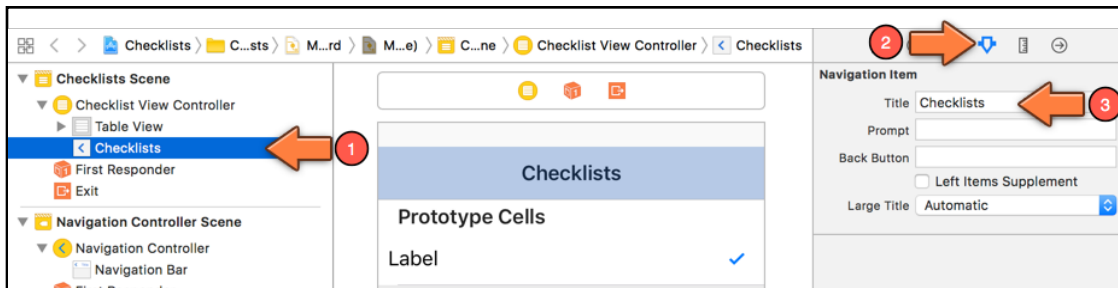


The app now has a navigation bar at the top

The only thing different (visually) is that the app now has a navigation bar at the top.

Set the navigation bar title

► Go back to the storyboard, select **Navigation Item** under Checklist View Controller in the Document Outline, switch to the Attributes Inspector on the right-hand pane, and set the value of **Title** to **Checklists**.



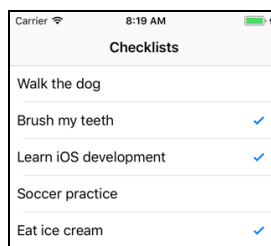
Changing the title in the navigation bar

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.

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.

Run your app and your screen should look something like this:



Navigation bar with title

Display large titles

Before iOS 11, that was all you could do in terms of setting up the navigation bar title. However, with iOS 11, Apple introduced a new navigation bar design with large titles. Large titles are not enabled by default, but you can enable them quite easily with just a checkbox in storyboard, or a single line of code. So, let's do that!

► Switch to **ChecklistViewController.swift** and add the following line to `viewDidLoad`, right after the existing `super.viewDidLoad()` line:

```
navigationController?.navigationBar.prefersLargeTitles = true
```

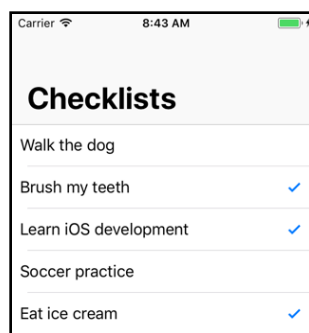
There are a few interesting things in that bit of code but we don't want to get into all of it now. For now, the important things to remember are these:

1. Generally, there is a single navigation controller for a given navigation flow.
2. A single navigation controller could present multiple view controllers as part of its navigation flow.
3. Each view controller in a navigation hierarchy has a reference to the navigation controller which presented it.

Given the above information, the previous code snippet simply uses the view controller's reference to the navigation controller to access the navigation bar for the app. Then, it sets the `prefersLargeTitles` property on the navigation bar to `true`. And it is this property, as the name implies, which enables large titles on iOS 11.

Note: If you wanted to make the same change via storyboard instead of code, you'd select the Navigation Bar under your Navigation Controller in your storyboard and set the **Prefers Large Titles** checkbox in the **Attributes inspector**.

Run your app again. Do you see a difference?



Navigation bar with large title

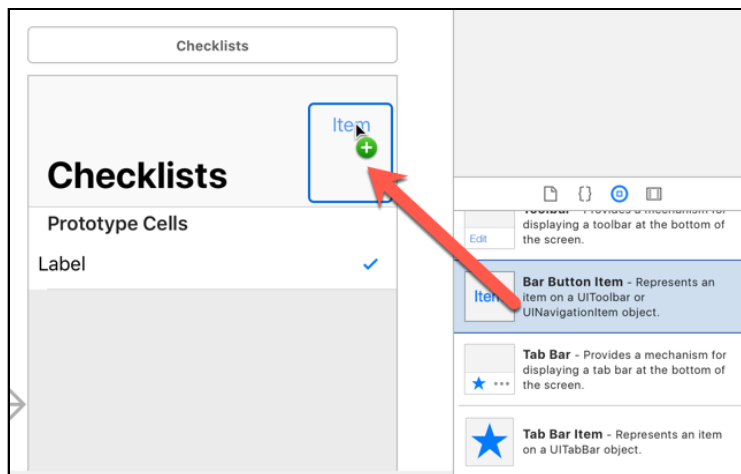
Note: Apple does not recommend using large titles for all of your screens. Rather, their recommendation is to use large titles on your main screen and any other subsequent screens where it might make sense to have a prominent title. You will learn how to turn off large titles for secondary views later on.

Interesting, huh? Of course, you might wonder why there is so much space above the title - that seems like a waste of space, right? That space will be utilized by the navigation items - the back button on the left (if you are in a secondary screen), and any other button you assign to the right.

Add a navigation button to add items

Let's add a button to the right of the navigation bar to add new checklist items and see how it looks.

- Open your main storyboard.
- Go to the Object 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 Checklist View Controller, not the one from the navigation controller!)

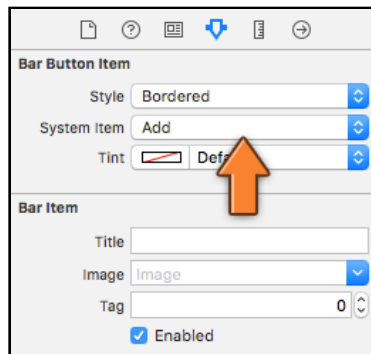


Dragging a Bar Button Item into the navigation bar

Note: You will see large titles on the navigation bar as in the above screenshot only if you enabled large titles via the storyboard. If you enable large titles via code, you will only see the small text title on the navigation bar.

By default, this new button is named “Item” but for this app you want it to have a big + sign.

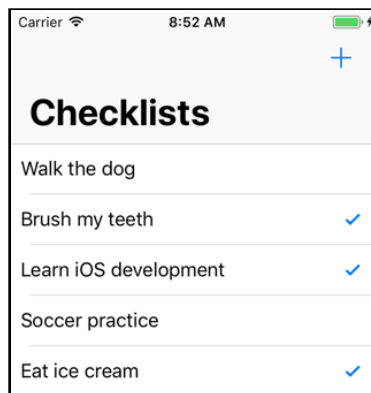
► In the **Attributes inspector** for the bar button item, choose **System Item: Add**.



Bar Button Item attributes

If you look through the list for the System Item dropdown, you'll see a lot of predefined bar button types: Add, Compose, Reply, Camera, and so on. You can use these in your own apps, but be sure to use them only for their intended purpose - you shouldn't use the camera icon on a button that sends an email, for example. Improper use of these icons may lead Apple to reject your app from the App Store. And that sucks.

OK, that gives us a button. If you run the app, it should look like this:



The app with the Add button

Now it looks a little less bare, right? If you are still not happy with the amount of space taken up by large titles, you can always turn off large titles, but do note that when you have a screenful of items and you need to scroll to see more information, the large title will retract into the top navigation bar and give you the "classic"-look navigation bar. So you might want to try this out a bit before deciding to disable it.

Make the navigation button do something

Now, if you tap on your new add button, it doesn't actually do anything. That's because you haven't hooked it up to an action. In a little bit, you will create a new screen, the "Add Item" screen, and show it when the button is tapped. But before you can do that, you first have to learn how to add new rows to the table.

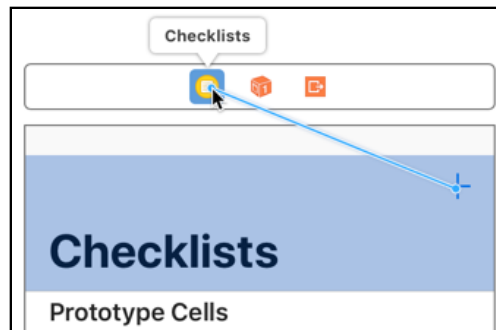
Let's hook up the Add button to an action. You got plenty of exercise with this for *Bull's Eye*, so it should be child's play for you by now.

► Add a new action method to **ChecklistViewController.swift**:

```
@IBAction func addItem() {  
}
```

You're leaving the method empty for the moment, but it needs to be there so you have something to connect the button to.

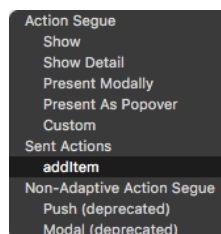
► Open the storyboard and connect the Add button to this action. To do this, **Control-drag** from the + button to the yellow circle in the bar above the view (this circle represents the Checklist View Controller):



Control-drag from Add button to Checklist 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 **addItem** from the popup (under **Sent Actions**):



Connecting to the addItem action

► Let's give `addItem()` something to do. Back in **ChecklistViewController.swift**, add some code to the method as follows:

```
@IBAction func addItem() {  
    let newRowIndex = items.count  
  
    let item = ChecklistItem()  
    item.text = "I am a new row"  
    item.checked = false  
    items.append(item)  
  
    let indexPath = IndexPath(row: newRowIndex, section: 0)  
    let indexPaths = [indexPath]  
    tableView.insertRows(at: indexPaths, with: .automatic)  
}
```

The new code creates a new `ChecklistItem` object and adds it to the data model (the `items` array). You also have to tell the table view, “I’ve inserted a row at this index, please update yourself.”

Let’s review the code section-by-section:

```
let newRowIndex = items.count
```

You need to know what the index of the new row in your array would be. This is necessary in order to properly update the table view later.

When you start the app there are 5 items in the array and 5 rows on the screen. Computers start counting at 0, so the existing rows have indexes 0, 1, 2, 3 and 4. To add the new row to the end of the array, the index for that new row must be 5.

In other words, when you add a row to the end of an array, the index for the new row is always equal to the number of items currently in the array. Let that sink in for a second.

You store the index for the new row in the local constant `newRowIndex`. This can be a constant instead of a variable because it never has to change.

The following few lines should look familiar:

```
let item = ChecklistItem()  
item.text = "I am a new row"  
item.checked = false  
items.append(item)
```

You have seen this code before in `init?(coder)`. It creates a new `ChecklistItem` object and adds it to the end of the array.

The data model now consists of 6 `ChecklistItem` objects inside the `items` array. Note that at this point `newRowIndex` is still 5 even though `items.count` is now 6. That’s why

you read the item count and stored this value in `newRowIndex` *before* you added the new item to the array.

Just adding the new `CheckListItem` object to the data model's array isn't enough though. You also have to tell the table view about this new row so it can add a new cell for that row.

```
let indexPath = IndexPath(row: newRowIndex, section: 0)
```

As you know by now, table views use index-paths to identify rows. So, you first make an `IndexPath` object that points to the new row, using the row number from the `newRowIndex` variable. This index-path object now points to row 5 (in section 0).

The next line creates a new, temporary array holding just the one index-path item:

```
let indexPaths = [indexPath]
```

You use the table view method `insertRows(at:with:)` to tell the table view about the new row. While you only have one inserted row here, as its name implies, this method actually lets you insert multiple rows at the same time, if you wanted to.

So, instead of a single `IndexPath` object, you need to pass an array of index-paths to the method. Fortunately, it is easy to create an array that contains a single index-path object by writing `[indexPath]`. The notation `[]` creates a new `Array` object that contains the objects between the brackets.

Finally, you tell the table view to insert this new row. The `with: .automatic` parameter makes the table view use a nice animation when it inserts the row:

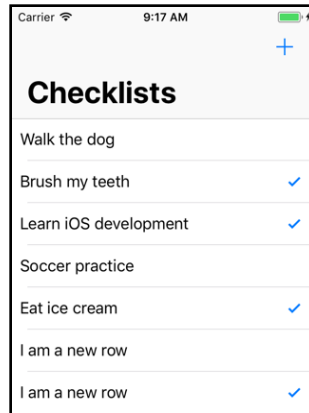
```
tableView.insertRows(at: indexPaths, with: .automatic)
```

To recap, you:

1. Created a new `CheckListItem` object.
2. Added it to the data model.
3. Inserted a new row for it in the table view.

When you call `tableView.insertRows(at:with:)` to insert a new row, the table view makes a cell for this new row by calling your `tableView(_:cellForRowAt:)` data source method. (But it only does this if the new row is actually in the visible portion of the table view.)

► Try it out. You can now add many new rows to the table. You can also tap these new rows to turn their checkmarks on and off again. When you scroll the table up and down, the checkmarks stay with the proper rows.



After adding new rows with the + button

Note: If you were concerned by the change to large titles, also notice how the large title becomes a smaller title (and vice versa) when you scroll up and down.

Remember, the rows always have to be added to both your data model and the table view. When you send the `insertRows(at:with:)` message to the table view, you say: “Hey table, my data model has a bunch of new items added to it.”

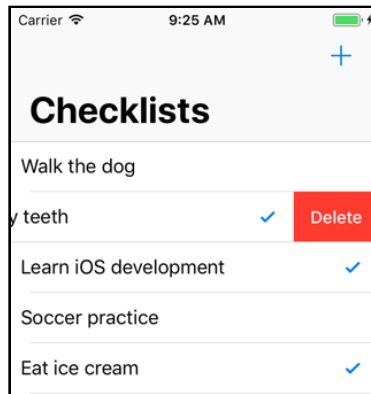
This is important! If you forget to tell the table view about your new items or if you tell the table view there are new items, but you don’t actually add them to your data model, then your app will crash. The data model and the table view always have to be in sync.

Exercise: Give the new items checkmarks by default.

Delete rows

While you’re at it, you might as well give users the ability to delete rows.

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.



Swipe-to-delete in action

Swipe-to-delete

Swipe-to-delete is very easy to implement.

► Add the following method to **ChecklistViewController.swift**. Just to keep things organized, I suggest you put this near the other table view delegate methods.

```
override func tableView(
    _ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    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.

This mirrors what you did in `addItem()`. Again, you make a temporary array with the index-path object and then tell the table view to remove the rows with an animation.

► Run the app to try it out!

Destroying objects

When you call `items.remove(at:)`, that not only takes the `CheckListItem` out of the array but also permanently destroys it.

We'll talk more about this later on, but if there are no more references to an object, it is automatically destroyed. As long as a `CheckListItem` object sits inside an array, that array has a reference to it.

But when you pull that `CheckListItem` out of the array, the reference goes away and the object is destroyed. Or in computer-speak, it is *deallocated*.

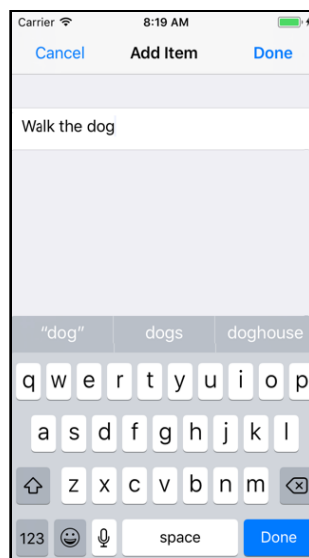
What does it mean for an object to be destroyed? Each object occupies a small section of the computer's memory. When you create an object instance, a chunk of memory is reserved to hold the object's data.

If the object is deallocated, that memory becomes available again and will eventually be occupied by new objects. After it has been deleted, the object does not exist in memory any more and you can no longer use it.

On older versions of iOS, you had to take care of this memory bookkeeping by hand. Fortunately times have changed for the better. Swift uses a mechanism called **Automatic Reference Counting**, or **ARC**, to manage the lifetime of the objects in your app, freeing you from having to worry about that bookkeeping. I like not having to worry about things!

The Add Item screen

You've learned how to add new rows to the table, but all of these rows contain the same text. You will now change the `addItem()` action to open a new screen that lets the user enter their own text for new `CheckListItems`.

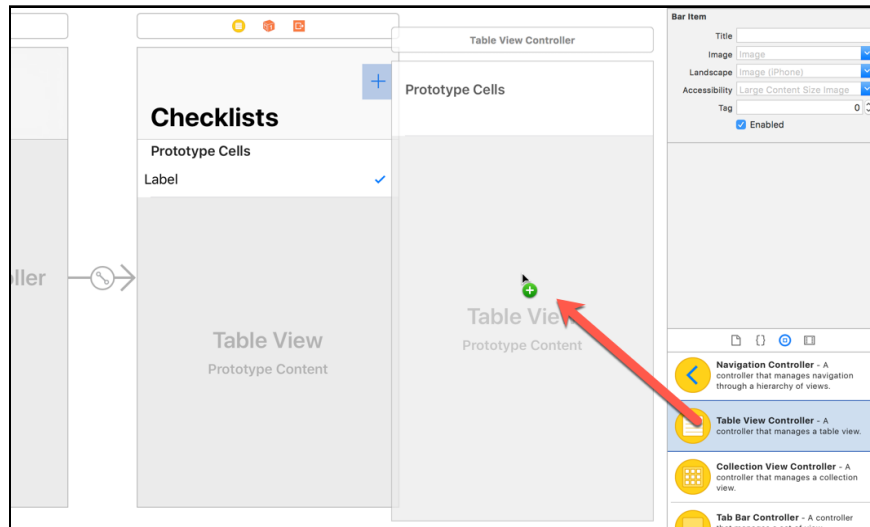


The Add Item screen

Add a new view controller to the storyboard

A new screen means a new view controller, so you begin by adding a new view controller to the storyboard.

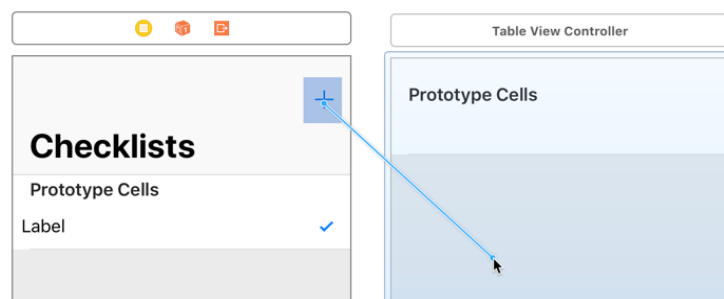
- Go to the Object Library and drag a new **Table View Controller** (not a regular view controller) on to the storyboard canvas.



Dragging a new Table View Controller into the canvas

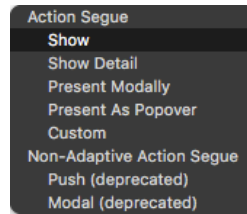
You may need to zoom out to fit everything properly. Right-click on the canvas to get a popup with zoom options, or use the **- 100% +** controls at the bottom of the Interface Builder canvas. (You can also double-click on an empty spot in the canvas to zoom in or out. Or, if you have a Trackpad, simply pinch with two fingers to zoom in or out.)

- With the new view controller in place, select the **Add button** from the Checklist View Controller. **Control-drag** to the new view controller.



Control-drag from the Add button to the new table view controller

Let go of the mouse and a list of options pops up:



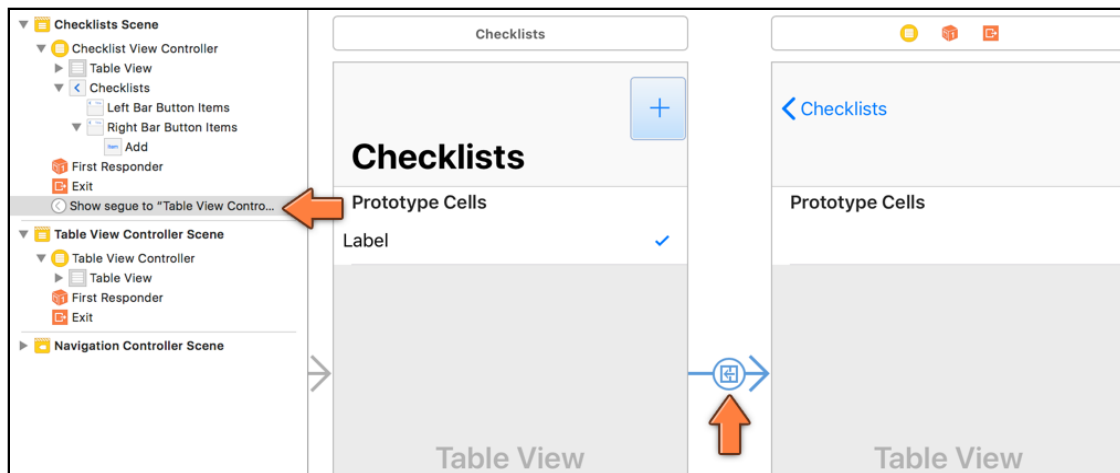
The Action Segue popup

The options in this menu are the different types of connections you can make between the Add button and the new screen.

► Choose **Show** from the menu.

As I mentioned when adding the About screen for *Bull's Eye*, this type of connection is named a segue.

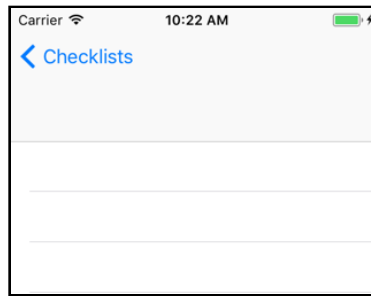
The segue is represented by the arrow between the two view controllers:



A new segue is added between the two view controllers

► Run the app to see what it does.

When you press the Add button, a new empty table view slides in from the right. You can press the back button – the one that says “Checklists” – at the top to go back to the previous screen.



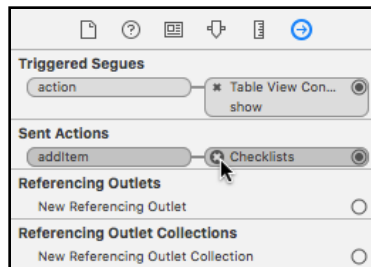
The screen that shows up after you press the Add button

You didn't even have to write much code and you now have yourself a working navigation controller where you can go from one screen to another!

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.

Note that the Add button no longer adds a new row to the table. That connection has been broken and is replaced by the segue. Just in case, you should remove the button's connection with the `addItem` action.

► Select the Add button, go to the **Connections inspector**, and press the small X next to **addItem**.



Removing the addItem action from the Add button

Notice that this inspector also shows the connection with the segue that you've just made (under **Triggered Segues**).

Segue Types

When showing the new view controller above, you opted for a Show segue. But what does it mean? And what do the other options in the Action Segue section of the Interface Builder popup 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*

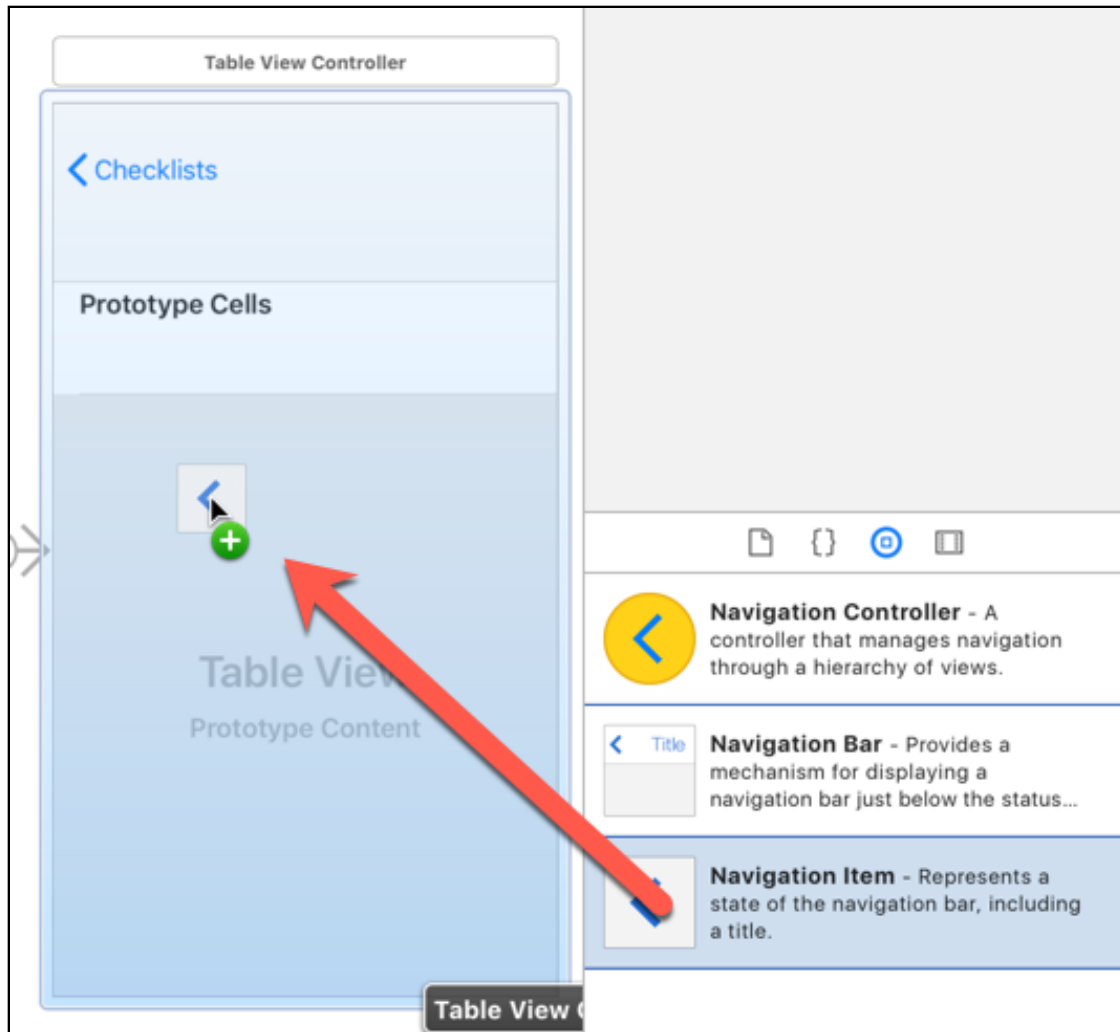
- **Custom:** Allows you to implement your own custom segue and have control over its behavior. (You will learn more about this in a later chapter.)

Customize the navigation bar

So now you have a new table view controller that slides into the screen when you press the Add button. However, this is not quite what you want.

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.

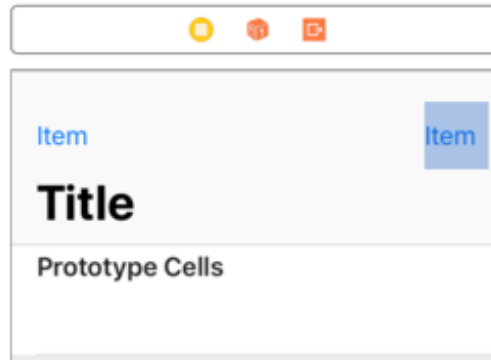
► First, drag a Navigation Item on to the new scene.



Add a navigation item to the view controller

If you check the Document Outline before you drag the Navigation Item on, you will notice that the new table view controller scene does not have a Navigation Item. So, we are not able to customize the storyboard elements for this table view controller - such as the navigation buttons, or the title, without the Navigation Item. Which is the reason for adding one.

► 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.

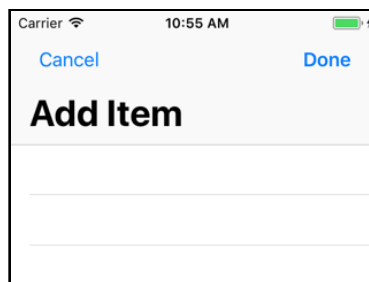


The navigation bar items for the new screen

- 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 user's language.

- Double-click the navigation bar for the new table view controller to edit its title and change it to **Add Item**. (You can also change this via the Attributes inspector as you did before.)
- Run the app, tap the Add button on the main screen, and you'll see that your new screen has Cancel and Done buttons.



The Cancel and Done buttons in the app

The new buttons look good, but (as you would have noticed from the storyboard if you had enabled large titles from the storyboard) the title is huge! If Apple recommends using large titles only on main screens, we probably should change this screen to have smaller titles. But how do we do that?

While some view controller (or table view controller) customizations can be done via storyboard (and this one can too), some require writing some code. Our new view controller does not have a matching source file. So, in the next section we'll create the

source file and add the custom code instead of doing the changes via storyboard just so you know how to do it via code.

Note: If you'd prefer to make the change via storyboard, then simply select the Navigation Item for the new view controller, go to the **Attributes inspector** and select **Never** from the **Large Title** dropdown.

Make your own view controller class

You created a custom view controller in *Bull's Eye* for the About screen. Do you remember how to do it on your own? If not, here are the steps:

- Right-click on the Checklists 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 **AddItemViewController** 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:

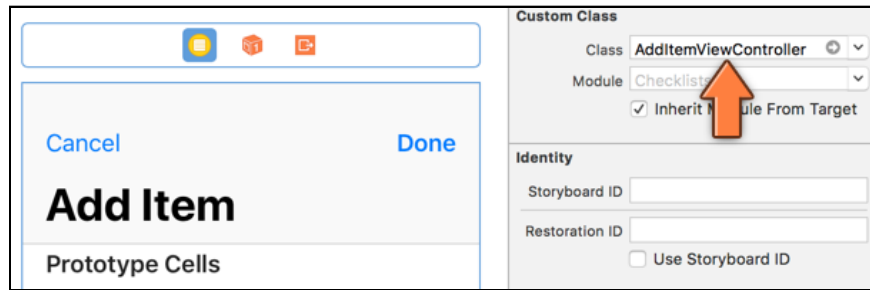
```
import UIKit

class AddItemViewController: 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 `AddItemViewController`. You'll add the rest of the code soon. First, you have to let the storyboard know about this new view controller too.

- In the storyboard, select the Add Item table view controller and go to the **Identity inspector**. Under **Custom Class**, type **AddItemViewController**.

This tells the storyboard that the view controller from this scene is actually your new `AddItemViewController` object.



Changing the class name of the AddItemViewController

Don't forget this step! Without it, the Add Item screen will simply not work.

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.

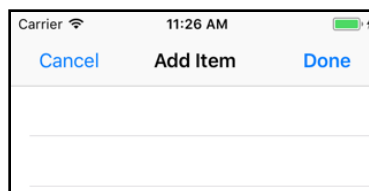
Turn off large titles

Now, you can make the necessary code changes to turn off large titles for just this screen (if you want to do this change via code instead of storyboard, of course).

► Add the following line to the end of `viewDidLoad` in **AddItemViewController.swift**:

```
navigationItem.largeTitleDisplayMode = .never
```

The above code customizes the Navigation Item for the Add Item screen, to never show large titles. Try running the app now.



Large titles begone!

Make the navigation buttons work

Much better, right? But 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 **AddItemViewController.swift**.

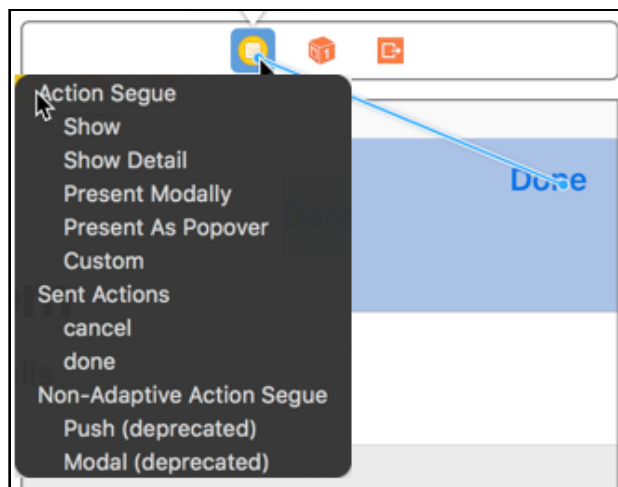
➤ Add these new `cancel()` and `done()` action methods:

```
@IBAction func cancel() {  
    navigationController?.popViewController(animated: true)  
}  
  
@IBAction func done() {  
    navigationController?.popViewController(animated: 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 bar button to the `cancel()` action and the Done bar 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 popup 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 `AddItemViewController` 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 Add Item 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

I've been saying 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 `ChecklistViewController` provides the content for the first screen; the content for the second screen comes from the `AddItemViewController`.

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.

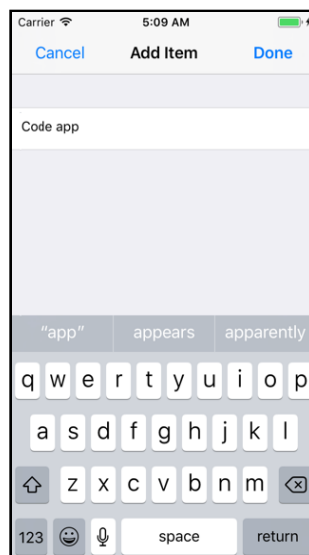
This completes the implementation of the navigation functionality for your app's two screens. If at any point you got stuck, you can refer to the project files for the app from the **24 - Navigation Controllers** folder in the Source Code folder.

Chapter 25: Add Item Screen

By Fahim Farook and Matthijs Hollemans

Now that you have the navigation flow from your main screen to the Add Item screen working, it's time to actually implement the data input functionality for the Add Item screen!

Let's change the look of the Add Item screen. Currently it is an empty table with a navigation bar on top, but I want it to look like this:



What the Add Item screen will look like when you're done

This chapter covers the following:

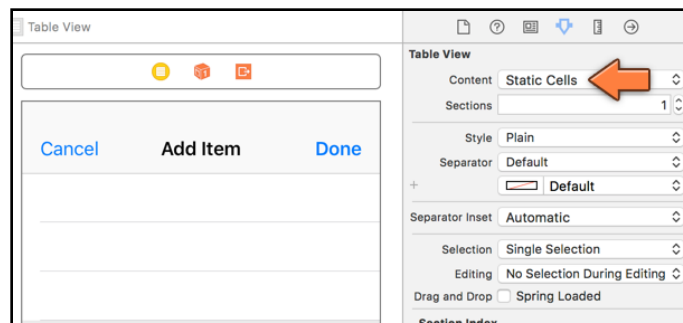
- **Static table cells:** Add a static table view cell to the table to display the text field for data entry.
- **Read from the text field:** Access the contents of the text field.
- **Polish it up:** Improve the look and functionality of the Add Item screen.

Static table cells

First, you need to add a table view cell to handle the data input for the Add Item 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 Add Item scene.
- In the **Attributes inspector**, change the **Content** setting from Dynamic Prototypes to **Static Cells**.

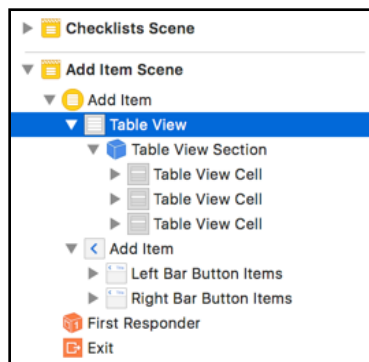


Changing the table view to static cells

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.

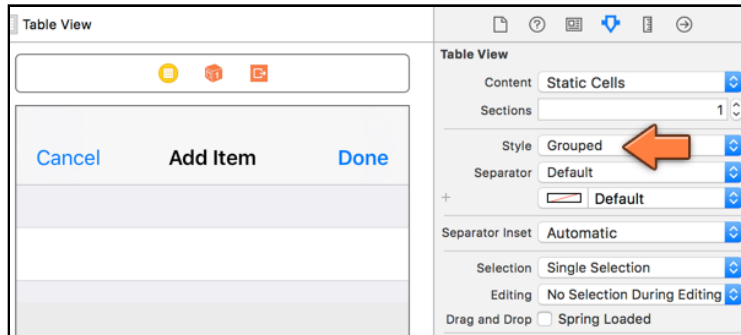
With static cells, you can design the rows directly in the storyboard. For a table with static cells you don't need to provide a data source, and you can hook up the labels and other controls from the cells directly to outlets on the view controller.

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

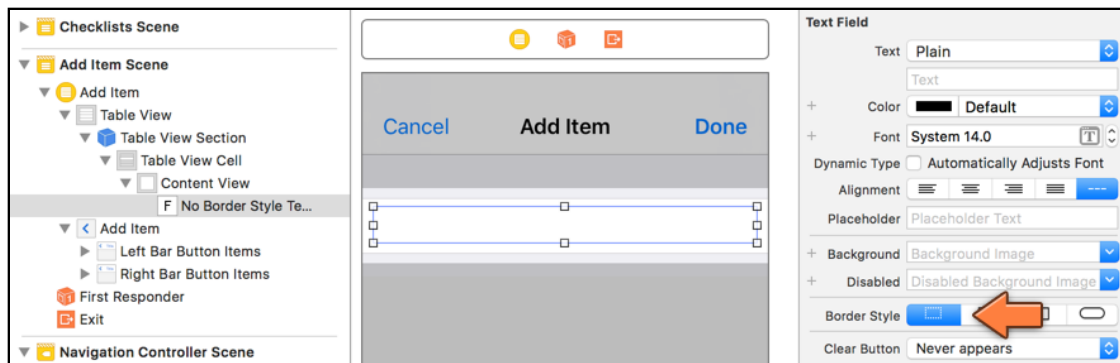
- Click on the bottom two cells and delete them (press the **delete** key on your keyboard). You only need one cell for now.
- Select the Table View again and in the **Attributes inspector** set its **Style** to **Grouped**. That gives us the look we want.



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.

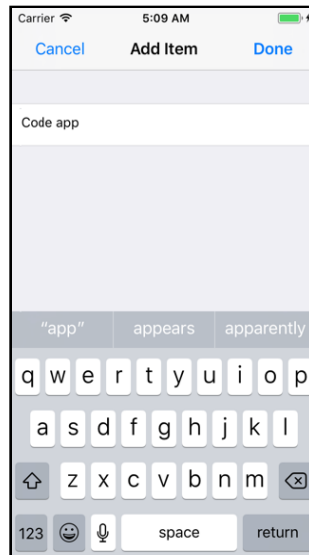
- Drag a **Text Field** object into the cell and size it up nicely.
- In the **Attributes inspector** for the text field, set the **Border Style** to **no border** (select the dotted box):



Adding a text field to the table view cell

- Run the app and press the + button to open the Add Item screen. Tap on the cell and you'll see the keyboard slide in from the bottom of the screen.

Any time you make a text field active, the keyboard automatically appears. You can type into the text field by tapping on the letters. (On the Simulator, you can simply type using your Mac's keyboard.)

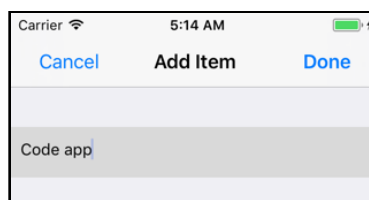


You can now type text into the table view cell

Note: If the keyboard does not appear in the Simulator, press **⌘K** or use the **Hardware** → **Keyboard** → **Toggle Software Keyboard** menu option. You can also use your normal Mac keyboard to type into the text field, even if the on-screen keyboard is not visible. If that doesn't work, also select **Hardware** → **Keyboard** → **Connect Hardware Keyboard** from the menu.

Disable cell selection

Look what happens when you tap just outside 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 **AddItemViewController.swift**:

```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath)
    -> IndexPath? {
    return nil
}
```

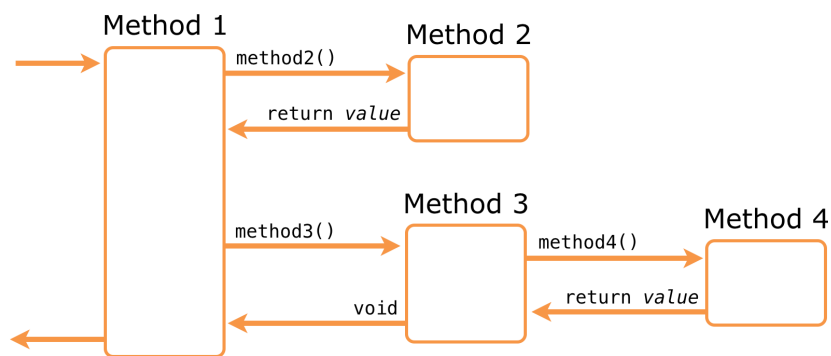
When the user taps on a cell, the table view sends the delegate a `willSelectRowAt` message that says: “Hi delegate, I am about to select this particular row.”

By returning the special value `nil`, the delegate answers: “Sorry, but you’re not allowed to!”

Return to sender

You’ve seen the `return` statement a few times now. You use `return` to send a value from a method back to the method that called it.

Let’s take a more detailed look at what happens.



Methods call other methods and receive values in return.

You cannot just return any value. The value you return must be of the data type that is specified after the `->` arrow that follows the method name.

For example, `tableView(_:numberOfRowsInSection:)` must return an `Int` value:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return 1
}
```

If instead your code was like this:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return "1"
}
```

Then, the compiler would give an error message, as "1" is a string, not an Int. To a human reader they look similar and you can easily understand the intent, but Swift isn't that tolerant. Data types have to match or they just aren't allowed.

Your most recent version of this method looks like this:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return items.count
}
```

That is also a valid return statement because items is an Array and the count property from Array is also of the type Int.

The tableView(_:cellForRowAt:) method is supposed to return a UITableViewCell object:

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(
        withIdentifier: "TheCellIdentifier",
        for: indexPath)

    return cell
}
```

The local constant cell contains a UITableViewCell object, so it's OK to return the value of cell from the method.

The tableView(_:willSelectRowAt:) method is supposed to return an IndexPath object. However, you can also make it return "nil", which means no object.

```
override func tableView(_ tableView: UITableView,
    willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    return nil
}
```

That's what the ? behind IndexPath is for: The question mark tells Swift 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 index-path 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 index-path you were given. This confirms that this row can be selected.
2. Return another index-path in order to select a different row.
3. Return `nil` to prevent the row from being selected, which is what you did.

So remember, you need to use the `return` statement to exit a method that expects to return something. If you forget, then Xcode will give the following error: “Missing return in a function expect to return”.

You've also seen methods that do not return anything:

```
@IBAction func addItem()
```

and:

```
func configureCheckmark(for cell: UITableViewCell,  
                        with item: ChecklistItem)
```

These methods do not have an arrow (`->`) indicating a return value. Such a method does not pass a value back to the caller and therefore does not need a return statement. (You can still use `return` to exit from such methods, but the return statement should not be followed by a value.)

Strictly speaking, even methods without a return type *do* return a value, an *empty tuple*. Think of this as a special object that embodies the concept of “nothing”. (Don't confuse this with *nil*, which is an actual value.)

You sometimes see this written as:

```
func methodThatDoesNotReturnValue() -> ()  
  
func anotherMethodThatDoesNotReturnValue() -> Void
```

The notation for an empty tuple is `()`, so in this context the parentheses mean there is no return value. The term `Void` is a synonym for `()`.

But really, if a method does not return anything it's just as easy to leave out the `->` arrow. Also note that `@IBAction` methods never return a value - this is a rule.

While it's already impossible to select the row, as you've just told the table view you won't allow it, there is one more thing you need to do to prevent the row from going gray. In fact, most of the time, this second change is enough to not show cell selection, even without the code change above.

Table view cells have a selection color property. Even if you make it impossible for a row to be selected, sometimes UIKit still briefly draws the cell gray when you tap it. Therefore, it is best to also disable this selection color.

► In the storyboard, select the table view cell and go to the **Attributes inspector**. Set the **Selection** attribute to **None**.

Now if you run the app, it is impossible to select the row and make it turn gray. Try and prove me wrong! :]

Read from the text field

You have a text field in a table view cell that the user can type into, but how do you read the text that the user has typed?

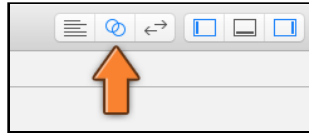
Add an outlet for the text field

When the user taps Done, you need to get that text and somehow put it into a new `CheckListItem` and add it to the list of to-do items. This means the `done()` action needs to be able to refer to 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, I told you to type in the `@IBOutlet` declaration in the source file and make the connection in the storyboard.

I'm going to show you 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!

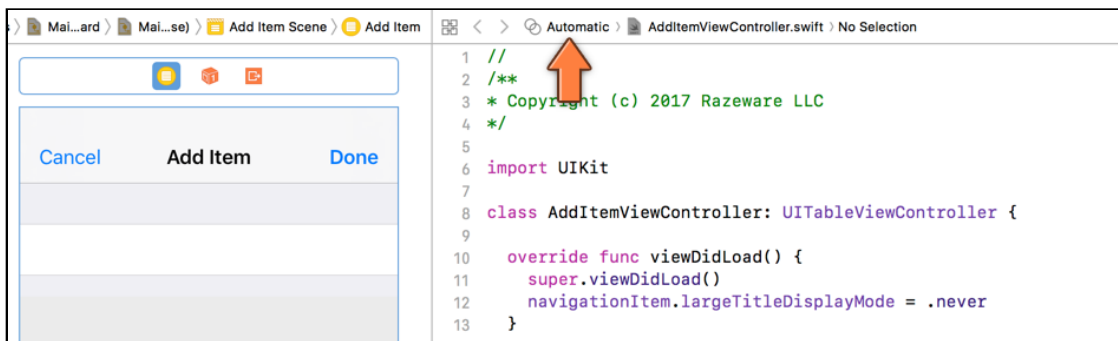
► First, go to the storyboard and select the **Add Item View Controller**. Then open the **Assistant editor** using the toolbar button on the top right. This button looks like two circles:



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 (the bar below the toolbar) it should say **Automatic** and the Assistant editor should be displaying the **AddItemViewController.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 **AddItemViewController.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 popup appears:



The popup 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 UITableViewCell or UIView, 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 let 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?

Read the contents of 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 **AddItemViewController.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, press the + button and type something in the text field. When you press Done, the Add Item 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 my faithful debugging companions :]

Recall that you can print the value of a variable by placing it inside `\(and)` in a string. Here you used `\(textField.text!)` to print out the contents of the text field's `text` property. (I'll explain what the exclamation point is for later.)

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.

Polish it up

Before you write the code to take the text and insert it as a new item into the items list, let's improve the design and workings of the Add Item screen a little.

Give the text field focus on screen opening

For instance, it would be nice if you didn't have to tap on the text field in order to bring up the keyboard. It would be more convenient if the keyboard automatically showed up on the screen opening.

► To accomplish this, add a new method to **AddItemViewController.swift**:

```
override func viewWillAppear(_ animated: Bool) {  
    super.viewWillAppear(animated)  
    textField.becomeFirstResponder()  
}
```

The view controller receives the `viewWillAppear()` 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 Add Item 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.)

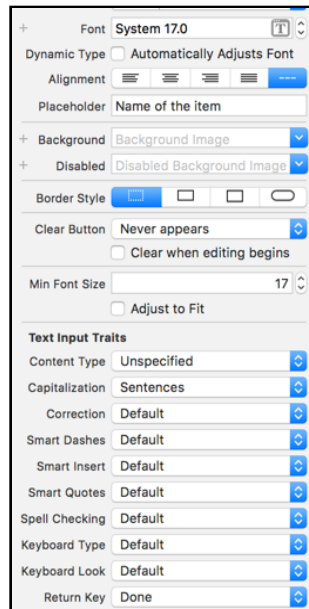
It's often little features like this that make an app a joy to use. Having to tap on the text field before you can start typing gets old really fast. In this fast-paced age, using their mobiles on the go, users don't have the patience for that. Such minor annoyances may be reason enough for users to switch to a competitor's app. I always put a lot of effort into making my apps as frictionless as possible.

Style 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: **Name of the Item**
- 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.

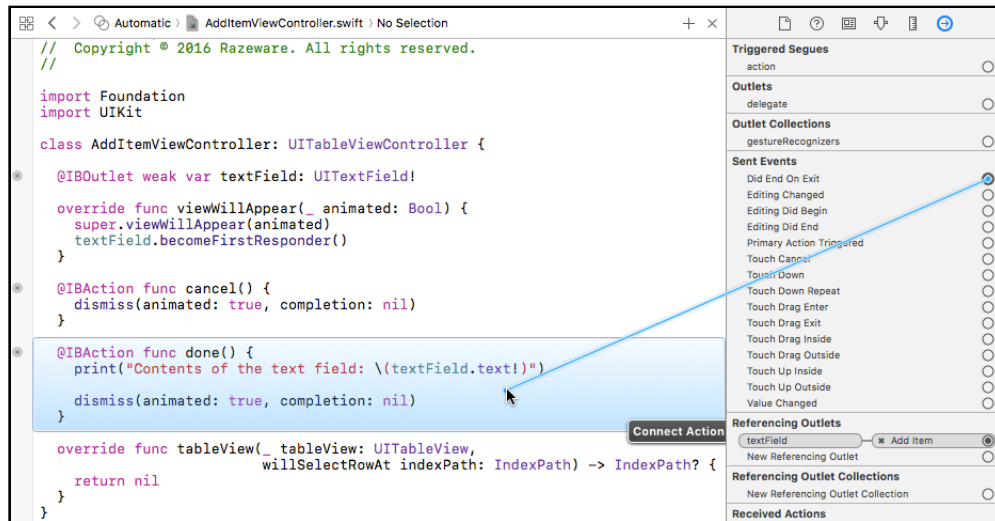
If this were a field that only allowed numbers, for example, you would set the Keyboard Type to Number Pad. If it were an email address field, you'd set it to E-mail Address. For our purposes, the Default keyboard is appropriate.

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.

Handle 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.



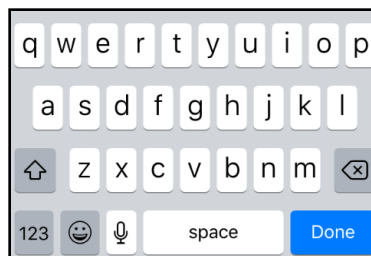
Connecting the text field to the done() action method

To see the connections for the done action, click on the circle in the gutter next to the method name. The popup shows that done() is now connected to both the bar button and the text field:



Viewing the connections for the done() method

► Run the app. 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

Disallow empty input

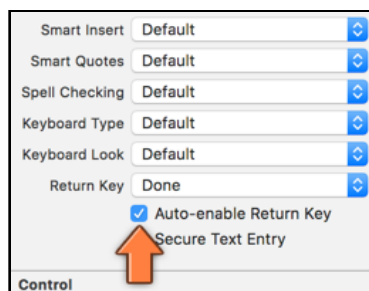
Now that you have user input working, It's always good to validate what the user entered to make sure that the input is acceptable. For instance, what should happen if the user immediately taps the Done button on the Add Item screen without entering any text?

Adding a to-do item to the list that has no description text is not very useful. So, in order 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.

In order 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 AddItemViewController, 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 AddItemViewController 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.

How to become 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 **AddItemViewController.swift**, add `UITextFieldDelegate` to the class declaration:

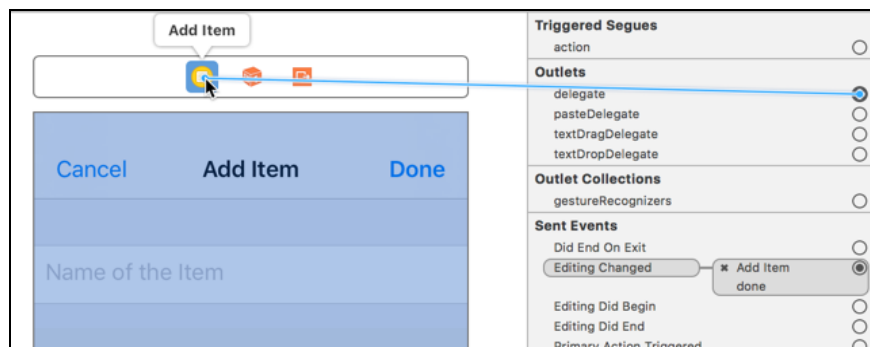
```
class AddItemViewController: UITableViewController, UITextFieldDelegate
```

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. I prefer 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

You also have to add an outlet for the Done bar button item, so you can send it messages from within the view controller in order to enable or disable it.

► Open the **Assistant editor** and make sure **AddItemViewController.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 **AddItemViewController.swift**, at the bottom (before the final curly brace):

```
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 are 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.

Which 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. And 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 thing 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)
```

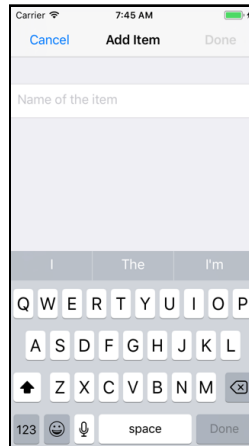
In practice it doesn't really matter which version you use. I prefer the shorter one; that's what the pros do. Just remember that comparison operators such as `==` and `>` always return `true` or `false`, so the extra `if` really isn't necessary.

► 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.

One problem: The Done button is initially enabled when the Add Item screen opens, but there is no text in the text field at that point. So, it really should be disabled. This is simple enough to fix.

► In the storyboard, select the **Done** bar button and go to the **Attributes inspector**. Uncheck the **Enabled** box.

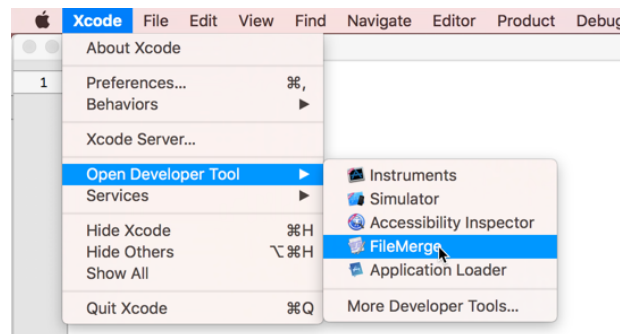
The Done button is now properly disabled when you first enter the Add Item screen:



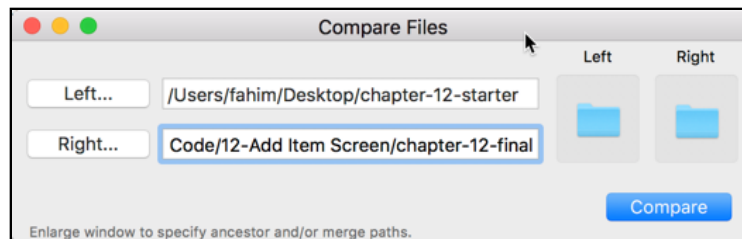
The Done button is not enabled if there is no text

Using FileMerge to compare files

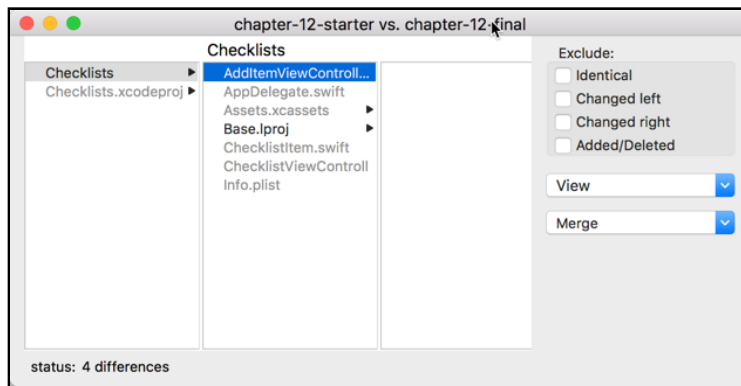
You can compare your own work with my version of the app using the FileMerge tool. Open this tool from the Xcode menu bar, under **Xcode** → **Open Developer Tool** → **FileMerge**:



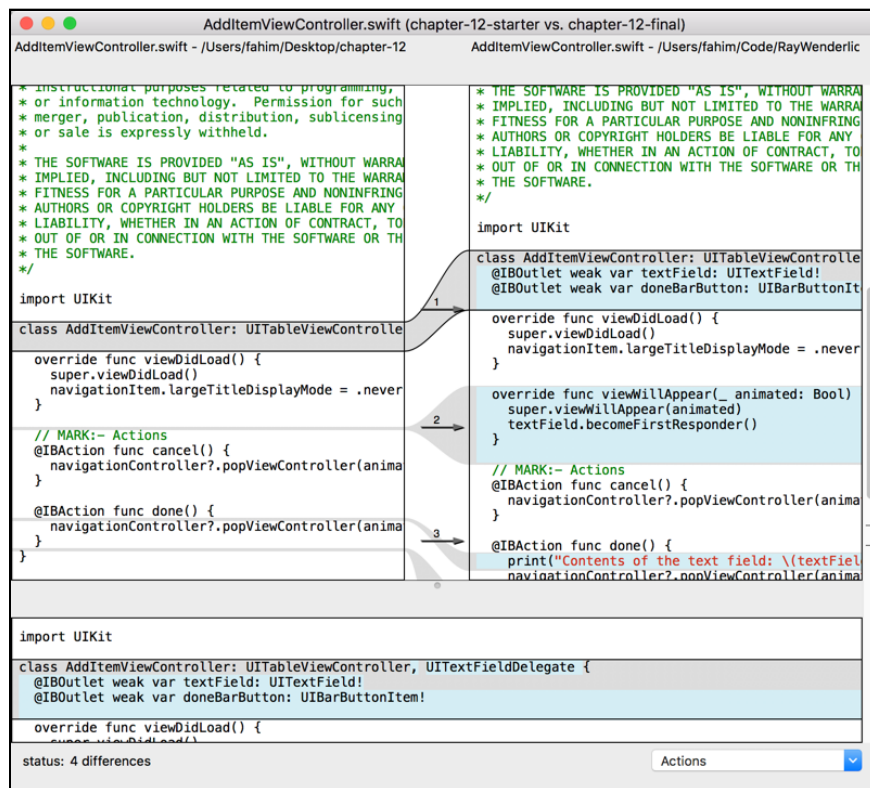
You give FileMerge two files or folders to compare:



After working hard for a few seconds or so, FileMerge tells you what is different:



Double-click on a filename from the list to view the differences between the two files:



FileMerge is a wonderful tool for spotting the differences between two files or even entire folders. I use it all the time!

If something from the book doesn't work as it should, then do a "diff" – that's what you're supposed to call it – between your own files and the ones from the Source Code folder to see if you can find any anomalies.

You can find the project files for the app up to this point under **25 - Add Item Screen** in the Source Code folder.

Chapter 26: Delegates and Protocols

By Fahim Farook and Matthijs Hollemans

You now have an Add Item screen with 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 is empty.

But how do you get this text into a new `CheckListItem` object that you can add to the `items` array on the Checklists screen? That is the topic that this chapter will explore.

Add new ChecklistItems

In order for a new item addition to work, you'll have to get the Add Item screen to notify the Checklist View Controller of the new item addition. This is one of the fundamental tasks that every iOS app needs to do: sending messages from one view controller to another.



Sending a `CheckListItem` object to the screen with the items array

The messy way

Exercise: How would you tackle this problem? The `done()` method needs to create a new `CheckListItem` object with the text from the text field (easy), then add it to the `items` array and the table view in `ChecklistViewController` (not so easy).

Maybe you came up with something like this:

```
class AddItemViewController: UITableViewController, . . . {
    // This variable refers to the other view controller
    var checklistViewController: ChecklistViewController

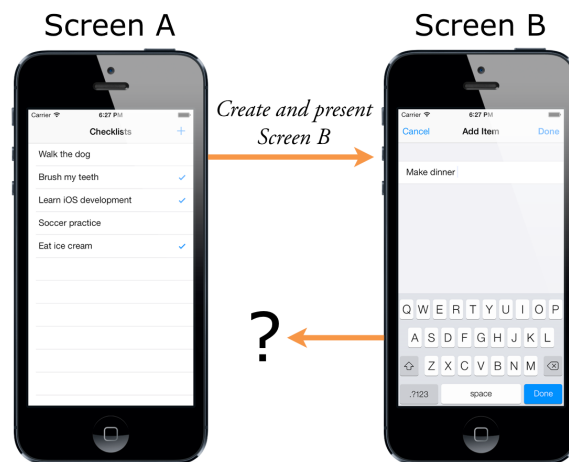
    @IBAction func done() {
        // Create the new checklist item object
        let item = CheckListItem()
        item.text = textField.text!

        // Directly call a method from ChecklistViewController
        checklistViewController.add(item)
    }
}
```

In this scenario, `AddItemViewController` has a variable that refers to the `ChecklistViewController`, and `done()` calls its `add()` method with the new `CheckListItem` object.

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.



Screen A knows all about screen B, but B knows nothing of A

Giving `AddItemViewController` a direct reference to `ChecklistViewController` prevents you from opening the Add Item screen from somewhere else in the app. It can only ever talk back to `ChecklistViewController`. That's a big disadvantage.

You won't actually need to do this in *Checklists*, but in many apps it's common for one screen to be accessible from multiple places. For example, a login screen that appears after the user has been logged out due to inactivity. Or, a details screen that shows more information about a tapped item, no matter where that item is located in the app (you'll see an example of this in the next app).

Therefore, it's best if `AddItemViewController` doesn't know anything about `ChecklistViewController`.

But if that's the case, then how can you make the two communicate?

The solution is to make your own *delegate*.

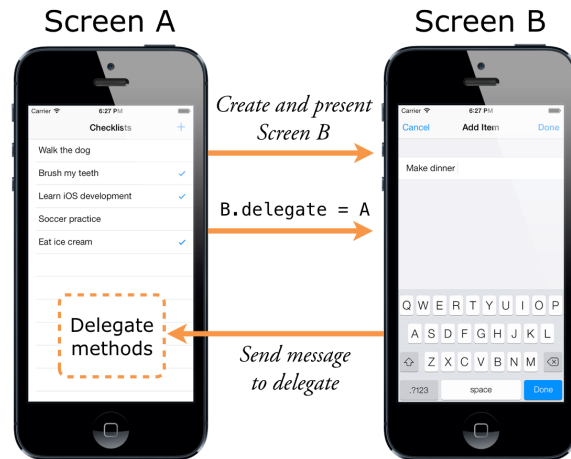
The delegate way

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; and the app also has something named the `AppDelegate` (see the project navigator).

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.

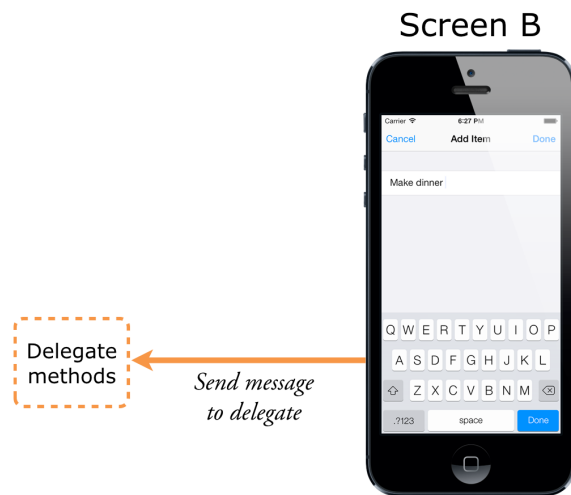
The solution is to make screen A the delegate of screen B, so that B can send its messages to A whenever it needs to.



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 just knows that *some* object is its delegate, but 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 considered good software design practice.



This is what Screen B sees: only the delegate part, not the rest of screen A

You will use the delegate pattern to let the `AddItemViewController` send notifications back to the `ChecklistViewController`, 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 **AddItemViewController.swift**, add this in after the import line (but before the `class` line - it is not part of the `AddItemViewController` object):

```
protocol AddItemViewControllerDelegate: class {  
    func addItemViewControllerDidCancel(  
        _ controller: AddItemViewController)  
    func addItemViewController(  
        _ controller: AddItemViewController,  
        didFinishAdding item: ChecklistItem)  
}
```

This defines the `AddItemViewControllerDelegate` protocol. You should recognize the lines inside the `protocol { ... }` block as method declarations, but unlike the previous methods you've seen, these 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, in this case the `Add Item View Controller`, and any screens that wish to use it.

Protocols

In Swift, a *protocol* doesn't have anything to do with computer networks or meeting royalty. It is simply a name for a group of methods.

A protocol normally doesn't 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 we don't need to get into right now :))

The two methods listed in the `AddItemViewControllerDelegate` protocol are:

- `addItemViewControllerDidCancel(_:)`
- `addItemViewController(_:didFinishAdding:)`

Delegates often have very long method names!

The first method is for when the user presses `Cancel`, the second is for when they press `Done`. In the latter case, the `didFinishAdding` parameter passes along the new `ChecklistItem` object.

To make the `ChecklistViewController` conform to this protocol, it must provide implementations for these two methods. From then on, you can refer to `ChecklistViewController` using the protocol name, instead of the class name.

(If you’ve programmed in other languages before, you may recognize protocols as being very similar to “interfaces”.)

In `AddItemViewController`, you can use the following to refer back to `ChecklistViewController`:

```
var delegate: AddItemViewControllerDelegate
```

The variable `delegate` is nothing more than a reference to *some* object that implements the methods of the `AddItemViewControllerDelegate` protocol. You can send messages to the object referenced by the `delegate` variable without knowing what kind of object it really is.

Of course, *you* know the object referenced by `delegate` is the `ChecklistViewController`, but `AddItemViewController` doesn’t need to be aware of that. All it sees is some object that implements its delegate protocol.

If you wanted to, you could make some other object implement the protocol and `AddItemViewController` would be perfectly OK with that. That’s the power of delegation: you have removed – or *abstracted* away – the dependency between the `AddItemViewController` 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!

Notify the delegate

You’re not done yet in **`AddItemViewController.swift`**. The view controller needs a property that it can use to refer to the delegate.

➤ Add this inside the `AddItemViewController` class, below the outlets:

```
weak var delegate: AddItemViewControllerDelegate?
```

It looks like a regular instance variable declaration, with two differences: *weak* and the question mark.

Delegates are usually declared as being *weak* – not a statement of their moral character but a way to describe the relationship between the view controller and its delegate. Delegates are also *optional* (the question mark - which you learnt a bit about in the previous chapter).

You’ll learn more about those things in a moment.

► Replace the `cancel()` and `done()` actions with the following:

```
@IBAction func cancel() {
    delegate?.addItemViewControllerDidCancel(self)
}

@IBAction func done() {
    let item = ChecklistItem()
    item.text = textField.text!
    item.checked = false

    delegate?.addItemViewController(self, didFinishAdding: item)
}
```

Let's look at the changes you made. When the user taps the Cancel button, you send the `addItemViewControllerDidCancel(_:)` message back to the delegate.

You do something similar for the Done button, except that the message is `addItemViewController(_:didFinishAdding:)` and you pass along a new `ChecklistItem` object that has the text string from the text field.

Note: It is customary for the delegate methods to have a reference to their owner as the first (or only) parameter.

Doing this is not required, but still a good idea. For example, in the case of table views, it may happen that an object is the delegate or data source for more than one table view. In that case, you 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` object 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, in this case `AddItemViewController`. It's also why all the delegate method names start with `addItemViewController`.

► Run the app and try the Cancel and Done buttons. They no longer work!

I hope you're not too surprised... The Add Item screen now depends on a delegate to make it close, but you haven't told the Add Item screen 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

I mentioned a few times 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. This is not allowed in Swift for normal variables.

The problem with `nil` and `NULL` is that they are a frequent cause of crashing apps. If an app attempts to use a variable that is `nil` when you don't expect it to be `nil`, the app will crash. This is the dreaded “null pointer dereference”.

Swift stops this from happening 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 made optional can have the value `nil`.

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 is a question mark behind the type:

```
weak var delegate: AddItemViewControllerDelegate?
```

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? There are two reasons.

Often, delegates are truly optional; a `UITableView` works fine even if you don't implement any of its delegate methods (but you do need to provide at least some of its data source methods).

More importantly, when `AddItemViewController` is loaded from the storyboard and instantiated, 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 it is only temporary, 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?.addItemViewControllerDidCancel(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 it should never happen that `delegate` is `nil` – that would get users stuck on the Add Item screen. But 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 these 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.

Conform to the delegate protocol

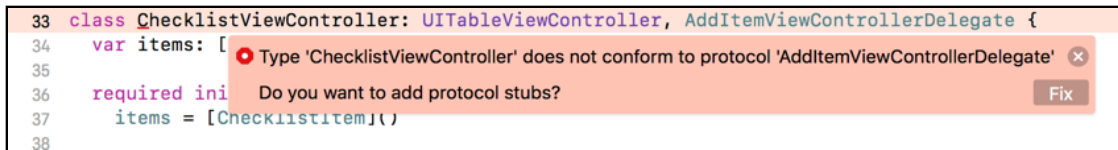
Before you can give `AddItemViewController` its delegate, you first need to make the `ChecklistViewController` suitable to play the role of delegate.

► In **ChecklistViewController.swift**, change the class line to the following (this goes all on one line):

```
class ChecklistViewController: UITableViewController,  
    AddItemViewControllerDelegate {
```

This tells the compiler that `ChecklistViewController` now promises to do the things from the `AddItemViewControllerDelegate` protocol. Or, in programming terminology, that it *conforms* to the `AddItemViewControllerDelegate` protocol.

At this point, Xcode should throw up an error: “Type `ChecklistViewController` does not conform to protocol `AddItemViewControllerDelegate`.”



Xcode warns about not conforming to protocol

That is correct: you still need to add the methods that are listed in AddItemViewControllerDelegate. With the latest Xcode, there is an easy way to get started with fixing this issue - see that "Fix" button? Simply click it :]

Xcode will add in the stubs (the bare minimum code) for the missing methods. You will have to add in the actual implementation for each method, of course.

➤ Add the implementations for the protocol methods to ChecklistViewController:

```

func addItemViewControllerDidCancel(
    controller: AddItemViewController) {
    navigationController?.popViewController(animated:true)
}

func addItemViewController(
    controller: AddItemViewController,
    didFinishAdding item: ChecklistItem) {
    navigationController?.popViewController(animated:true)
}

```

Currently, both methods simply close the Add Item screen. This is what the AddItemViewController used to do in its cancel() and done() actions. You've simply moved that responsibility to the delegate.

The code that puts the new ChecklistItem object into the table view is yet to be added. You'll do that in a moment, but there's something else you need to do first.

Delegates in five easy steps

These are the steps for setting up the delegate pattern between two objects, where object A is the delegate for object B, and object B will send messages back to A. The steps are:

- 1 - Define a delegate protocol for object B.
- 2 - Give object B a delegate optional variable. This variable should be weak.
- 3 - Update object B to send messages to its delegate when something interesting happens, such as the user pressing the Cancel or Done buttons, or when it needs a piece of information. You write `delegate?.methodName(self, . . .)`
- 4 - Make object A conform to the delegate protocol. It should put the name of the protocol in its class line and implement the methods from the protocol.

5 - Tell object B that object A is now its delegate.

You've done steps 1 - 4, so there is just one more thing you need to do - step 5: tell `AddItemViewController` that `ChecklistViewController` is its delegate.

The proper place to do that is in the `prepare(for:sender:)` method, also known as *prepare-for-segue*.

The `prepare(for:sender:)` method is invoked by UIKit when a segue from one screen to another is about to be performed. Recall 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 it is displayed. Usually you'll do this by setting its properties.

► Add this method to **ChecklistViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    // 1
    if segue.identifier == "AddItem" {
        // 2
        let controller = segue.destination
                               as! AddItemViewController
        // 3
        controller.delegate = self
    }
}
```

This is what the above code does, step-by-step:

1. Because there may be more than one segue per view controller, it's a good idea to give each one a unique identifier and to check for that identifier first to make sure you're handling the correct segue. Swift's `==` comparison operator works on not just numbers but also on strings and most other types of objects.
2. The new view controller to be displayed can be found in `segue.destination`, but `destination` is of type `UIViewController` since the new view controller could be any view controller sub-class. So, you *cast* `destination` to `AddItemViewController` to get a reference to an object with the right type. (The `as!` keyword is known as a *type cast* or a *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. The casting works here because `AddItemViewController` is a sub-class of `UIViewController`.)

- Once you have a reference to the `AddItemViewController` object, you set its delegate property to `self` and the connection is complete. This tells `AddItemViewController` that from now on, the object identified as `self` is its delegate. But what is “self” here? Well, since you’re editing **ChecklistViewController.swift**, `self` refers to `ChecklistViewController`.

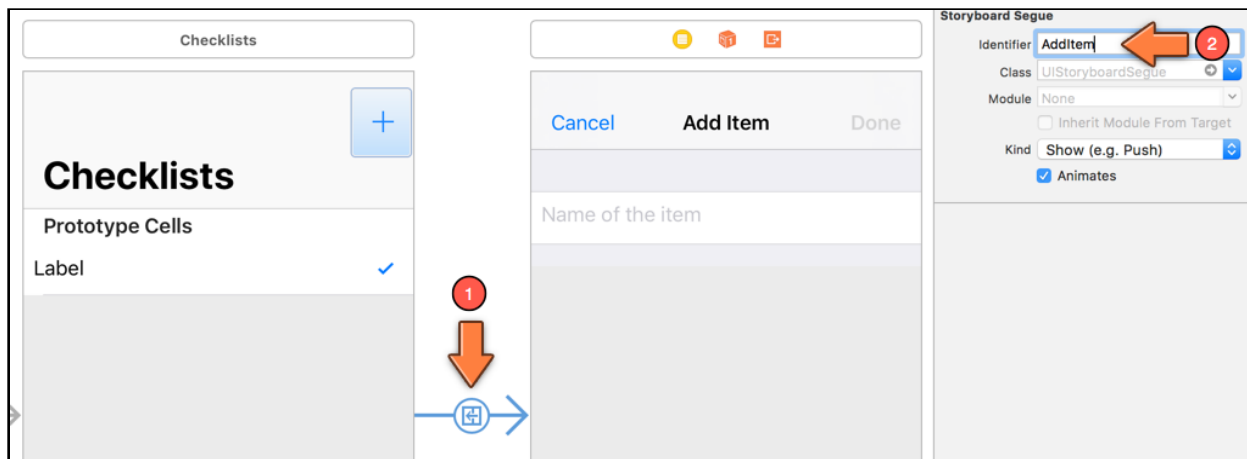
Excellent! `ChecklistViewController` is now the delegate of `AddItemViewController`. It took some work, but you’re almost set now - except for one teensy thing :]

Set the segue identifier

See the segue identifier that is checked in the code above? Where was it set? The answer is, that it wasn't. We need to set the identifier in order for the above code to work.

➤ Open the storyboard and select the segue between the Checklist View Controller and the Add Item View Controller.

➤ In the **Attributes inspector**, type **AddItem** into the **Identifier** field:



Naming the segue between the Checklists scene and the Add Item scene

➤ Run the app to see if it works. (Make sure the storyboard is saved before you press Run, or the app may crash.)

Pressing the + button will perform the segue to the Add Item screen with the Checklists screen set as its delegate.

When you press Cancel or Done, `AddItemViewController` sends a message to its delegate, `ChecklistViewController`. Currently the delegate simply closes the Add Item screen. But now that you know it works, you can make it do more.

Let’s add the new `CheckListItem` to the data model and the table view. Finally!

Add new to-do items

► Change the implementation of the `didFinishAdding` delegate method in **ChecklistViewController.swift** to the following:

```
func addItemViewController(
    _ controller: AddItemViewController,
    didFinishAdding item: ChecklistItem) {
    let newRowIndex = items.count
    items.append(item)

    let indexPath = IndexPath(row: newRowIndex, section: 0)
    let indexPaths = [indexPath]
    tableView.insertRows(at: indexPaths, with: .automatic)
    navigationController?.popViewController(animated: true)
}
```

This is basically the same as what you did in `addItem()` before. In fact, I simply copied the contents of `addItem()` and pasted that into this method with some slight modifications. Compare the two methods and see for yourself.

The only difference is that you no longer create the `CheckListItem` object here; that happens in the `AddItemViewController`. You merely insert this new object into the `items` array.

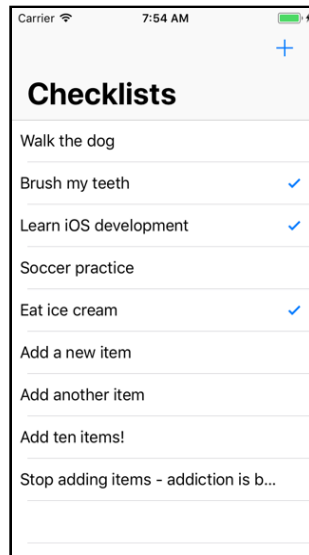
As before, you tell the table view you have a new row for it and then close the Add Items screen.

► Remove `addItem()` from **ChecklistViewController.swift** as you no longer need this method.

Just to make sure, open the storyboard and double-check that the `+` button is no longer connected to the `addItem` action. You should have already removed the connection to the action when you set up the segue to the Add Items scene, but it doesn't hurt to check since bad things happen if buttons are connected to methods that no longer exist...

(You can check this in the Connections inspector for the `+` button, under **Sent Actions**. Nothing should be connected there. Only the segue under Triggered Segues should be present.)

► Run the app and you should be able to add your own items to the list!



You can finally add new items to the to-do list

Weak

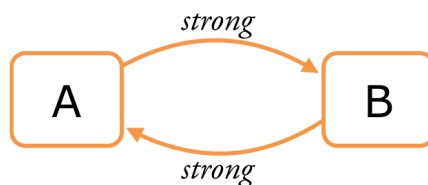
I still owe you an explanation about the `weak` keyword. Relationships between objects can be weak or strong. You use weak relationships to avoid what is known as an *ownership cycle*.

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, an object is destroyed – or *deallocated* – 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 that ought to be destroyed, isn't, 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!

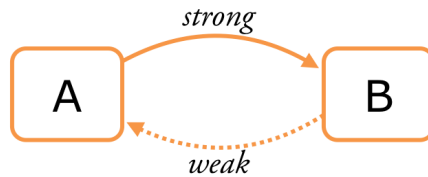
Due to the strong references between them, A owns B and at the same time, B also owns A:



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, delegates are always made weak.

(There is another relationship type, unowned, that is similar to weak and can be used for delegates too. The difference is that weak variables are allowed to become `nil` again. You may forget this right now.)

`@IBOutlet`s are usually also declared 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 **26 - Delegates and Protocols** in the Source Code folder.

Chapter 27: Edit Items

By Fahim Farook and Matthijs Hollemans

Adding new items to the list is a great step forward for the app, but there are usually three things an app needs to do with data:

1. Add new items (which you've tackled).
2. Deleting items (you allow that with swipe-to-delete).
3. Editing existing items (uhh...).

The last is useful when you want to rename an item from your list - we all make typos.

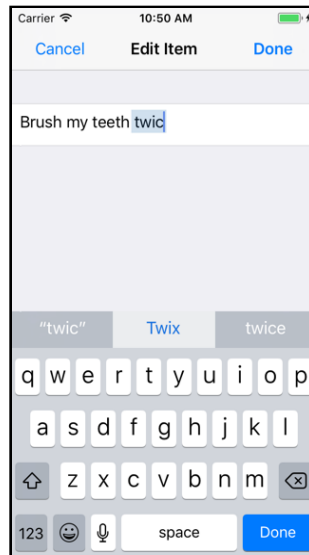
This chapter covers the following:

- **Edit items:** Edit existing to-do items via the app interface.
- **Refactor the code:** Using Xcode's built-in refactoring capability to rename code to be easily identifiable.
- **One more thing:** Fix missed code changes after the code refactoring using the Find navigator.

Edit items

You could make a completely new Edit Item screen but it would work mostly the same as the Add Item screen. The only difference is that it doesn't start out empty - instead, it works with an existing to-do item.

So, let's re-use the Add Item screen and make it capable of editing an existing `CheckListItem` object.



Editing a to-do item

For the edit option, when the user presses Done, you won't have to make a new `CheckListItem` object, instead, you will simply update the text in the existing `CheckListItem`.

You'll also tell the delegate about these changes so that it can update the text label of the corresponding table view cell.

Exercise: What changes would you need to make to the Add Item screen to enable it to edit existing items?

Answer:

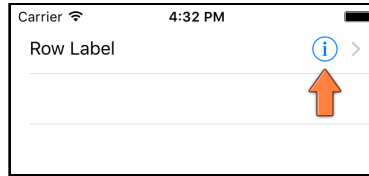
1. The screen title must be changed to **Edit Item**.
2. You must be able to pass it an existing `CheckListItem` object.
3. You have to place the `CheckListItem`'s text into the text field.
4. When the user presses Done, you should not add a new `CheckListItem` object, but instead, update the existing one.

There is a bit of a user interface problem, though... How will the user actually open the Edit Item screen? In many apps that is done by tapping on the item's row, but in *Checklists* that already toggles the checkmark on or off.

To solve this problem, you'll have to revise the UI a little first.

Revise the UI to allow editing

When a row is given two functions, the standard approach is to use a **detail disclosure button** for the secondary task:



The detail disclosure button

Tapping the row itself will still perform the row's main function, in this case, toggling the checkmark. But tapping the disclosure button will open the Edit Item screen.

Note: An alternative approach is taken by Apple's *Reminders* app. There, the checkmark is on the left and tapping only this part of the row will toggle the checkmark. Tapping anywhere else in the row will bring up the Edit screen for that item.

There are also apps that can toggle the whole screen into "Edit mode" and then let you change the text of an item inline. Which solution you choose depends on what works best for your data.

► Go to the table view cell in the storyboard for the Checklists scene and in the **Attributes inspector** set its **Accessory** to **Detail Disclosure**.

Instead of the checkmark, you'll now see a chevron (>) and a blue info button on the cell. This means you'll have to place the checkmark somewhere else.

The new checkmark

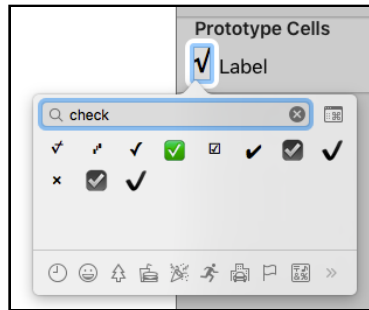
► Drag a new **Label** on to the cell and place it to the left of the text label. Give it the following attributes:

- Text: √ (you can type this with **Alt/Option+V**)
- Font: Helvetica Neue, Bold, size 22
- Tag: 1001

You've given this new label its own tag, so you can easily find it later.

If typing Option-V does not work for you, or you'd prefer a different image, choose **Edit** → **Emoji & Symbols** from the Xcode menu bar. Use the search bar to search for "check"

– or whatever takes your fancy. (Note that not all of these special symbols may actually work on your iPhone.)



The Emoji & Symbols palette

► Resize the text label so that it doesn't overlap the checkmark or the disclosure button. It should be about 215 points wide.

The design of the prototype cell now should look similar to this:



The new design of the prototype cell

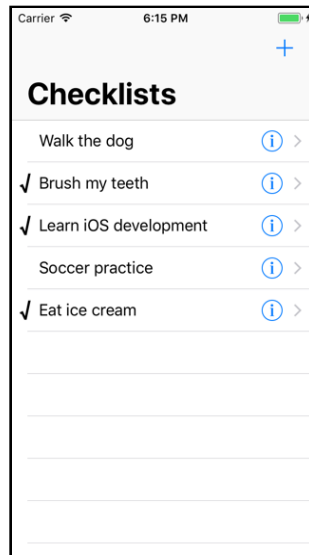
► In **ChecklistViewController.swift**, change `configureCheckmark(for:with:)` to:

```
func configureCheckmark(for cell: UITableViewCell,
                        with item: ChecklistItem) {
    let label = cell.viewWithTag(1001) as! UILabel

    if item.checked {
        label.text = "✓"
    } else {
        label.text = ""
    }
}
```

Instead of setting the cell's `accessoryType` property, this now changes the text in the new label.

► Run the app and you'll see that the checkmark has moved to the left. There is also a blue detail disclosure button on the right. Tapping the row still toggles the checkmark, but tapping the blue button doesn't do anything.

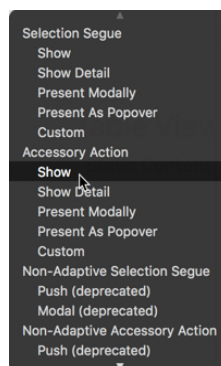


The checkmarks are now on the other side of the cell

The edit screen segue

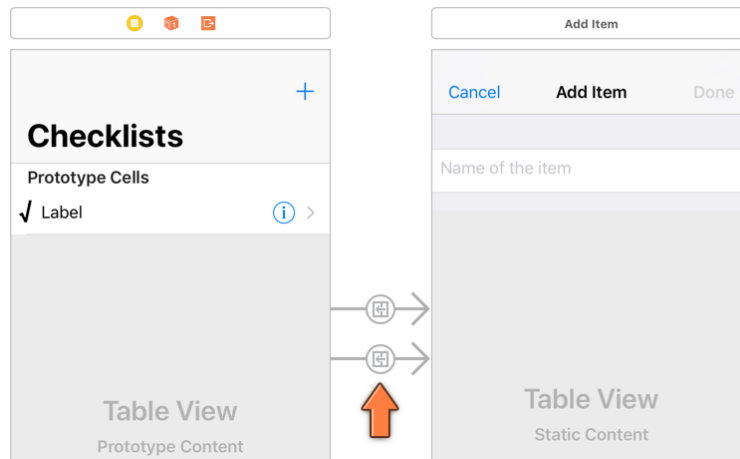
Next, you're going to make the detail disclosure button open the Add/Edit Item screen. This is pretty simple because Interface Builder also allows you to make a segue for a disclosure button.

► Open the storyboard. Select the table view cell for the Checklists scene and **Control-drag** to the Add Item scene to make a segue. From the popup, choose **Show** from the **Accessory Action** section (not from Selection Segue):



Making a segue from the detail disclosure button

There should now be two segues going from the Checklists screen to the navigation controller. One is triggered by the + button, the other by the detail disclosure button from the prototype cell.



Two arrows for two segues

For the app to make a distinction between these two segues, they must have unique identifiers.

► Give this new segue the identifier **EditItem** (in the **Attributes inspector**).

If you run the app now, tapping the blue ⓘ button will also open the Add Item screen. But the Cancel and Done buttons won't work.

Exercise: Can you explain why not?

Answer: You haven't set the delegate yet. Remember that you set the delegate in `prepare(for:sender:)`, but only for when the + button is tapped to perform the "AddItem" segue. You haven't done the same for this new "EditItem" segue.

Before you do that, you should first make the Add Item screen capable of editing existing `CheckListItem` objects.

Update the Add Item screen to handle editing

► Add a new property for a `CheckListItem` object below the other instance variables in **AddItemViewController.swift**:

```
var itemToEdit: CheckListItem?
```

This variable contains the existing `CheckListItem` object that the user will edit. But when adding a new to-do item, `itemToEdit` will be `nil`. That is how the view controller will make the distinction between adding and editing.

Because `itemToEdit` can be `nil`, it needs to be an optional. That explains the question mark.

► Update `viewDidLoad()` in **AddItemViewController.swift** as follows:

```
override func viewDidLoad() {  
    . . .  
    if let item = itemToEdit {  
        title = "Edit Item"  
        textField.text = item.text  
    }  
}
```

Recall that `viewDidLoad()` is called by UIKit when the view controller is loaded from the storyboard, but before it is shown on the screen. That gives you time to put the user interface in order.

In editing mode, when `itemToEdit` is not `nil`, you change the title in the navigation bar to “Edit Item”. You do this by changing the `title` property.

Each view controller has a number of built-in properties and this is one of them. The navigation controller looks for the `title` property and automatically changes the text in the navigation bar.

You also set the text in the text field to the value from the item’s `text` property.

if let

You cannot use optionals like you would regular variables. For example, if `viewDidLoad()` had the following code:

```
textField.text = itemToEdit.text
```

Xcode would complain with the error message, “Value of optional type `CheckListItem?` not unwrapped”.

That’s because `itemToEdit` is the optional version of `CheckListItem`.

In order to use it, you first need to *unwrap* the optional. You do that with the following special syntax:

```
if let temporaryConstant = optionalVariable {  
    // temporaryConstant now contains the unwrapped value  
    // of the optional variable  
}
```

If the optional is not `nil`, then the code inside the `if` statement is performed.

There are a few other ways to read the value of an optional, but using `if let` is the safest: if the optional has no value – i.e. it is `nil` – then the code inside the `if let` block is skipped over.

The new code you added to `viewDidLoad` can also be written like this:

```
if let itemToEdit = itemToEdit {  
    title = "Edit Item"  
    textField.text = itemToEdit.text  
}
```

Looks a bit weird, does it? Why are we assigning the value from `itemToEdit` back again to `itemToEdit`? And how come the compiler doesn't complain about optional unwrapping now if we write the code like that?

The above practice is called *variable shadowing* - you create a "shadow" instance of the `itemToEdit` variable just for the duration of the `if` condition and that shadow instance is an unwrapped instance of the originally optional `itemToEdit` variable.

So, when you refer to `itemToEdit` when assigning text to the text field, you are actually referring to the unwrapped instance of the variable instead of the original optional instance.

This might be a bit confusing if you are new to Swift and optionals. So, whether you use variable shadowing to unwrap optionals, or not, is entirely up to you. Personally, I prefer shadowing because then the code is clear about the variable being referred to in the code at all times since the same variable name is used for both the optional and unwrapped versions.

The `AddItemViewController` is now capable of recognizing when it needs to edit an item. If the `itemToEdit` property is given a `CheckListItem` object, then the screen magically changes into the Edit Item screen.

But where do you set that `itemToEdit` property? In `prepare-for-segue`, of course! That's the ideal place for placing values into the properties of the new screen before it becomes visible.

Set the item to be edited

➤ Change `prepare(for:sender:)` in **`ChecklistViewController.swift`** to the following:

```
override func prepare(for segue: UIStoryboardSegue,  
                      sender: Any?) {  
    if segue.identifier == "AddItem" {  
        . . .  
    } else if segue.identifier == "EditItem" {  
        let controller = segue.destination  
            as! AddItemViewController  
        controller.delegate = self  
    }  
}
```

```
if let indexPath = tableView.indexPath(  
    for: sender as! UITableViewCell) {  
    controller.itemToEdit = items[indexPath.row]  
}  
}
```

As before, you get the `AddItemViewController` via the segue's `destination`.

You also set the view controller's `delegate` property so you're notified when the user taps `Cancel` or `Done`. Nothing new there. This is the same as for the `AddItem` segue.

This is the interesting new bit:

```
if let indexPath = tableView.indexPath(  
    for: sender as! UITableViewCell){  
    controller.itemToEdit = items[indexPath.row]  
}
```

You're in the `prepare(for:sender:)` method, which has a parameter named `sender`. This parameter contains a reference to the control that triggered the segue, in this case, the table view cell whose disclosure button was tapped.

You use that `UITableViewCell` object to find the table view row number by looking up the corresponding index path using `tableView.indexPath(for:)`.

The return type of `indexPath(for:)` is `IndexPath?`, an optional, meaning it can possibly 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 `ChecklistItem` object to edit, and you assign this to `AddItemViewController`'s `itemToEdit` property.

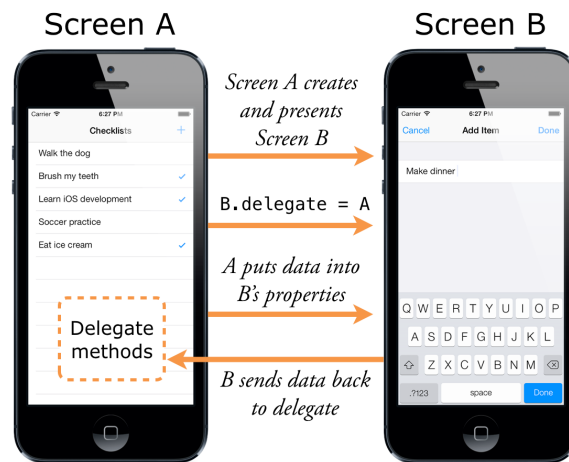
Sending data between view controllers

We've talked about screen B (the Add/Edit Item screen) passing data back to screen A (the Checklists screen) via delegates. But here, you're passing a piece of data the other way around – from screen A to screen B – namely, the `ChecklistItem` to edit.

Data transfer between view controllers works two ways:

1. From A to B. When screen A opens screen B, A can give B the data it needs. You simply make a new instance variable in B's view controller. Screen A then puts an object into this property right before it makes screen B visible, usually in `prepare(for:sender:)`.
2. From B to A. To pass data back from B to A you use a delegate.

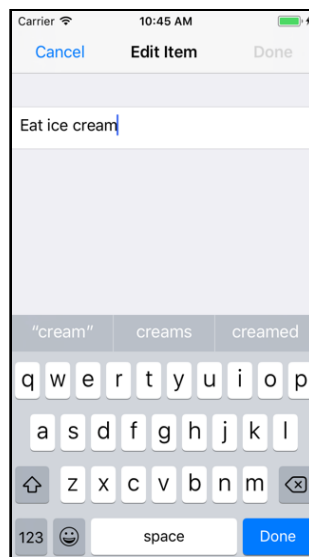
This illustration shows how screen A sends data to screen B by putting it into B's properties, and how screen B sends data back to the delegate:



I hope the flow between view controllers is starting to make sense to you now. You're going to do this sort of thing a few more times in this app, just to make sure you get comfortable with it.

Making iOS apps is all about creating view controllers and sending messages between them, so you want this to become second nature.

► With these steps done, you can now run the app. A tap on the + button opens the Add Item screen as before. But tap the accessory button on an existing row and the screen that opens is named Edit Item. It already contains the to-do item's text:



Editing an item

Enable the Done button for edits

One small problem: the Done button in the navigation bar is initially disabled. This is because you originally set it to be disabled in the storyboard.

► Change `viewDidLoad()` in **AddItemViewController.swift** to fix this:

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

    if let item = itemToEdit {
        title = "Edit Item"
        textField.text = item.text
        doneBarButton.isEnabled = true    // add this line
    }
}
```

When in edit mode, you simply enable the Done button since you are guaranteed to be passed some text for the item.

The problems don't end here, though. Run the app, tap a row to edit it, and press Done. Instead of changing the text on the existing item, a brand new to-do item with the new text is added to the list.

How come? You didn't write the code yet to update the data model! So, the delegate always thinks it needs to add a new row.

To solve this, you will add a new method to the delegate protocol.

Handle edits in the delegate protocol

► Add the following line to the protocol section in **AddItemViewController.swift**:

```
func addItemViewController(_ controller: AddItemViewController,
                           didFinishEditing item: ChecklistItem)
```

The full protocol now looks like this:

```
protocol AddItemViewControllerDelegate: class {
    func addItemViewControllerDidCancel(
        _ controller: AddItemViewController)
    func addItemViewController(
        _ controller: AddItemViewController,
        didFinishAdding item: ChecklistItem)
    func addItemViewController(
        _ controller: AddItemViewController,
        didFinishEditing item: ChecklistItem)
}
```

There is a method that is invoked when the user presses Cancel and two methods for when the user presses Done.

After adding a new item you call `didFinishAdding`, but when editing an existing item, the new `didFinishEditing` method should now be called instead.

By using different methods the delegate (the `ChecklistViewController`) can make a distinction between those two situations.

► In **AddItemViewController.swift**, change the `done()` method to:

```
@IBAction func done() {
    if let itemToEdit = itemToEdit {
        itemToEdit.text = textField.text!
        delegate?.addItemViewController(self,
                                       didFinishEditing: itemToEdit)
    } else {
        let item = ChecklistItem()
        item.text = textField.text!
        item.checked = false
        delegate?.addItemViewController(self, didFinishAdding: item)
    }
}
```

First the code checks whether the `itemToEdit` property contains an object - you should recognize the `if let` syntax for unwrapping an optional.

If the optional is not `nil`, you put the text from the text field into the existing `ChecklistItem` object and then call the new delegate method.

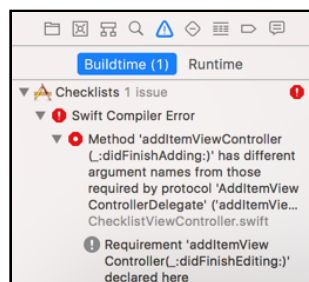
In the case that `itemToEdit` is `nil`, the user is adding a new item and you do the stuff you did before (inside the `else` block).

Implement the new delegate method

► Try to build the app. It won't work.

Xcode says “Build Failed” but there don't seem to be any error messages in **AddItemViewController.swift**. So what went wrong?

You can see all errors and warnings from Xcode in the **Issue navigator**:



Xcode warns about incomplete implementation

The error is apparently in `ChecklistViewController` because it does not implement a method from the protocol. That is not so strange because you just added the new `addItemViewController(_:didFinishEditing:)` method to the delegate protocol. But you did not yet tell the view controller, which plays the role of the delegate, what to do with it.

Note: The exact error message in my version of Xcode is “Method ... has different argument names from those required by protocol ...”. That’s a bit of a strange error message, wouldn’t you say? It doesn’t really describe what’s wrong, just what Swift is confused about.

As you write your own apps, you’ll probably run into other strange or even undecipherable Swift error messages. This should get better in time. The Swift compiler is quite new at the job and still needs to work on its bedside manner.

➤ Add the following to **`ChecklistViewController.swift`** and the compiler error will be history:

```
func addItemViewController(
    _ controller: AddItemViewController,
    didFinishEditing item: ChecklistItem) {
    if let index = items.index(of: item) {
        let indexPath = IndexPath(row: index, section: 0)
        if let cell = tableView.cellForRow(at: indexPath) {
            configureText(for: cell, with: item)
        }
    }
    navigationController?.popViewController(animated: true)
}
```

The `ChecklistItem` object already has the new text – it was put there by `done()` – and the cell for it already exists in the table view. But you do need to update the label for its table view cell.

So, in this new method you look for the cell that corresponds to the `ChecklistItem` object and, using the `configureText(for:with:)` method you wrote earlier, tell it to refresh its label.

The first statement is the most interesting:

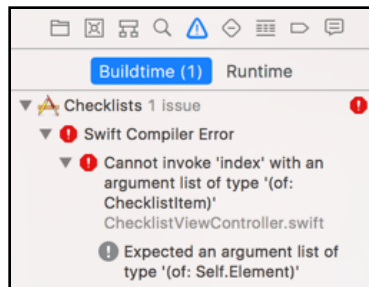
```
if let index = items.index(of: item) {
```

In order to create the `IndexPath` that you need to retrieve the cell, you first need to find the row number for this `ChecklistItem`. The row number is the same as the index of the `ChecklistItem` in the `items` array - you can use the `index(of:)` method to return that index.

Now, it won't happen here, but in theory it's possible that you use `index(of:)` on an object that is not actually in the array. To account for the possibility, `index(of:)` does not 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:)`.

► Try to build the app. Oops, I guess I spoke too soon! Xcode has found another reason to complain: “Cannot invoke `index` with an argument list of type `blah blah blah`”. What does *that* mean?



New Xcode error

This error is displayed because you can't use `index(of:)` on just any array (or collection of objects). An object has to be “equatable” if you are to use `index(of:)` on an array of that object type. This is because `index(of:)` must be able to somehow compare the object that you're looking for against the objects in the array, to see if they are equal.

Your `ChecklistItem` object does not have any functionality for that yet. There are a few ways you can fix this, but we'll go for the easy one.

► In **ChecklistItem.swift**, change the class line to:

```
class ChecklistItem: NSObject {
```

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 provided by iOS, 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 `CheckListItem` on top of `NSObject` is enough to satisfy the “equatable” requirement. In a later chapter, when you learn about saving the checklist items, you would have had to make `CheckListItem` an `NSObject` anyway. So, this is a good solution for this app.

► Run the app again and verify that editing items works now. Excellent!

Refactor the code

At this point, you have an app that can add new items and edit existing items using the combined Add/Edit Item screen. Pretty sweet!

Given the recent changes, I don't think the name `AddItemViewController` is appropriate anymore as this screen is now used to both add and edit items.

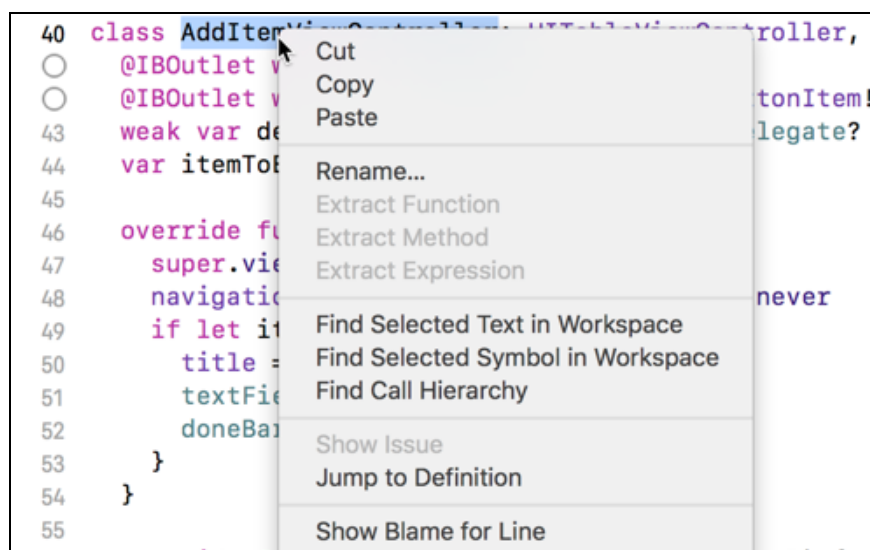
I suggest you rename it to `ItemDetailViewController`.

Rename the view controller

Most IDEs (or Integrated **D**evelopment **E**nvironments) such as Xcode have a feature named *refactoring*, which allows you to change the name of a class, method, or variable through the entire project safely. Unfortunately, the refactoring functionality in Xcode did not work correctly for several years with Swift source files :[

The good news is that as of Xcode 9, the refactoring functionality in Xcode has not only been restored for Swift files, but it has been re-written from the ground up to work for most of the source code types you would generally work on in Xcode!

Yes, I hear you saying, "Enough of the sales pitch, show me how to refactor!" There are a couple of ways to access the refactor functionality, but the easiest is to simply **right-click** (or, **Control-click**) on any class name, method, or variable. You'll get a menu similar to this:

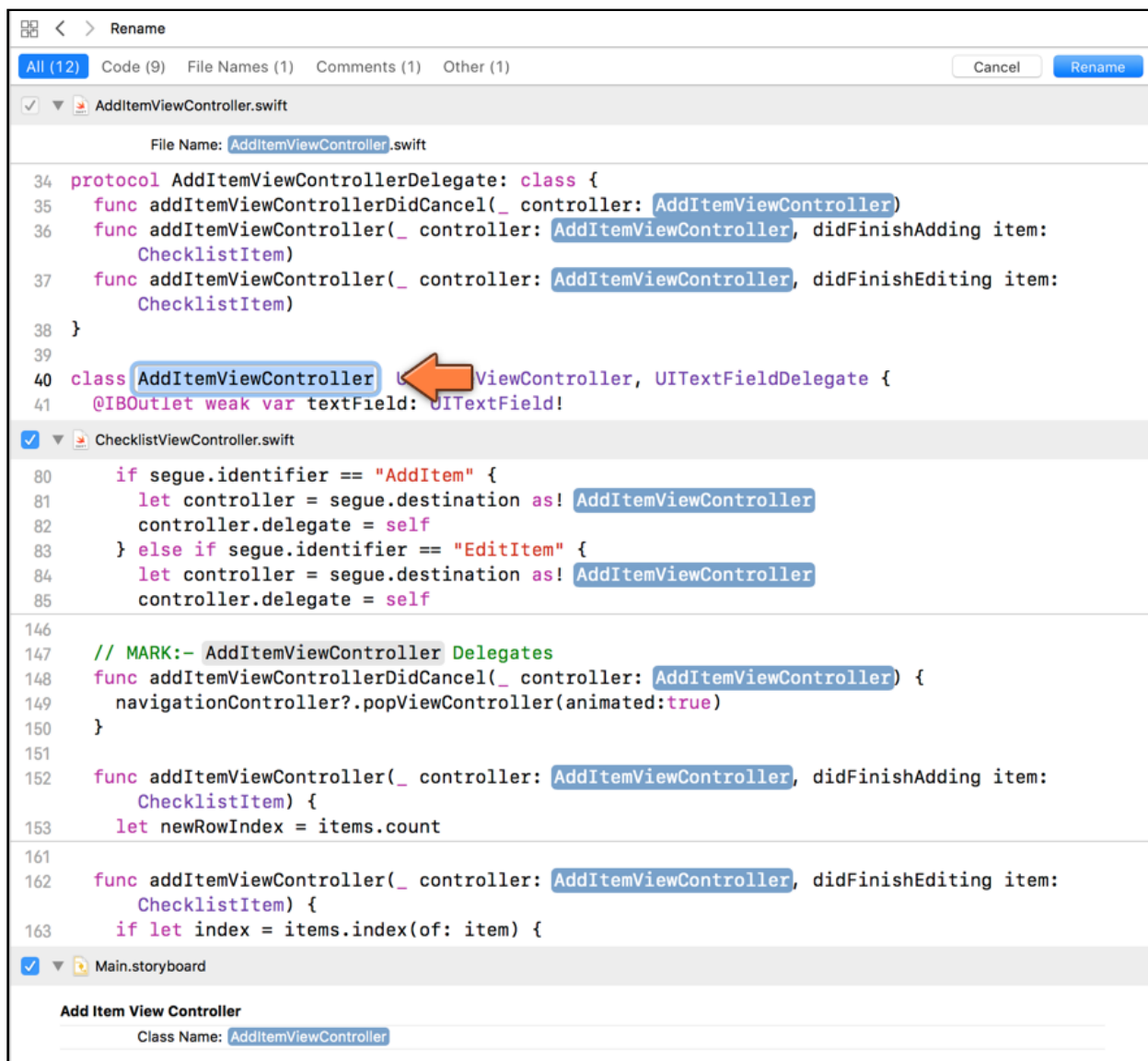


The Xcode context menu

You should notice two things about the above screenshot:

1. Notice how the class name (or method name, or variable name) that was under your cursor when you right-clicked was highlighted? That indicates that the highlighted name is the one that would be renamed.
2. Notice the **Rename...** option on the menu? It's this menu option which provides the refactor functionality.

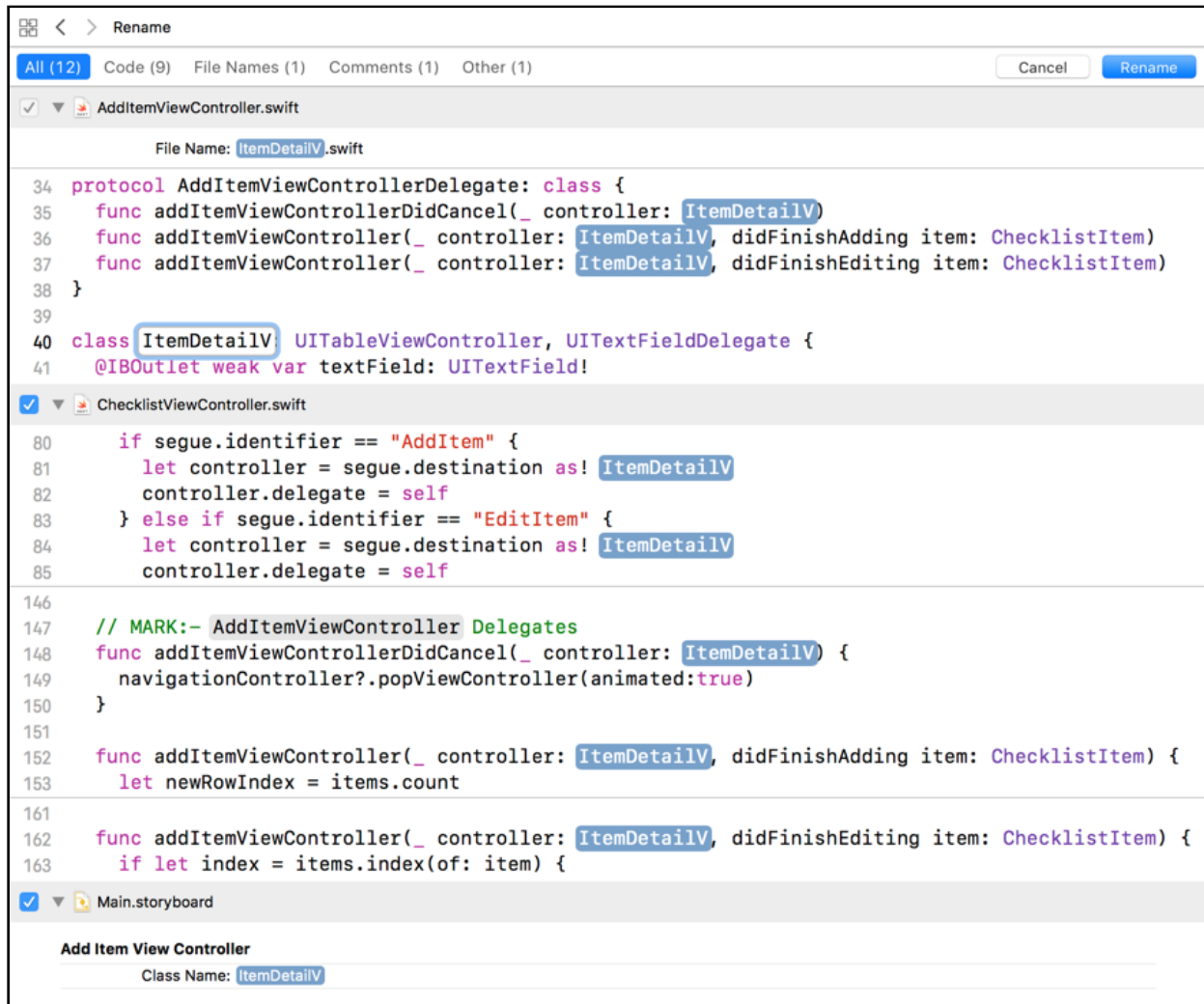
► If you right-clicked over the `AddItemViewController` class name, select the **Rename...** option now. (If you right-clicked elsewhere, first move your cursor over the class name, right-click, and then select **Rename...**). You should get a screen similar to the following:



Xcode rename view

The new screen shows you all the files and instances (including the storyboard and file names) in the project where the particular name you selected is used. Also notice how the name at the instance where you right-clicked is now editable.

Start typing in the new name you want and you'll notice that all the matching names for all the other instances in the view update in real-time. Cool!



Xcode real-time renaming

When you've entered the correct name and verified that everything will be updated correctly, just click the **Rename** button on the top right corner and you're done :] It's as simple as that!

Note: While the refactoring worked flawlessly most of the time, I've sometimes had Xcode do all the refactoring correctly except for renaming the file itself. If this does happen to you, you might have to rename the file manually.

Test the code after a refactor

Let's see if everything works correctly now.

► Press **⌘+B** to compile the app.

Note: Getting a “Build Failed” error? Sometimes this does happen after a massive change across the whole project like this. The first thing to try is to use the Xcode menu's **Product** → **Clean** option and try building again. It should work in most cases at that point.

Because you made quite a few changes all over the place, it's a good idea to clean up the debris and detritus from old compiler runs so that Xcode picks up all the new changes. You don't have to be paranoid about this, but it's good practice to clean house once in a while.

► From Xcode's menu bar choose **Product** → **Clean**. When the clean is done, choose **Product** → **Build** (or simply press the Run button).

If there are no build issues, run the app again and test the various features just to make sure everything still works! (If the build succeeds but Xcode still shows red error icons in your source file, then close the project and open it again, or restart Xcode. Restarting Xcode is the solution that Almost Always Works™. And if it doesn't, restarting your computer is the last resort. That does get rid of even the most stubborn issues.)

One more thing

The rename process appears to have gone through flawlessly, your app works fine when you test it, and there are no crashes. So, everything should be fine and you can move on to the next feature in the app, right?

Well ... not quite :) Switch to **ItemDetailViewController.swift** and check the protocol definition at the top. What do you see?

```
33
34 protocol AddItemViewControllerDelegate: class {
35     func addItemViewControllerDidCancel(_ controller: ItemDetailViewController)
36     func addItemViewController(_ controller: ItemDetailViewController, didFinishAdding item:
        ChecklistItem)
37     func addItemViewController(_ controller: ItemDetailViewController, didFinishEditing item:
        ChecklistItem)
38 }
```

The protocol name has not changed after renaming

Looks as if the protocol name, `AddItemViewControllerDelegate`, did not change when you renamed `AddItemViewController`.

If you think about it, it makes sense. `AddItemViewControllerDelegate` is a different entity than `AddItemViewController`. So all the renaming did was to change the all the references to `AddItemViewController` class, not the `AddItemViewControllerDelegate` protocol.

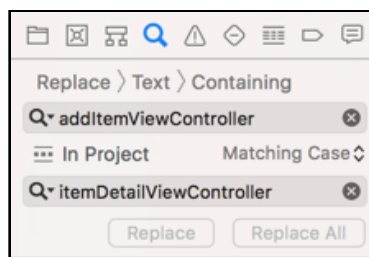
You can easily change the name of the protocol to `ItemDetailViewControllerDelegate` by using Xcode's rename functionality yet again. But you'll notice that that only changes the protocol name itself - not the protocol method names. Hmm ... this is getting to be a lot of work!

You can try renaming each protocol method separately and Xcode's rename functionality will do a good job with the renaming, but you'd have to do this three times for the three methods. This could get really time consuming, especially if you were dealing with a protocol with lots of methods. There's an easier way.

What is this easier way? To use Xcode's search and replace functionality, of course! As you'll notice, all that remains to change in the `ItemDetailViewControllerDelegate` is the method names, all of which begin with `addItemViewController`. So, if you can search for the term `addItemViewController` across the entire project and replace it with `itemDetailViewController`, you should be done!

Here's how you do it:

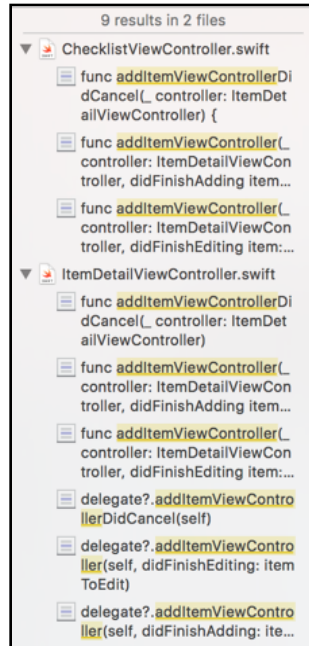
- Switch to the **Find navigator** (fourth tab in the navigator pane).
- Click on **Find** to change it to **Replace**.
- Change Ignoring Case to **Matching Case**.
- Type as the search text: **addItemViewController**. Important: Make sure you spell it exactly like this since your search term is going to be case-sensitive!
- Type in the replacement field: **itemDetailViewController**, again making sure that you type it exactly.



The search & replace options

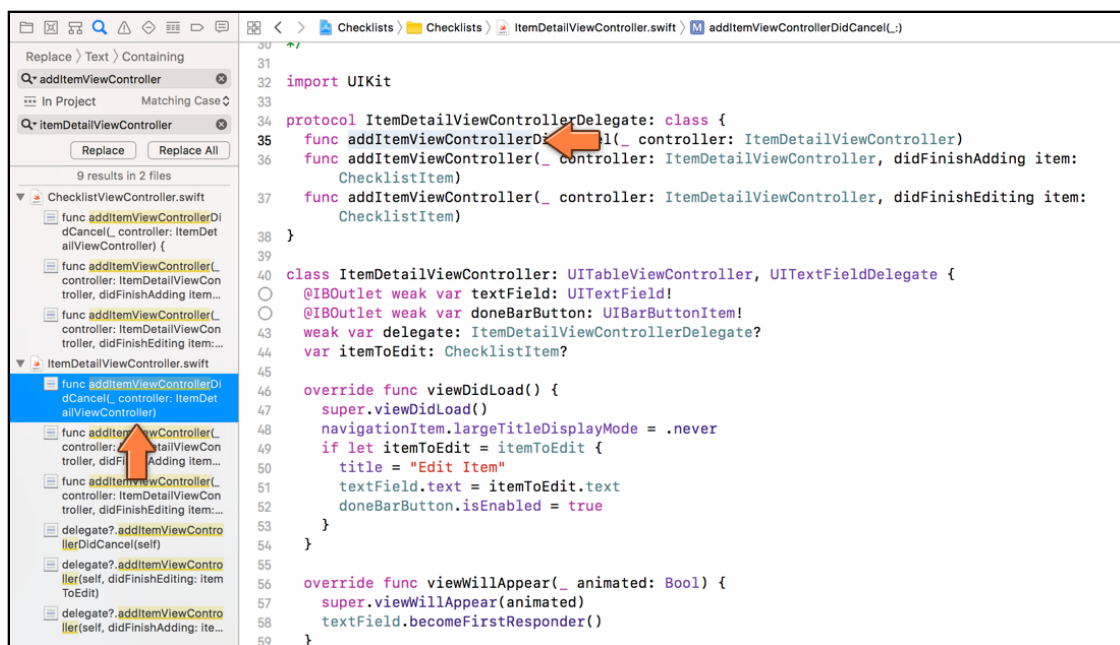
► Press **return** on your keyboard to start the search. This doesn't replace anything yet.

The Find navigator shows the files containing matches for the search term. You should see two Swift source files in this list.



The search results

► Click on any item in the file list above to be taken to that particular match in the relevant file with the match highlighted in the source code:



The results list allows you to verify each match

Have a look through the search results just to make sure Xcode isn't doing anything you'll regret later. It should only rename everything that says `addItemViewController` to `itemDetailViewController`.

► If you are satisfied that the matches are correct, click **Replace All**. (You could also select only some results in the list and then click **Replace** to have only those results be changed.)

I always repeat the search afterwards, ignoring case, to make sure I didn't skip anything by accident.

Now Run the app and test its functionality once again to make sure that everything works. If it does, you are done with this particular task, finally :]

Iterative development

If you think this approach to development we've taken so far is a little messy, then you're absolutely right.

You started out with one design, but as you continued development you found out that things didn't work out so well in practice, and that you had to refactor your approach a few times to find a way that works.

This is actually how software development goes in practice.

You first build a small part of your app and everything looks and works fine. Then you add the next small part on top of that and suddenly everything breaks down. The proper thing to do is to go back and restructure your entire approach so that everything is hunky-dory again... Until the next change you need to make.

Software development is a constant process of refinement. In this book I didn't want to just give you a perfect piece of code and explain how each part works. That's not how software development happens in the real world.

Instead, you're working your way from zero to a full app, exactly the way a pro developer would, including the mistakes and dead ends.

Isn't it possible to create a design up-front – sometimes called a “software architecture design” – that deals with all of these situations, something like a blueprint for software?

I don't believe in such designs. Sure, it's always good to plan ahead. Before writing this book, I made a few quick sketches of how I imagined each app would turn out. That was useful to envision the amount of work, but as usual, some of my assumptions and guesses turned out to be wrong and the design stopped being useful about halfway in.

And this is only a simple app!

That doesn't mean you shouldn't spend any time on planning and design, just not too much. ;-)

Simply start somewhere and keep going until you get stuck, then backtrack and improve on your approach. This is called *iterative development* and it's usually faster and provides better results than meticulous up-front planning.

You can find the project files for the app up to this point under **27 - Edit Items** in the Source Code folder.

Chapter 28: Saving and Loading

By Fahim Farook and Matthijs Hollemans

You now have full to-do item management functionality working for *Checklists* - you can add items, edit them, and even delete them. However, any new to-do items that you add to the list cease to exist when you terminate the app (by pressing the Stop button in Xcode, for example). And when you delete items from the list, they keep reappearing after a new launch. That's not how a real app should behave!

So, it's time to consider *data persistence* - or, to put it simply, saving and loading items ...

In this chapter you will cover the following:

- **The need for data persistence:** A quick look at why you need data persistence.
- **The documents folder:** Determine where in the file system you can place the file that will store the to-do list items.
- **Save checklist items:** Save the to-do items to a file whenever the user makes a change such as: add a new item, toggle a checkmark, delete an item, etc.
- **Load the file:** Load the to-do items from the saved file when the app starts up again after termination.

The need for data persistence

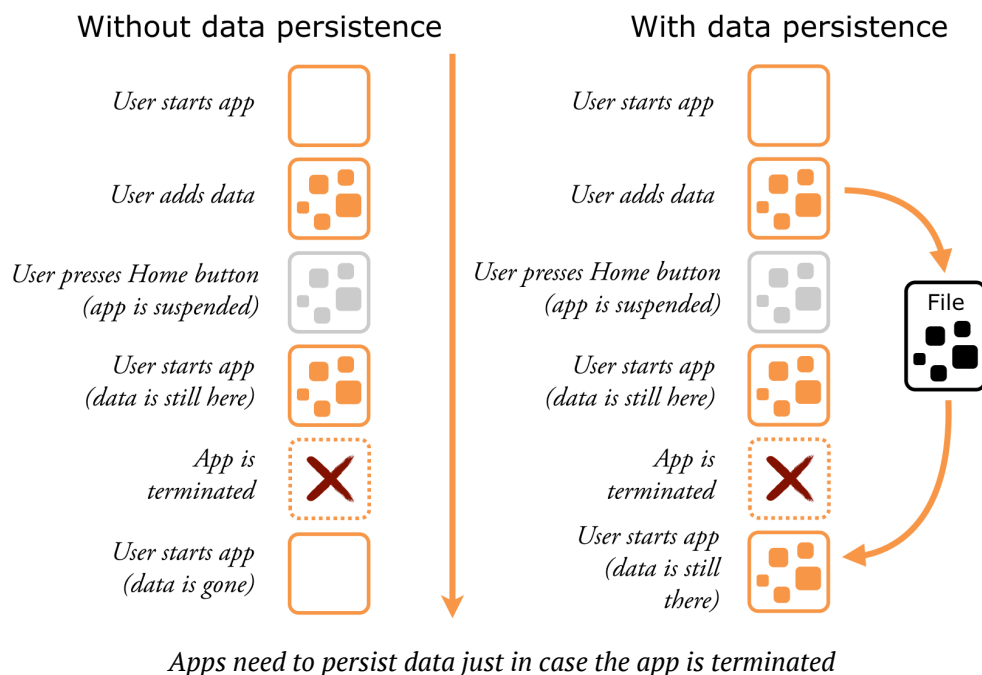
Thanks to the multitasking nature of iOS, an app stays in memory when you close it and go back to the home screen or switch to another app. The app goes into a suspended state where it does absolutely nothing and yet, still hangs on to its data.

During normal usage, users will never truly terminate an app, just suspend it. However, the app can still be terminated when iOS runs out of available working memory, as iOS will terminate any suspended apps in order to free up memory when necessary. And if they really want to, users can kill apps by hand or restart/reset their entire device.

Just keeping the list of items in memory is not good enough because there is no guarantee that the app will remain in memory forever, whether active or suspended.

Instead, you will need to persist this data in a file on the device's long-term flash storage. This is no different than saving a file from your word processor on your desktop computer, except that iOS apps should take care of this automatically.

The user shouldn't have to press a Save button just to make sure unsaved data is safely placed in long-term storage.



So let's get crackin' on that data persistence functionality!

The documents folder

iOS apps live in a sheltered environment known as the **sandbox**. Each app has its own folder for storing files but cannot access the directories or files of any other app.

This is a security measure, designed to prevent malicious software such as viruses from doing any damage. If an app can only change its own files, it cannot modify (or affect) any other part of the system.

Your apps can store files in the “Documents” folder in the app’s sandbox.

The contents of the Documents folder are backed up when the user syncs their device with iTunes or iCloud.

When you release a new version of your app and users install the update, the Documents folder is left untouched. Any data the app has saved into this folder stays there when the app is updated.

In other words, the Documents folder is the perfect place for storing your user’s data files.

Get the save file path

Let’s look at how this works.

► Add the following methods to **ChecklistViewController.swift**:

```
func documentsDirectory() -> URL {
    let paths = FileManager.default.urls(for: .documentDirectory,
                                         in: .userDomainMask)
    return paths[0]
}

func dataFilePath() -> URL {
    return documentsDirectory().appendingPathComponent(
        "Checklists.plist")
}
```

The `documentsDirectory()` method is something I’ve added for convenience. There is no standard method you can call to get the full path to the Documents folder, so I rolled my own.

The `dataFilePath()` method uses `documentsDirectory()` to construct the full path to the file that will store the checklist items. This file is named **Checklists.plist** and it lives inside the Documents folder.

Notice that both methods return a URL object. iOS uses URLs to refer to files in its filesystem. Where websites use `http://` or `https://` URLs, to refer to a file you use a `file://` URL.

Note: Double check to make sure your code says `.documentDirectory` and not `.documentationDirectory`. Xcode’s autocomplete can easily trip you up here!

► Still in **ChecklistViewController.swift**, add the following two print statements to the bottom of `init?(coder:)`, below the call to `super.init()`:

```
required init?(coder aDecoder: NSCoder) {  
    .  
    .  
    .  
    super.init(coder: aDecoder)  
  
    print("Documents folder is \(documentsDirectory())")  
    print("Data file path is \(dataFilePath())")  
}
```

► Run the app. Xcode's Console will now show you where your app's Documents folder is actually located.

If I run the app from the Simulator, on my system it shows something like this:

```
Documents folder is file:///Users/fahim/Library/Developer/  
CoreSimulator/Devices/CA23DAEA-DF30-43C3-8611-E713F96D4780/  
data/Containers/Data/Application/CA115C3A-E1FB-4EF9-A776-  
F434DAB8029E/Documents/  
Data file path is file:///Users/fahim/Library/Developer/  
CoreSimulator/Devices/CA23DAEA-DF30-43C3-8611-E713F96D4780/  
data/Containers/Data/Application/CA115C3A-E1FB-4EF9-A776-  
F434DAB8029E/Documents/Checklists.plist
```

All Output ▾

Filter



Console output showing Documents folder and data file locations

If you run it on your iPhone, the path will look somewhat different. Here's what mine says:

```
Documents folder is file:///var/mobile/Applications/  
FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE/Documents  
  
Data file path is file:///var/mobile/Applications/  
FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE/Documents/Checklists.plist
```

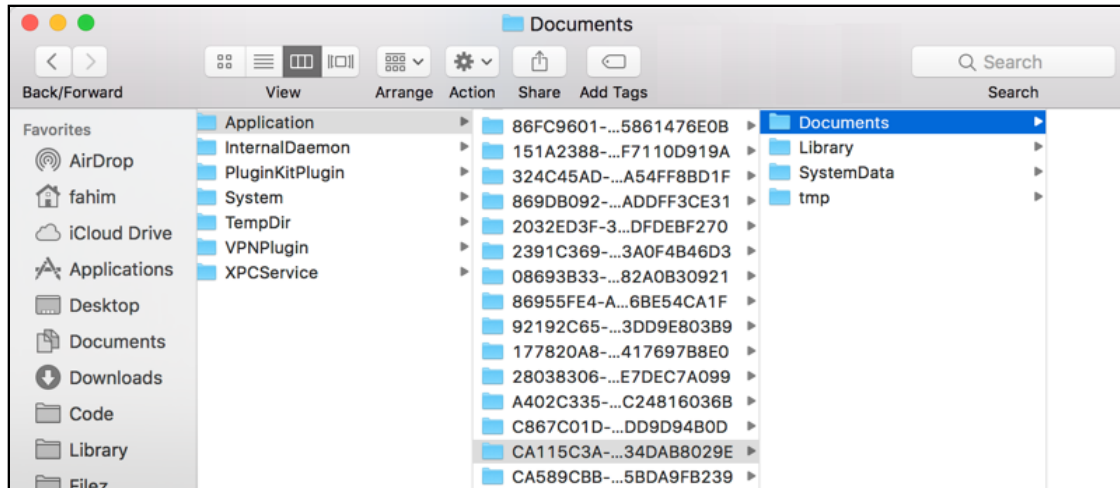
As you'll notice, the folder name is a random 32-character ID. Xcode picks this ID when it installs the app on the Simulator or the device. Anything inside that folder is part of the app's sandbox.

Browse the documents folder

For the rest of this app, run the app on the Simulator instead of a device. That makes it easier to look at the files you'll be writing into the Documents folder. Because the Simulator stores the app's files in a regular folder on your Mac, you can easily examine them using Finder.

► Open a new Finder window by clicking on the Desktop and typing **⌘+N** (or, by clicking the Finder icon in your dock, if you have one.). Then press **⌘+Shift+G** (or, select **Go → Go to Folder...** from the menu), copy the Documents folder path from Xcode Console, and paste the full path to the Documents folder in the dialog. (Don't include the **file://** bit. The path starts with **/Users/yourname/...**)

The Finder window will go to that folder. Keep this window open so you can verify that the Checklists.plist file is actually created when you get to that part.



The app's directory structure in the Simulator

Tip: If you want to navigate to the Simulator's app directory by traversing your folder structure, then you should know that the Library folder, which is in your home folder, is normally hidden. Hold down the Alt/Option key and click on Finder's Go menu (or hold down the Alt key while the Go menu is open). This will reveal a shortcut to the Library folder on the Go menu.

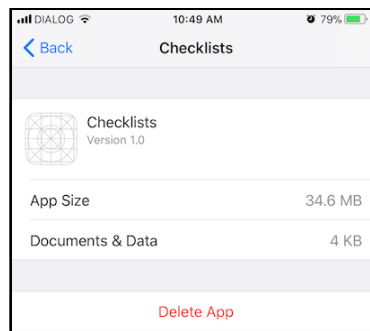
You can see several folders inside the app's sandbox folder:

- The Documents folder where the app will put its data files. Currently the Documents folder is empty.
- The Library folder has cache files and preferences files. The contents of this folder are managed by the operating system.
- The SystemData folder, as the name implies is for use by the operating system 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 usage. 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.

It is also possible to get an overview of the Documents folder of apps on your device.

► On your device, go to **Settings** → **General** → **iPhone Storage**, scroll down to the list of installed apps (you might have to wait for the list to load) and tap the name of an app.

You'll now see the size of the contents of its Documents folder (but not the actual content):



Viewing the Documents folder info on the device

Save checklist items

In this section you are going to write code that saves the list of to-do items to a file named `Checklists.plist` when the user adds a new item or edits an existing item. Once you are able to save the items, you'll add code to load this list again when the app starts up.

Plist files

So what is a **.plist** file?

You've already seen a file named `Info.plist` in the *Bull's Eye* lesson. All apps have one, including the *Checklists* app (see the project navigator). `Info.plist` contains several configuration options that give iOS additional information about the app, such as what name to display under the app's icon on the home screen.

“plist” stands for Property List and it is an XML file format that stores structured data, usually in the form of a list of settings and their values. Property List files are very common in iOS. They are suitable for many types of data storage, and best of all, they are simple to use. What's not to like?

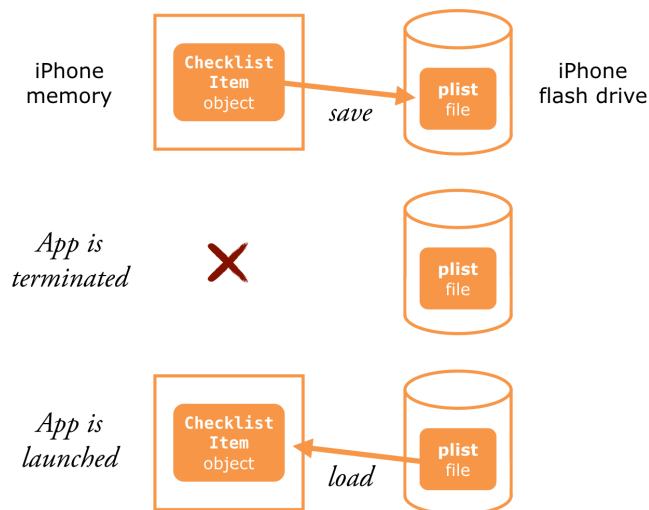
To save the checklist items, you'll use Swift's `Codable` protocol, which lets objects which support the `Codable` protocol to store themselves in a structured file format.

You actually don't have to care much about that format. In this case it happens to be a .plist file but you're not directly going to mess with that file. All you care about is that the data gets stored in some kind of file in the app's Documents folder, but you'll leave the technical details to Codable.

While Codable is a new protocol introduced in Swift 4, you have already used its Objective-C cousin, NSCoder, behind the scenes because that's exactly how storyboards work. When you add a view controller to a storyboard, Xcode uses the NSCoder system to write this object to a file (encoding). Then when your application starts up, it uses NSCoder again to read the objects from the storyboard file (decoding). The Codable protocol works similarly.

The process of converting objects to files and back again is also known as **serialization**. It's a big topic in software engineering.

I like to think of this whole process as freezing objects. You take a living object and freeze it so that it is suspended in time. You store that frozen object into 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 to bring it back to life again.



The process of freezing (saving) and unfreezing (loading) objects

Save data to a file

► Add the following method to **ChecklistViewController.swift**:

```
func saveChecklistItems() {
    // 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!")
}
}
```

This method takes the contents of the `items` array, converts it to a block of binary data, and then writes this data to a file. Let's take the commented lines step-by-step to understand the code:

1. First create an instance of `PropertyListEncoder` which will encode the `items` array and all the `CheckListItem`s in it into some sort of binary data format that can be written to a file.
2. The `do` keyword, which you have not encountered before, sets up a block of code to catch Swift errors. Swift handles errors under certain conditions by *throwing* an error. In such cases, you need a block of code to catch the error and to handle it. The `do` keyword indicates the start of such a block. You will see the error catching code after comment #5, where the `catch` keyword can be seen.
3. The encoder you created in earlier is used to try to encode the `items` array. The `encode` method throws a Swift error if it is unable to encode the data for some reason - for example, the data is not in the expected format, or it is corrupted etc. The `try` keyword indicates that the call to encode can fail and if that happens, that it will throw an error. (If you do not have the `try` keyword before a call to a method which throws an error, you will get an Xcode error. Try it and see.) If the call to encode fails, execution will immediately jump to the catch block instead of proceeding on to the next line.
4. If the data constant was successfully created by the call to encode in the previous line, then you write the data to a file using the file path returned by a call to `dataFilePath()`. Note that the `write` method also can throw an error. So again, you have to precede the method call with another `try` statement.
5. The `catch` statement indicates the block of code to be executed if an error was thrown by any line of code in the enclosing `do` block.
6. Handle the caught error. Here, you simply print out an error message to the Xcode Console.

It's not really important that you understand how `PropertyListEncoder` works internally. The format that it stores the data in is not relevant. All you care about is that it allows you to put your objects into a file and read them back later.

You have to call this new `saveChecklistItems()` method whenever the list of items is modified.

Exercise: Where in the source code would you call this method?

Answer: Look at where the `items` array is modified. This happens inside the `ItemDetailViewControllerDelegate` methods. That's where the party's at!

► Add a call to `saveChecklistItems()` to the end of these methods in **ChecklistViewController.swift**:

```
func itemDetailViewController(
    _ controller: ItemDetailViewController,
    didFinishAdding item: ChecklistItem) {
    ...
    saveChecklistItems()
}
```

```
func itemDetailViewController(
    _ controller: ItemDetailViewController,
    didFinishEditing item: ChecklistItem) {
    ...
    saveChecklistItems()
}
```

► Let's not forget the swipe-to-delete function:

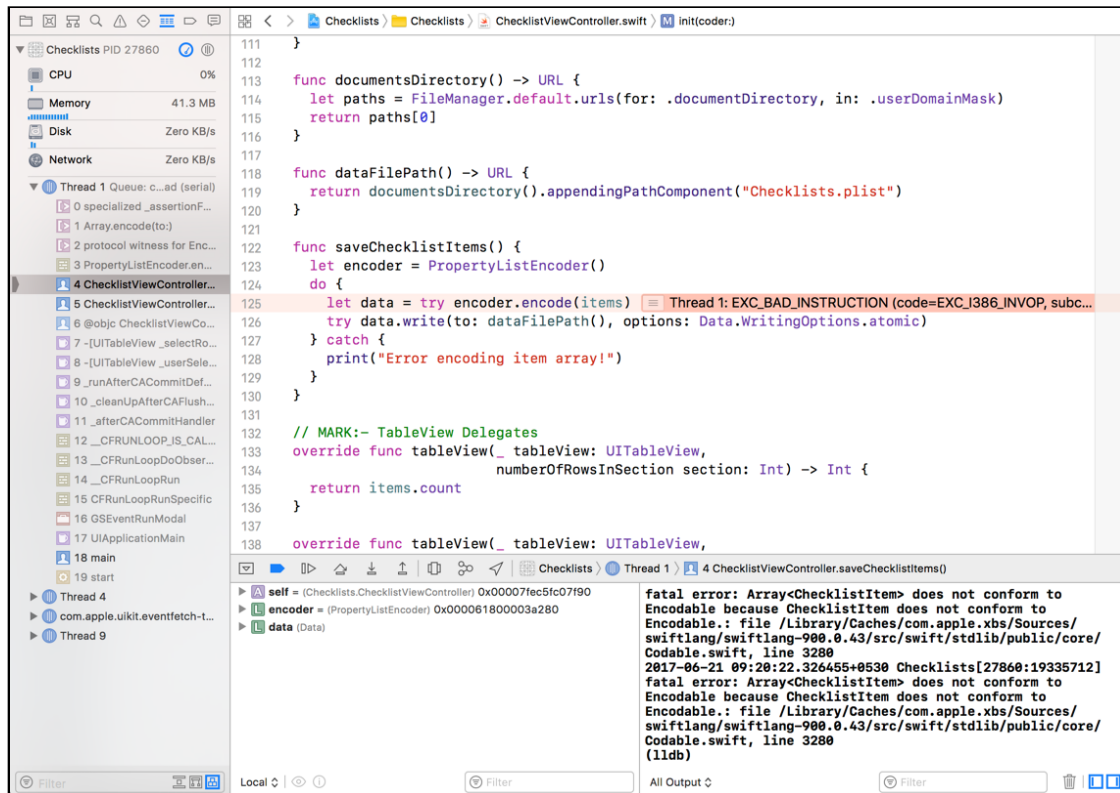
```
override func tableView(
    _ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath) {
    ...
    saveChecklistItems()
}
```

► And toggling the checkmark on a row on or off:

```
override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {
    ...
    saveChecklistItems()
}
```


The Codable protocol

Just encoding the `items` array using `PropertyListEncoder` is not enough. If you were to run the app now and do something that results in a save, such as tapping a row to flip the checkmark, the app will crash (try it out):



Xcode error about Codable support

The Xcode window has switched to the *debugger* and points out which line caused the crash, more or less, but the error message in the source editor is not very helpful. However, if you look at the Console, you'll see a much more descriptive error indicating the issue:

```
fatal error: Array<CheckListItem> does not conform to Encodable because
CheckListItem does not conform to Encodable.: file /Library/Caches/
com.apple.xbs/Sources/swiflang/swiflang-900.0.43/src/swift/stdlib/
public/core/Codable.swift, line 3280
```

According to the above error, the `PropertyListEncoder` was unable to encode the `items` array because `CheckListItem` does not conform to the `Encodable` protocol. Now you might be wondering where `Encodable` came from since I only talked about a `Codable` protocol before.

The thing is, Codable is a protocol which combines two other protocols - Encodable and Decodable. Basically, the Codable protocol encompasses both sides of the serialization process. But since at this point what you wanted to do was encode something, you get an error about the missing Encodable protocol support. If you later tried to decode a ChecklistItem you'd get an error about the missing Decodable protocol support as well.

Just to clarify, most standard Swift objects and structures support the Codable protocol by default. So, you don't get an error here for the items array itself. But ChecklistItem is a custom object that we created. The issue is with ChecklistItem and it is very easy to fix :]

► Switch to **CheckListItem.swift** and modify the class line as follows:

```
class ChecklistItem: NSObject, Codable {
```

In the above, you tell the compiler that ChecklistItem will conform to the Codable protocol. That's all you need to do!

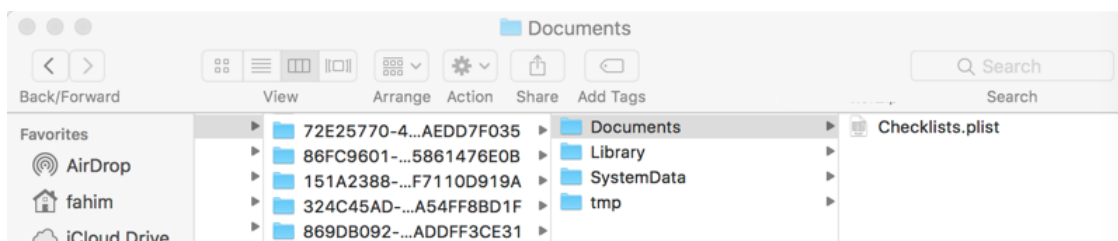
"Now, hold on," I hear you say. "We had to implement methods to support a protocol before. How come we don't have to do that here?"

Remember how I mentioned in a previous chapter that protocols can have default implementations? No? OK, it was in the **Delegates and Protocols** chapter in the section about protocols :] Sometimes, it is useful to have a default implementation for a protocol to provide functionality that would make things easier - or would cover a lot of standard scenarios.

In our case, all of the properties of ChecklistItem are standard Swift types and Swift already knows how to encode/decode those types. So, we can simply piggyback on things without having to write any code of our own to implement encoding/decoding in ChecklistItem. Handy, eh?

Verify the saved file

- Run the app again and tap a row to toggle a checkmark. The app didn't crash? Good!
- Go to the Finder window that has the app's Documents directory open:



The Documents directory now contains a Checklists.plist file

There is now a **Checklists.plist** file in the Documents folder, which contains the items from the list.

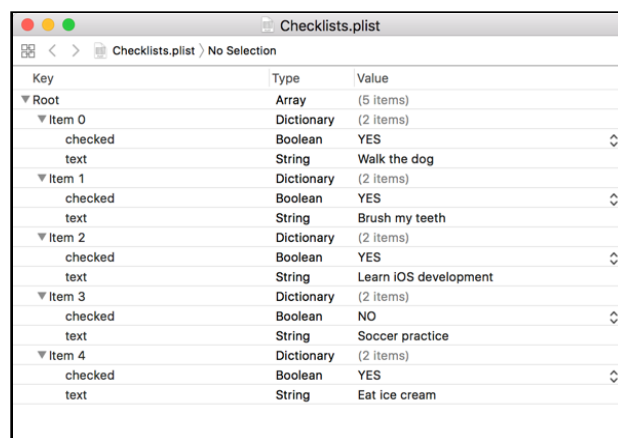
You can look inside this file if you want, but the contents won't make much sense. Even though it is XML, this file wasn't intended to be read by humans, only by something like `PropertyListDecoder`, the counterpart to the `PropertyListEncoder` that we already used.

If you're having trouble viewing the XML, it may be because the plist file isn't stored as text but as a binary format. Some text editors support this file format and can read it as if it were text (`TextWrangler` is a good one and is a free download on the Mac App Store).

You can also use Finder's Quick Look feature to view the file. Simply select the file in Finder and press the space bar.

Naturally, you can also open the plist file with Xcode.

► Right-click the `Checklists.plist` file and choose **Open With → Xcode**.



Key	Type	Value
▼ Root	Array	(5 items)
▼ Item 0	Dictionary	(2 items)
checked	Boolean	YES
text	String	Walk the dog
▼ Item 1	Dictionary	(2 items)
checked	Boolean	YES
text	String	Brush my teeth
▼ Item 2	Dictionary	(2 items)
checked	Boolean	YES
text	String	Learn iOS development
▼ Item 3	Dictionary	(2 items)
checked	Boolean	NO
text	String	Soccer practice
▼ Item 4	Dictionary	(2 items)
checked	Boolean	YES
text	String	Eat ice cream

Checklist.plist in Xcode

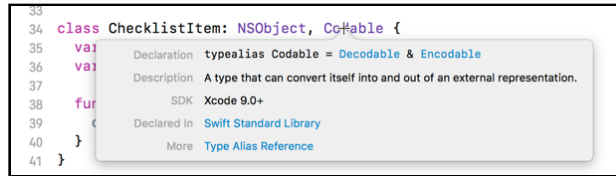
It still won't make much sense but it's fun to look at anyway.

Expand some of the rows and you'll see that the names of the `ChecklistItems` are in there as well as their checked/unchecked state. But exactly how all these data items fit together, might not make much sense to you just yet.

"NS" objects

Objects whose name start with the "NS" prefix, like `NSObject`, `NSString`, or `NSCoder`, are provided by the Foundation framework. NS stands for NextStep, the operating system from the 1990's that later became Mac OS X and which also forms the basis of iOS.

If you are curious about exactly how objects such as `NSObject` and `NSString` work, you can Alt/Option-click any item in your source code to bring up a popup with a brief description. And this works for non-NS prefixed objects too :] In fact, you can look up details about any class, object, variable, or method this way in Xcode.



I use this all the time to remind myself of how to use framework objects and their methods. You can click on any of the blue color items since they are links to more detailed documentation on the topic. That will take you to the Developer Documentation app which lets you read up further on the selected subject.

It's good to have a general idea of what objects are available in the frameworks, but no one can remember all the specifics. So get into the habit of looking up the documentation for any new objects and methods that you encounter. It'll help you learn the iOS frameworks that much quicker!

Load the file

Saving is all well and good, but pretty useless by itself. So, let's also implement the loading of the `Checklists.plist` file. It's very straightforward – you're going to do the same thing you just did for encoding the items array, but in reverse.

Read data from a file

► Switch to **ChecklistViewController.swift** and add the following new method:

```

func loadChecklistItems() {
    // 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)
        } catch {
            print("Error decoding item array!")
        }
    }
}

```

Let's go through this step-by-step:

1. First you put the results of `dataFilePath()` in a temporary constant named `path`.
2. Try to load the contents of `Checklists.plist` into a new `Data` object. The `try?` command attempts to create the `Data` object, but returns `nil` if it fails. That's why you put it in an `if let` statement.

Why would it fail? If there is no `Checklists.plist` then there are obviously no `CheckListItem` objects to load. This is what happens when the app is started up for the very first time. In that case, you'll skip the rest of this method.

Also, do notice that this is another way to use the `try` statement - instead of enclosing the `try` statement within a `do` block, like you did previously, you can have a `try?` statement which indicates that the `try` could fail and if it does, that 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 `Checklists.plist` file, you'll load the entire array and its contents from the file using a `PropertyListDecoder`. So, create the decoder instance.
4. Load the saved data back into `items` using the decoder's `decode` method. The only item of interest here would be the first parameter passed to `decode`. The decoder needs to know what type of data will be the result of the `decode` operation and you let it know by indicating that it will be an array of `CheckListItem` objects.

This populates the array with exact copies of the `CheckListItem` objects that were frozen into the `Checklists.plist` file.

You now have your `loadChecklistItems()` method, but it needs to be called from somewhere in order for this to work. There are several places from which you can do this.

Take a look at the current coder in **`ChecklistViewController.swift`** - you have `init?(coder:)` which currently populates the static data displayed by the app, but you also have `viewDidLoad()` which is called when the view controller is first loaded. So which should you use?

The difference between the two is this - `init?(coder:)` is called only when the view controller is created from a storyboard. However, view controllers could also be instantiated from code, as you'll find out later. So, in the second case, depending on how you write your code, `init?(coder:)` might not be called, but `viewDidLoad()` will always be called for a view controller no matter how the view controller was created.

So, my vote is to delete the `init?(coder:)` implementation (to get rid of the static data that is loaded on app start up), and to call `loadChecklistItems()` from `viewDidLoad()`. (If you decide to go the other way, do remember to clean up `init?(coder:)` so as to remove the static item loading.)

Load the saved data on app start

Here's what you need to do:

- Change the following line in **ChecklistViewController.swift** (at the very top):

```
var items: [CheckListItem]
```

To:

```
var items = [CheckListItem]()
```

The only difference between the two is that in the former, you declare `items` as being an array of `CheckListItem` objects, but you don't initialize it, in the latter, you initialize `items` to be an empty `CheckListItem` array.

Now, when `ChecklistViewController` is created, it would have `items` initialized to an empty array instead of you having to do this explicitly in an `init` method or in `viewDidLoad`. Personally, I find it easier and simpler to have variable (or constant) declarations and initializations at the same place where possible. Again, you can follow whichever practice which best suits you - there is no right or wrong way :]

- Remove the `init?(coder:)` method from **ChecklistViewController.swift**.

Now that you initialize the `items` array at the top of the class, you don't have any code in `init?(coder:)` that is useful. So, you can delete all the static item creation code (and the method itself) to clean up your code a bit.

- Add a call to `loadChecklistItems()` in `viewDidLoad()` so that the method looks like this:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    // Enable large titles  
    navigationController?.navigationBar.prefersLargeTitles = true  
    // Load items  
    loadChecklistItems()  
}
```

You don't need to add the comments in there but it's always good to have some comments in your source so that you can understand your own code a month or two (or a few years) down the line :]

All that's changed in the above is the addition of a call to `loadChecklistItems()` to ensure that the saved item data is loaded back when the view controller is first loaded.

- Run the app and make some changes to the to-do items. Press Stop to terminate the app. Start it again and notice that your changes are still there.
- Stop the app again. Go to the Finder window with the Documents folder and remove the `Checklists.plist` file. Run the app once more. You should now have an empty list of items.
- Add an item and notice that the `Checklists.plist` file re-appears.

Awesome! You've written an app that not only lets you add and edit data, but which also persists the data between sessions. These techniques form the basis of many, many apps.

Being able to use a navigation controller, show secondary screens, and pass data around through delegates are essential iOS development skills.

Initializers

Methods named `init` are special in Swift. They are only used when you're creating new objects, to make those new objects ready for use.

Think of it as having bought 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 go change and 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 item = ChecklistItem()
```

Swift first allocates a chunk of memory big enough to hold the new object and then calls `ChecklistItem`'s `init()` method with no parameters.

It is pretty common for objects to have more than one `init` method. Which one is used depends on the circumstances.

For example, amongst the `init` methods for `UITableViewController` you'll find - `init(nibName:bundle:)`, `init(style:)` and the one you've already seen, `init?(coder:)`. As you've already learnt, `init?(coder:)` is used when the view controller is instantiated from a storyboard. But you can also create a `UITableViewController` instance directly by calling either `init(nibName:bundle:)` or `init(style:)`. So, how you initialize an object depends on the circumstances.

The implementations of these `init` methods, whether they're just called `init()` or `init?(coder:)` or something else, always follow the same series of steps. When you write your own `init` methods, you need to stick to those steps as well.

This is the standard way to write an `init` method:

```
init() {  
    // Put values into your instance variables and constants.  
  
    super.init()  
  
    // Other initialization code, such as calling methods, goes here.  
}
```

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 is necessary when you're adding the `init` method to an object that is a subclass of some other object. Much more about that later.

The question mark is for when `init?` can potentially fail and return a `nil` value instead of a real object. You can imagine that 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  
    super.init()  
}
```

You must use either 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, they do not need to have a value (in which case they are `nil`).

Once you've given all your instance variables and constants values, you call `super.init()` to initialize the object's superclass. If you haven't done any object-oriented programming at all, you may not know what a *superclass* is. That's fine; we'll completely ignore this topic till later.

Just remember that sometimes objects need to send messages to something called *super* and if you forget to do this, bad things are likely to happen.

After calling `super.init()`, you can do additional initialization, such as calling the object's own methods. You're not allowed to do that before the call to `super.init()` because Swift has no guarantee that your object's variables all have proper values until then.

You don't always need to provide an `init` method. If your `init` method doesn't need to do anything – if there are no instance variables to fill in – then you can leave it out completely and the compiler will provide one for you. As an example, take a look at `CheckListItem` – it doesn't have an `init()` method since all its variables are initialized when they are declared.

Swift's rules for initializers can be a bit complicated, but fortunately the compiler will remind you when you forget to provide an `init` method.

What next?

Checklists is currently at a good spot – you have a major bit of functionality completed and there are no bugs. This is a good time to take a break, put your feet up, and daydream about all the cool apps you'll soon be writing :]

It's also smart 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've been doing.

As always, feel free to change the app and experiment. Breaking things is allowed – even encouraged – here at iOS Apprentice Academy!

You can find the project files for the app up to this point under **28 - Saving and Loading** in the Source Code folder.

Chapter 29: Lists

By Fahim Farook and Matthijs Hollemans

Just to make sure you truly get everything you’ve done so far, next up, you’ll expand the app with new features that more or less repeat what you just did.

But I’ll also throw in a few twists to keep it interesting...

The app is named *Checklists* for a reason: it allows you to keep more than one list of to-do items. So far though, the app has only supported a single list. Now you’ll add the capability to handle multiple checklists.

In order to complete the functionality for this chapter, you will need two new screens, and that means two new view controllers:

1. `AllListsViewController` shows all the user’s lists.
2. `ListDetailViewController` allows adding a new list and editing the name and icon of an existing list.

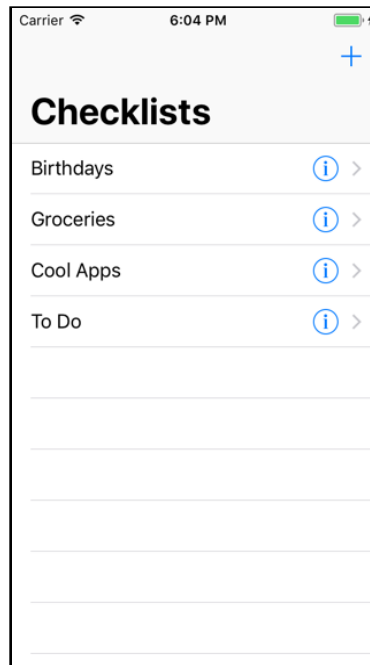
This chapter covers the following:

- **The All Lists view controllers:** Add a new view controller to show all the lists of to-do items.
- **The All Lists UI:** Complete the user interface for the All Lists screen.
- **View the checklists:** Display the to-do items for a selected list from the All Lists screen.
- **Manage checklists:** Add a view controller to add/edit checklists.

The All Lists view controller

You will first add `AllListsViewController`. This becomes the new main screen of the app.

When you're done, this is what it will look like:



The new main screen of the app

This screen is very similar to what you created before. It's a table view controller that shows a list of `Checklist` objects (not `ChecklistItem` objects).

From now on, I will refer to this screen as the “All Lists” screen, and to the screen that shows the to-do items from a single checklist as the “Checklist” screen.

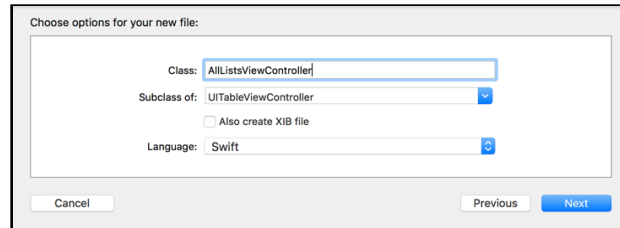
Add the new view controller

► Right-click the Checklists group in the project navigator and choose **New File**. Choose the **Cocoa Touch Class** template (under iOS, Source).

In the next step, choose the following options:

- Class: **AllListsViewController**
- Subclass of: **UITableViewController**
- Also create XIB file: Make sure this is **not** checked

- Language: **Swift**



Choosing the options for the new view controller

Note: Make sure the “Subclass of” field is set to **UITableViewController**, not “UIViewController”. Also be careful that Xcode didn’t rename what you typed into Class to “AllListsTableViewController” with the extra word “Table” when you change the “Subclass of” value. It can be sneaky like that...

- Press **Next** and then **Create** to finish.

As you might remember from a previous chapter, the Xcode template for a table view controller puts a lot of stuff in this new file that you don’t need. The template assumes you’ll fill in this placeholder code (known as *boilerplate* code) before you run the app. Let’s clean that up first.

You’ll also put some fake data in the table view just to get it up and running. As you know by now, I always like to take as small a step as possible and then run the app to see if it’s working. Once everything works, you can move forward and put in the real data.

Clean up the boilerplate code

- In **AllListsViewController.swift**, remove the `numberOfSections(in:)` method. Without it, there will always be a single section in the table view.

- Change the `tableView(_:numberOfRowsInSection:)` method to:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return 3
}
```

- Implement the `tableView(_:cellForRowAt:)` method to put some text into the cells, just so there is something to see.

Note that the boilerplate code already contains a commented-out version of this method. You can uncomment it by removing the `/*` and `*/` surrounding the method, and make your changes there.

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    let cell = makeCell(for: tableView)
    cell.textLabel!.text = "List \(indexPath.row)"
    return cell
}
```

In `CheckListViewController` the table view used prototype cells that you designed in Interface Builder. Just for the fun of it, in `AllListsViewController` you are taking a different approach where you'll create the cells in code instead.

► That requires you to add the following helper method:

```
func makeCell(for tableView: UITableView) -> UITableViewCell {
    let cellIdentifier = "Cell"
    if let cell =
        tableView.dequeueReusableCell(withIdentifier: cellIdentifier) {
        return cell
    } else {
        return UITableViewCell(style: .default,
                               reuseIdentifier: cellIdentifier)
    }
}
```

Later on I'll explain in more detail how this works, but for now recognize that you're using `dequeueReusableCell(withIdentifier:)` here too. If it returns `nil`, there is no cell that can be recycled and you construct a new one with `UITableViewCell(style:reuseIdentifier:)`.

The reason you put this logic into a separate method is so that it keeps the code in `tableView(_:cellForRowAt:)` simple and clean. I find it more readable that way.

► Remove all the commented-out cruft from `AllListsViewController.swift`. Xcode puts it there to be helpful, but it also makes a mess of things.

Special comments

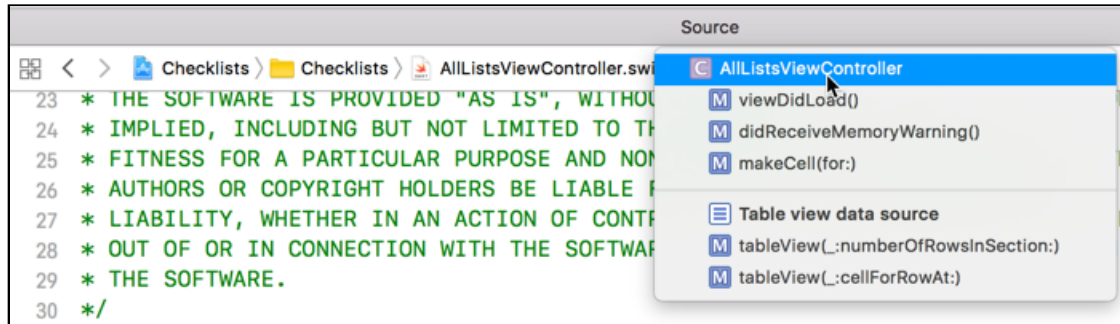
You might have noticed lines like the following in the boilerplate code in `AllListsViewController.swift`:

```
// MARK: - Table view data source
```

If you were wondering what they were, here's the scoop. Of course, you already know that they are comments, because the lines being with `//`, but they are not *just*

comments. As the keyword at the beginning of the comment line, MARK, indicates, they are markers. But markers for what?

They are markers to organize the code and for you to find a section of code (for example, a set of related methods, like for table view delegates) via the Xcode Jump Bar. Take a look at the method list for **AllListsViewController.swift** in the Xcode Jump Bar:



The Xcode Jump Bar with code organization

Notice the separator line in the middle of the list of methods? Do you notice the bolded text title right after? Does that 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.

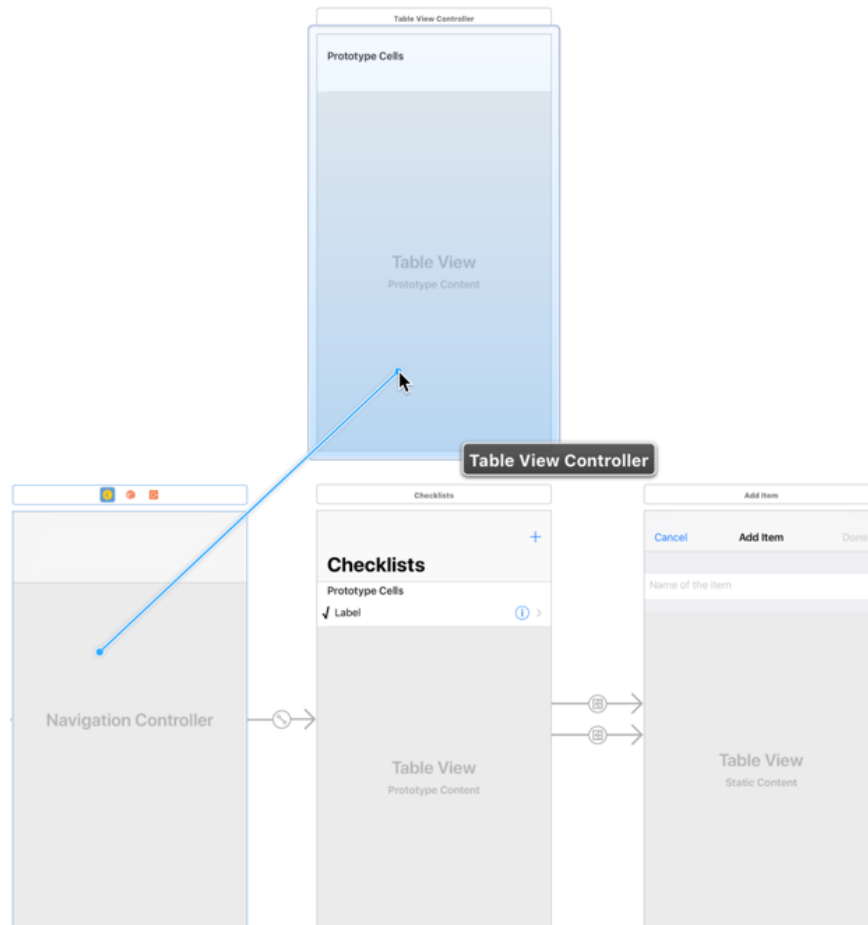
Consider using these tags to organize your code better. When you are in a hurry and need to find that particular bit of code in a long source file, they come in very handy. I certainly use them all the time in my own code :]

Storyboard changes

The final step is to add the new view controller to the storyboard.

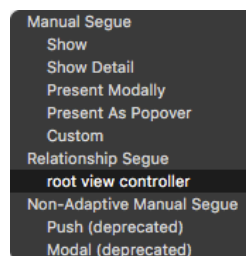
► Open the storyboard and drag a new **Table View Controller** onto the canvas. Put it somewhere near the initial navigation controller.

► **Control-drag** from the navigation controller to this new table view controller:



Control-drag from the navigation controller to the new table view controller

From the popup menu choose **Relationship Segue - root view controller**:



Relationships are also segues

This will break the existing connection between the navigation controller and the `ChecklistViewController` so that “Checklists” is no longer the app’s main screen.

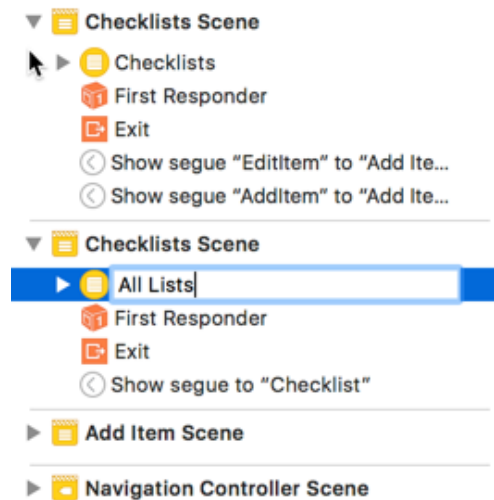
► Select the new table view controller and set its **Class** in the **Identity inspector** to **AllListsViewController**.

► Select the new view controller's Navigation Item in the Document Outline and then change its title to **Checklists** via the Attributes Inspector.

This may make Xcode rename the view controller in the Document Outline from All Lists View Controller to just Checklists. (Sometimes it won't happen till you restart Xcode.) This is a bit confusing because there's a Checklists view controller already.

It's simple enough to fix the scene names. Normally, the scene name is based on either the underlying view controller name or the navigation item title. But you can set whatever you want as the scene name by simply changing the displayed title on the Document Outline :]

► Tap the new view controller in the Document Outline (the yellow circle, not the rectangle representing the scene) and then tap it again to put the title into edit mode. Then, just rename it to **All Lists**.



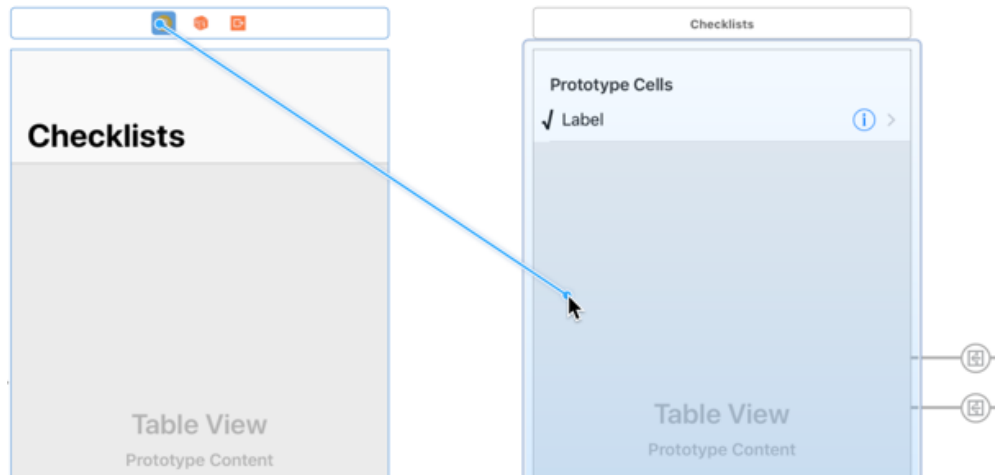
Rename scene

► Repeat the above step to rename the remaining Checklists scene to **Checklist** (note the missing "s" at the end).

You may want to reorganize your storyboard at this point to make everything look neat again. The All Lists scene goes in between the other scenes.

As I mentioned, you're not going to use prototype cells for this table view. It would be perfectly fine if you did, and as an exercise you could rewrite the code to use prototype cells later, but I want to show you another way of making table view cells.

- Delete the empty prototype cell from the All Lists scene. (Simply select the Table View Cell and press **delete** on your keyboard.)
- **Control-drag** from the yellow circle icon at the top of All Lists scene on to the Checklist scene and create a **Show** segue.



Control-dragging from the All Lists scene to the Checklist scene

This adds a “push” transition from the All Lists screen to the Checklist screen. It also puts the navigation bar back on the Checklist scene (the one on the right).

- Double-click the navigation bar on the Checklist scene to change its title to **(Name of the Checklist)**. This is just placeholder text.

Note that the new segue isn’t attached to any button or table view cell.

There is nothing on the All Lists screen that you can tap or otherwise interact with in order to trigger this segue. That means you have to perform the segue programmatically.

Perform a segue via code

- Click on the new segue to select it, go to the **Attributes inspector** and give it the identifier **ShowChecklist**.

The segue **Kind** should be **Show (e.g. Push)** because you’re pushing the Checklist View Controller onto the navigation stack when performing this segue.

- In **AllListsViewController.swift**, add the `tableView(_:didSelectRowAt:)` method:

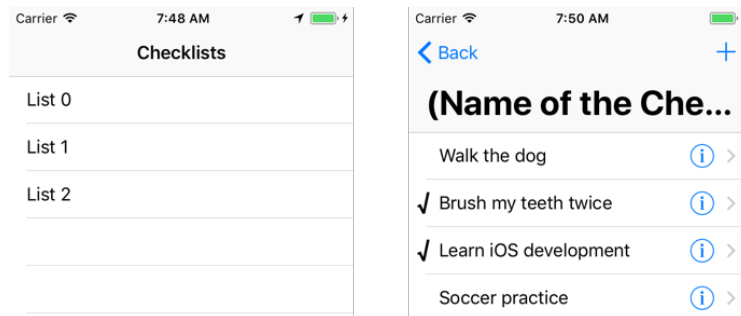
```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    performSegue(withIdentifier: "ShowChecklist", sender: nil)
}
```

Recall that this table view delegate method is invoked when you tap a row.

Previously, a tap on a row would automatically perform the segue because you had hooked up the segue to the prototype cell. However, the table view for this screen isn't using prototype cells. Therefore, you have to perform the segue manually.

That's simple enough: just call `performSegue(withIdentifier:sender:)` with the name of the segue and things will start moving.

► Run the app. It now looks like this:



The first version of the All Lists screen (left). Tapping a row opens the Checklist screen (right).

Tap a row and the familiar `ChecklistViewController` slides into the screen.

You can tap the “Back” button in the top-left to go back to the main list. Now you’re truly using the power of the navigation controller!

Fix the titles

Notice something about the titles on the two screens? (This only happens if you configured large titles via code. If you have been setting large titles via storyboard, then you simply have to change the Navigation Item setting on the Checklist scene and ignore the rest of the “Fix the titles” section.)

The second screen, Checklist, has the large title while the first one doesn't! This is because we originally set up large titles for `ChecklistViewController.swift`.

Exercise: Can you fix the titles on your own so that the large titles are enabled by `AllListsViewController.swift` and the Checklist screen does not show a large title?

The change is simple enough to implement.

- Move the following lines of code from `viewDidLoad` in **ChecklistViewController.swift** to `viewDidLoad` in **AllListsViewController.swift**:

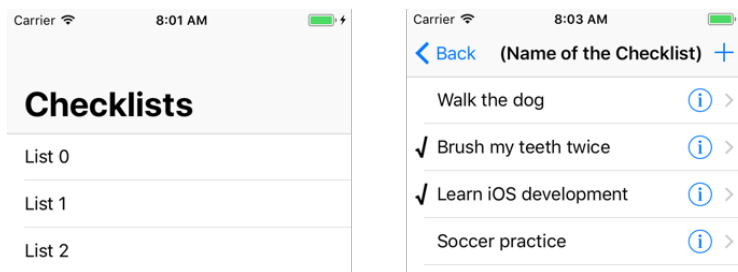
```
// Enable large titles
navigationController?.navigationBar.prefersLargeTitles = true
```

- Add this code to `viewDidLoad` in **ChecklistViewController.swift**:

```
// Disable large titles for this view controller
navigationItem.largeTitleDisplayMode = .never
```

In each case, the comments explain what the code does :]

Run the app again and you should see that the titles now display correctly.



The All Lists screen now shows large titles.

The All Lists UI

You're going to duplicate most of the functionality from the Checklist View Controller for this new All Lists screen.

There will be a + button at the top that lets users add new checklists, they can do swipe-to-delete, and they can tap the disclosure button to edit the name of the checklist.

Of course, you'll also save the array of Checklist objects to the Checklists.plist file.

Because you've already seen how this works, we'll go through the steps a bit quicker this time.

The data model

You begin by creating a data model object that represents a checklist.

- Add a new file to the project based on the **Cocoa Touch Class** template. Name it **Checklist** and make it a subclass of **NSObject**. (Also make sure that the language is set to **Swift**.)

This adds the file Checklist.swift to the project.

Just like `CheckListItem`, you're building `Checklist` on top of `NSObject`. As you found out previously, this is a requirement when you need to compare objects (in order to find a list item in an array of lists).

► Give **`Checklist.swift`** a name property:

```
import UIKit

class Checklist: NSObject {
    var name = ""
}
```

Next, you'll give `AllListsViewController` an array that will store these new `Checklist` objects.

► Add a new instance variable to **`AllListsViewController.swift`**:

```
var lists = [Checklist]()
```

This is an array that will hold the `Checklist` objects.

Note: You can also write the above as follows:

```
var lists = Array<Checklist>()
```

The version with the square brackets is what's known as *syntactic sugar* for the complete notation, which is `Array<type of the objects to put in the array>`.

You will see both forms used in Swift programs and they do exactly the same thing. Because arrays are used a lot, the designers of Swift included the handy shorthand with the square brackets.

As a first step, you will fill this new array with test data, which you'll do from `viewDidLoad()` as before. Remember that UIKit automatically invokes this method when the view controller is first loaded.

Dummy data

In **`AllListsViewController.swift`** you could add the following to `viewDidLoad()` (don't actually add it just yet, just read along with the description):

```
// 1
var list = Checklist()
list.name = "Birthdays"
lists.append(list)

// 2
list = Checklist()
```

```
list.name = "Groceries"
lists.append(list)

list = Checklist()
list.name = "Cool Apps"
lists.append(list)

list = Checklist()
list.name = "To Do"
lists.append(list)
```

You've seen something very much like it a while ago when you added the fake test data to `ChecklistViewController`. Here is what it does step-by-step:

1. Create a new `Checklist` object, give it a name, and add it to the array.
2. You create three more `Checklist` objects. Because you declared the local variable `list` as `var` instead of `let`, you can re-use it.

Notice how this is performing the same two steps for every new `Checklist` object you're creating?

```
list = Checklist()
list.name = "Name of the checklist"
```

It seems likely that every `Checklist` you'll ever make will also have a name. You can make this a requirement by writing your own `init` method that takes the name as a parameter. Then you can simply write:

```
list = Checklist(name: "Name of the checklist")
```

► Go to **Checklist.swift** and add the new `init` method:

```
init(name: String) {
    self.name = name
    super.init()
}
```

This initializer takes one parameter, `name`, and places it into the property called `name`.

Notice that while the parameter and property are both named `name` - they are two distinct entities. So, you use `self.name` to refer to the property (or instance variable, if you prefer that term).

If you used this code instead:

```
init(name: String) {
    name = name
    super.init()
}
```

Then Swift wouldn't understand that the first name referred to the property.

To disambiguate, you use `self`. Recall that `self` refers to the object that you're in, so `self.name` means the name variable of the current `Checklist` object.

► Go back to **AllListsViewController.swift** and add the following code to the end of `viewDidLoad()`, for real this time:

```
override func viewDidLoad() {  
    // Add placeholder data  
    var list = Checklist(name: "Birthdays")  
    lists.append(list)  
  
    list = Checklist(name: "Groceries")  
    lists.append(list)  
  
    list = Checklist(name: "Cool Apps")  
    lists.append(list)  
  
    list = Checklist(name: "To Do")  
    lists.append(list)  
}
```

That's a bit shorter than what I showed you before, and it guarantees that new `Checklist` objects will now always have their name property filled in.

Note that you don't write:

```
var list = Checklist.init(name: "Birthdays")
```

Even though the method is named `init`, it's not a regular method. Initializers are only used to construct new objects and you write that as:

```
var object = ObjectName(parameter1: value1, parameter2: value2, . . .)
```

Depending on the parameters that you specified, Swift will locate the corresponding `init` method and call that.

Clear? Great! Let's continue building the All Lists screen.

Display data in table view

► Change the `tableView(_:numberOfRowsInSection:)` method to return the number of objects in the new array:

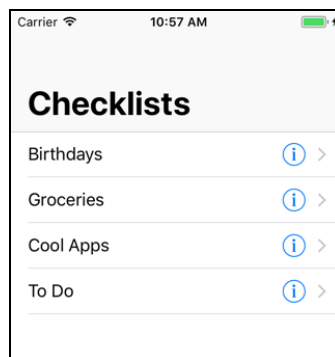
```
override func tableView(_ tableView: UITableView,  
    numberOfRowsInSection section: Int) -> Int {  
    return lists.count  
}
```

► Finally, change `tableView(_:cellForRowAt:)` to fill in the cells for the rows:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    let cell = makeCell(for: tableView)
    // Update cell information
    let checklist = lists[indexPath.row]
    cell.textLabel!.text = checklist.name
    cell.accessoryType = .detailDisclosureButton

    return cell
}
```

► Run the app. It should look like this:



The table view shows Checklist objects

It has a table view with cells representing Checklist objects. The rest of the screen doesn't do much yet, but it's a start.

The many ways to make table view cells

Creating a new table view cell in `AllListsViewController` is a little more involved than how it was done in `ChecklistViewController`. There you just did the following to obtain a new table view cell:

```
let cell = tableView.dequeueReusableCell(
    withIdentifier: "CheckListItem", for: indexPath)
```

But here you have a whole chunk of code to accomplish the same:

```
let cellIdentifier = "Cell"
if let cell =
    tableView.dequeueReusableCell(
        withIdentifier: cellIdentifier) {
    return cell
} else {
    return UITableViewCell(style: .default,
        reuseIdentifier: cellIdentifier)
}
```

The call to `dequeueReusableCell(withIdentifier:)` is still there, except that previously the storyboard had a prototype cell with that identifier and now it doesn't.

If the table view cannot find a cell to re-use (and it won't until it has enough cells to fill the entire visible area), this method will return `nil` and you have to create your own cell by hand. That's what happens in the `else` branch.

There are actually two versions of `dequeueReusableCell`, one with an extra `for` parameter that takes an `IndexPath`, and one without. Here you're using the one without. The difference is that `dequeueReusableCell(withIdentifier:for:)` only works with prototype cells. If you tried to use it here, it would crash the app.

There are four ways that you can make table view cells:

1. Using prototype cells. This is the simplest and quickest way. You did this in `ChecklistViewController`.
2. Using static cells. You did this for the Add/Edit Item screen. Static cells are limited to screens where you know in advance which cells you'll have. The big advantage with static cells is that you don't need to provide any of the data source methods (`cellForRowAt` etc.).
3. Using a *nib* file. A nib (also known as a XIB) is like a mini storyboard that only contains a single customized `UITableViewCell` object. This is very similar to using prototype cells, except that you can do it outside of a storyboard.
4. By hand, what you did above. This is how you were supposed to do it in the early days of iOS. Chances are, you'll run across code examples that do it this way, especially from older articles and books. It's a bit more work, but also offers you the most flexibility.

When you create a cell by hand, you specify a certain **cell style**, which gives you a cell with a preconfigured layout that already has labels and an image view.

For the All Lists scene you're using the "Default" style. Later on you'll switch it to "Subtitle", which gives the cell a second, smaller label below the main label.

Using standard cell styles means you don't have to design your own cell layout. For many apps these standard layouts are sufficient, so that saves you some work.

Prototype cells and static cells can also use these standard cell styles. The default style for a prototype or static cell is "Custom", which requires you to use your own labels, but you can change that to one of the built-in styles via Interface Builder.

And finally, a gentle warning: Sometimes I see code that creates a new cell for every row rather than trying to reuse cells. Don't do that! Always ask the table view first whether it has a cell available that can be recycled, using one of the `dequeueReusableCell` methods.

Creating a new cell for each row will cause your app to slow down, as object creation is slower than simply re-using an existing object. Creating all these new objects also takes up more memory, a precious commodity on mobile devices. For the best performance, reuse those cells!

View the checklists

Right now, the data model consists of the `lists` array from `AllListsViewController` that contains a handful of `Checklist` objects. There is also a separate `items` array in `ChecklistViewController` with `ChecklistItem` objects.

You may have noticed that when you tap the name of a list, the Checklist screen slides into view but it currently always shows the same to-do items, regardless of which list you tapped on.

Each checklist should really have its own to-do items. You'll work on that later on, as this requires a significant change to the data model.

As a start, let's set the title of the Checklist screen to reflect the chosen checklist.

Set the title of the screen

► Add a new instance variable to **ChecklistViewController.swift**:

```
var checklist: Checklist!
```

I'll explain why the exclamation mark is necessary in a moment.

► Change `viewDidLoad` in **ChecklistViewController.swift** to:

```
override func viewDidLoad() {  
    title = checklist.name  
}
```

This changes the title of the screen, which is shown in the navigation bar, to the name of the `Checklist` object.

You'll pass the necessary `Checklist` object to `ChecklistViewController` when the segue is performed.

► In **AllListsViewController.swift**, update `tableView(_:didSelectRowAt:)` to the following:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    let checklist = lists[indexPath.row]
    performSegue(withIdentifier: "ShowChecklist",
                  sender: checklist)
}
```

As before, you use `performSegue()` to start the segue. This method has a `sender` parameter that you previously set to `nil`. Now you'll use it to send along the `Checklist` object from the row that the user tapped on.

You can put anything you want into `sender`. If the segue is performed by the storyboard (rather than manually like you do here) then `sender` will refer to the control that triggered it, for example, the `UIBarButtonItem` object for the Add button, or the `UITableViewCell` for a row in the table.

But because you start this particular segue by hand, you can put whatever is most convenient into `sender`.

Putting the `Checklist` object into the `sender` parameter doesn't pass it to `ChecklistViewController` yet. That happens in "prepare-for-segue", which you still need to add for this view controller.

► Add the `prepare(for:sender:)` method to **AllListsViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "ShowChecklist" {
        let controller = segue.destination
                                as! ChecklistViewController
        controller.checklist = sender as! Checklist
    }
}
```

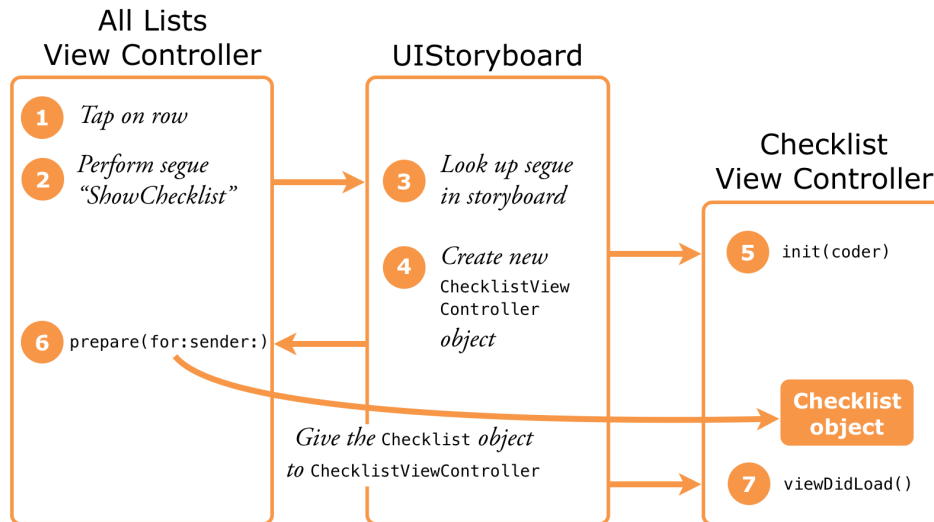
You've seen this method before. `prepare(for:sender:)` is called right before the segue happens. Here you get a chance to set the properties of the new view controller before it becomes visible.

Inside `prepare(for:sender:)`, you need to pass the `ChecklistViewController` the `Checklist` object from the row that the user tapped. That's why you put that object in the `sender` parameter earlier.

(You could have temporarily stored the `Checklist` object in an instance variable instead, but passing it along in the `sender` parameter is much easier and cleaner.)

All of this happens a short time after `ChecklistViewController` is instantiated but just before `ChecklistViewController`'s view is loaded. That means its `viewDidLoad()` method is called after `prepare(for:sender:)`.

At this point, the view controller's `checklist` property is set to the `Checklist` object from sender, and `viewDidLoad()` can set the title of the screen accordingly.



The steps involved in performing a segue

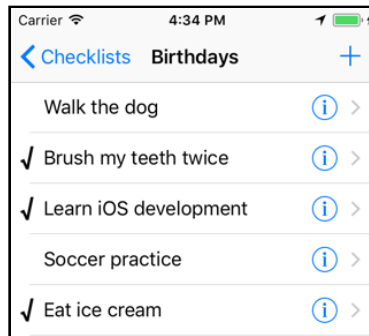
This sequence of events is why the `checklist` property is declared as `Checklist!` with an exclamation point. That allows its value to be temporarily `nil` until `viewDidLoad()` happens.

`nil` is normally not an allowed value for non-optional variables in Swift but by using the `!` you override that.

Does this sound an awful lot like optionals? The exclamation point turns `checklist` into a special kind of optional. It's very similar to optionals with a question mark, but you don't have to write `if let` to unwrap it.

Such *implicitly unwrapped* optionals should be used sparingly and with care, as they do not have any of the anti-crash protection that normal optionals do.

► Run the app and notice that when you tap the row for a checklist, the next screen properly displays the checklist title.



The name of the chosen checklist now appears in the navigation bar

Note that passing the Checklist object to the ChecklistViewController does not make a copy of it.

You only pass the view controller a *reference* to that object – any changes the user makes to that Checklist object are also seen by AllListsViewController.

Both view controllers have access to the exact same Checklist object. You'll use that to your advantage later in order to add new ChecklistItems to the selected Checklist.

Type Casts

In `prepare(for:sender:)` you do this:

```
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    . . .
    controller.checklist = sender as! Checklist
    . . .
}
```

What is that `as! Checklist` bit?

If you've been paying attention – of course you have! – then you've seen this “as something” used quite a few times now. This is known as a *type cast*.

A type cast tells Swift to interpret a value as having a different data type.

(It's the opposite of what happens to certain actors in the movies. For them, typecasting results in always playing the same character; in Swift, a type cast actually changes the character of an object.)

Here, `sender` has type `Any?`, meaning that it can be any sort of object: a `UIBarButtonItem`, a `UITableViewCell`, or in this case, a `Checklist`. Thanks to the question mark it can even be `nil`.

But the `controller.checklist` property always expects a `Checklist` object – it wouldn't know what to do with a `UITableViewCell`... Hence, Swift demands that you only put `Checklist` objects into the `checklist` property.

By writing `sender as! Checklist`, you tell Swift that it can safely treat `sender` as a `Checklist` object and to force unwrap it (since the `sender` is an optional).

Another example of a typecast is:

```
let controller = segue.destination as! ChecklistViewController
```

The segue's `destination` property refers to the view controller on the receiving end of the segue. But obviously the engineers at Apple could not predict beforehand that we would call it `ChecklistViewController`.

So you have to cast it from its generic type (`UIViewController`) to the specific type used in this app (`ChecklistViewController`) before you can access any of its properties.

Similar to the `as!` type cast, there is also `as?` with a question mark. This is for casting optionals, or when the type cast is allowed to fail. You'll see some examples of that later.

Don't worry if any of this goes over your head right now. You'll see plenty more examples of type casting in action.

The main reason you need all these type casts is for interoperability with the iOS frameworks that are written in Objective-C. Swift is less forgiving about types than Objective-C and requires you to be much more explicit about types.

Manage checklists

Let's quickly add the Add Checklist / Edit Checklist screen. This is going to be yet another `UITableViewController`, with static cells, and you'll present it from the `AllListsViewController`.

If the previous sentence made perfect sense to you, then you're getting the hang of this!

Add the view controller

➤ Add a new file to the project, **ListDetailViewController.swift**. You can use the **Swift File** template for this since you'll be adding the complete view controller implementation by hand.

► Add the following to **ListDetailViewController.swift**:

```
import UIKit

protocol ListDetailViewControllerDelegate: class {
    func listDetailViewControllerDidCancel(
        _ controller: ListDetailViewController)

    func listDetailViewController(
        _ controller: ListDetailViewController,
        didFinishAdding checklist: Checklist)

    func listDetailViewController(
        _ controller: ListDetailViewController,
        didFinishEditing checklist: Checklist)
}

class ListDetailViewController: UITableViewController,
    UITextFieldDelegate {
    @IBOutlet weak var textField: UITextField!
    @IBOutlet weak var doneBarButton: UIBarButtonItem!

    weak var delegate: ListDetailViewControllerDelegate?

    var checklistToEdit: Checklist?
}
```

I simply took the contents of **ItemDetailViewController.swift** and changed the names. Also, instead of a property for a `CheckListItem` you're now dealing with a `Checklist`.

► Add the `viewDidLoad()` method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Disable large titles for this view controller
    navigationItem.largeTitleDisplayMode = .never

    if let checklist = checklistToEdit {
        title = "Edit Checklist"
        textField.text = checklist.name
        doneBarButton.isEnabled = true
    }
}
```

This changes the title of the screen if the user is editing an existing checklist, and it puts the checklist's name into the text field already.

► Also add the `viewWillAppear()` method to pop up the keyboard:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    textField.becomeFirstResponder()
}
```

The Cancel and Done buttons

➤ Add the action methods for the Cancel and Done buttons:

```
// MARK:- Actions
@IBAction func cancel() {
    delegate?.listDetailViewControllerDidCancel(self)
}

@IBAction func done() {
    if let checklist = checklistToEdit {
        checklist.name = textField.text!
        delegate?.listDetailViewController(self,
                                           didFinishEditing: checklist)
    } else {
        let checklist = Checklist(name: textField.text!)
        delegate?.listDetailViewController(self,
                                           didFinishAdding: checklist)
    }
}
```

This should look familiar as well. It's essentially the same as what the Add/Edit Item screen does.

To create the new Checklist object in done(), you use its init(name:) method and pass the contents of textField.text as the name parameter.

You cannot write this the way you did for ChecklistItems – this won't work:

```
let checklist = Checklist()
checklist.name = textField.text!
```

Because Checklist does not have an init() method that takes no parameters, writing Checklist() results in a compiler error. It only has an init(name:) method, and you must always use that initializer to create new Checklist objects.

Other functionality

➤ Also make sure the user cannot select the table cell with the text field:

```
// MARK:- TableView Delegates
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    return nil
}
```

➤ And finally, add the text field delegate method that enables or disables the Done button depending on whether the text field is empty or not.

```
// MARK:- UITextField 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)
    doneBarButton.isEnabled = !newText.isEmpty
    return true
}

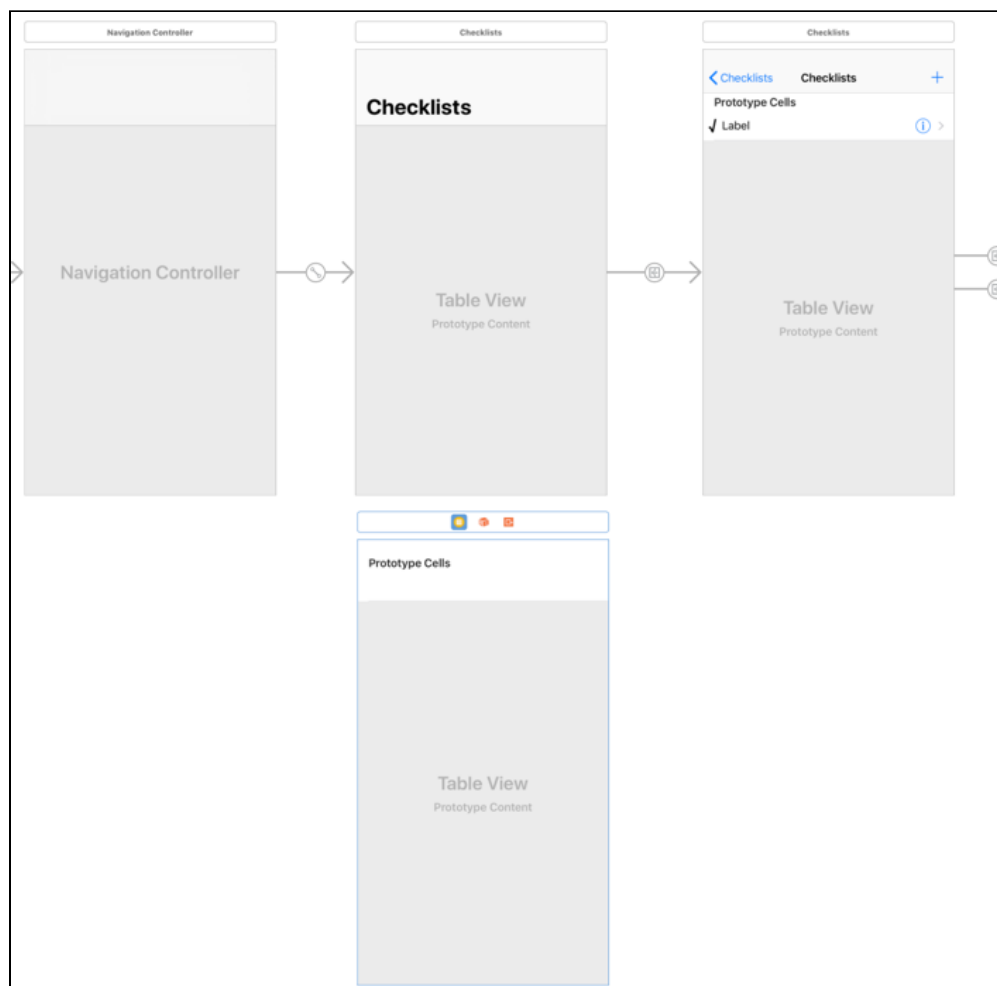
```

Again, this is the same as what you did in `ItemDetailViewController`.

Let's create the user interface for this new view controller in Interface Builder.

The storyboard

► Open the storyboard. Drag a new **Table View Controller** from the Object Library on to the canvas and move it below the other view controllers.



Adding a new table view controller to the canvas

- Select the new Table View Controller and go to the **Identity inspector**. Change its class to **ListDetailViewController**.
- **Control-drag** from the yellow circle at the top of the All Lists scene to the new scene. Select **Show** from the Manual Segue section of the popup menu.
- Add a Navigation Item to the new scene.
- Change the navigation bar title from “Title” to **Add Checklist**. (The new scene should now appear as Add Checklist scene in the Document Outline.)
- Select the Navigation Item and set **Large Title** in the Attributes inspector to **Never**.
- Add **Cancel** and **Done** bar button items and hook them up to the action methods in the Add Checklist scene. Also connect the Done button to the **doneBarButton** outlet and uncheck its **Enabled** option.

Remember, you can Control-drag from a button to the view controller to connect it to an action method. To connect an outlet, do it the other way around: Control-drag from the view controller to the button.

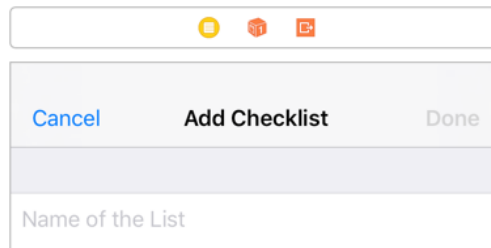
Tip: My Xcode acted a bit buggy and wouldn’t let me drop the bar buttons on the navigation bar. If this happens to you too, drop them on the navigation item – now called Add Checklist – in the Document Outline. You can also Control-drag in the Document Outline to make the connections to the actions and the outlet.

- Change the table view to **Static Cells**, style **Grouped**. You only need one cell, so remove the bottom two.
- Drop a new **Text Field** on to the cell. Here are the configuration options:
 - Border Style: none
 - Font size: 17
 - Placeholder text: **Name of the List**
 - Adjust to Fit: disabled
 - Capitalization: Sentences
 - Return Key: Done
 - Auto-enable Return key: check
- Control-drag from the view controller to the Text Field and connect it to the **textField** outlet.

➤ Then Control-drag the other way around, from the Text Field back to the view controller, and choose **delegate** under **Outlets**. Now the view controller is the delegate for the text field.

➤ Connect the text field's **Did End on Exit** event to the **done** action on the view controller.

This completes setting up the new view controller to be the Add / Edit Checklist screen:



The finished design of the ListDetailViewController

Connect the view controllers

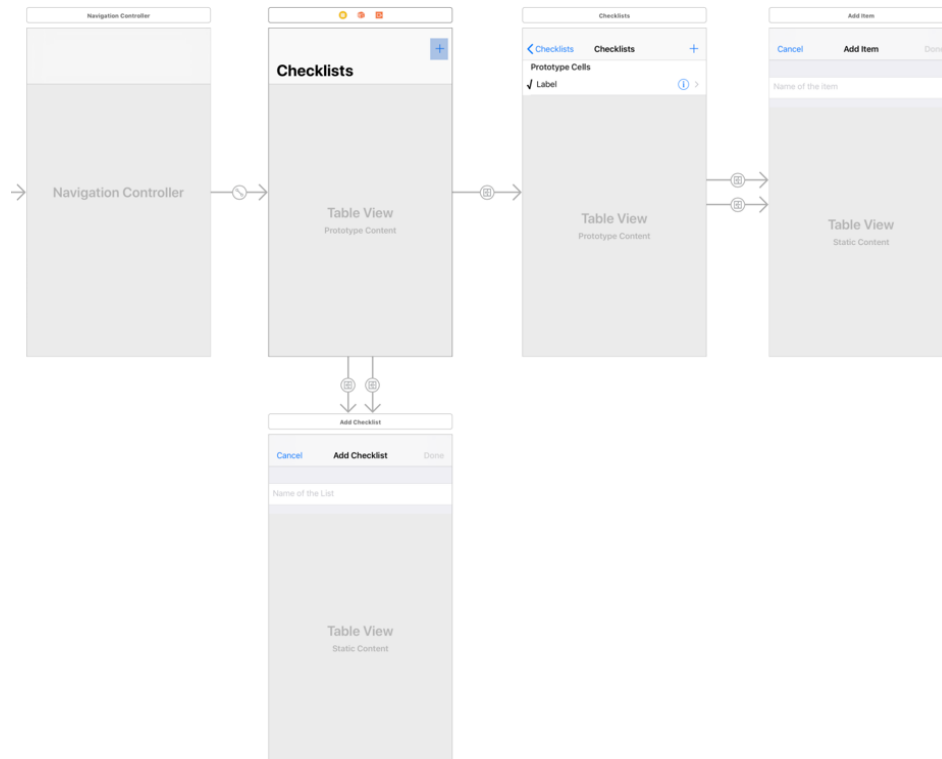
➤ Go to the **All Lists** scene (the one titled “Checklists”) and drag a **Bar Button Item** on to its navigation bar. Change it to an **Add** button.

➤ **Control-drag** from this new bar button to the Add Checklist scene below to add a new **Show** segue.

➤ Click on the new segue and name it **AddChecklist**.

➤ Click on the other segue (the one not connected the the Add button) and name it **EditChecklist**.

Your storyboard should now look something like this:



The full storyboard: 1 navigation controller, 4 table view controllers

Set up the delegates

Almost there. You still have to make the `AllListsViewController` the delegate for the `ListDetailViewController` and then you're done. Again, it's very similar to what you did before.

- Declare the All Lists view controller to conform to the delegate protocol by adding `ListDetailViewControllerDelegate` to its class line.

You do this in **`AllListsViewController.swift`**:

```
class AllListsViewController: UITableViewController,
                             ListDetailViewControllerDelegate {
```

- Still in **`AllListsViewController.swift`**, extend `prepare(for:sender:)` to:

```
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "ShowCheckList" {
        . . .
    } else if segue.identifier == "AddCheckList" {
        let controller = segue.destination
                               as! ListDetailViewController
        controller.delegate = self
    }
}
```

The first if doesn't change. You've added a second if for the new "AddChecklist" segue that you just defined in the storyboard. As before, you look for the view controller and set its delegate property to self.

► Next, implement the following delegate methods in **AllListsViewController.swift**:

```
// MARK:- List Detail View Controller Delegates
func listDetailViewControllerDidCancel(
    _ controller: ListDetailViewController) {
    navigationController?.popViewController(animated: true)
}

func listDetailViewController(
    _ controller: ListDetailViewController,
    didFinishAdding checklist: Checklist) {
    let newRowIndex = lists.count
    lists.append(checklist)

    let indexPath = IndexPath(row: newRowIndex, section: 0)
    let indexPaths = [indexPath]
    tableView.insertRows(at: indexPaths, with: .automatic)

    navigationController?.popViewController(animated: true)
}

func listDetailViewController(
    _ controller: ListDetailViewController,
    didFinishEditing checklist: Checklist) {
    if let index = lists.index(of: checklist) {
        let indexPath = IndexPath(row: index, section: 0)
        if let cell = tableView.cellForRow(at: indexPath) {
            cell.textLabel!.text = checklist.name
        }
    }
    navigationController?.popViewController(animated: true)
}
```

These methods are called when the user presses Cancel or Done inside the new Add/Edit Checklist screen.

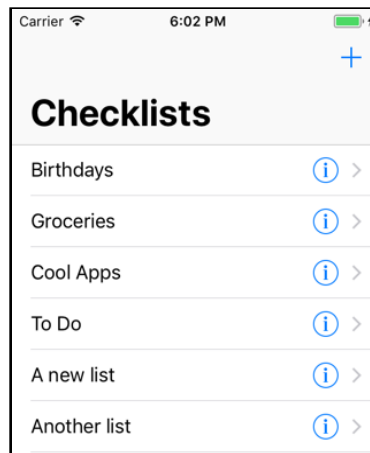
None of this code should surprise you. It's exactly what you did before but now for the `ListDetailViewController` and `Checklist` objects.

► Also add the table view data source method that allows the user to delete checklists:

```
override func tableView(
    _ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath) {
    lists.remove(at: indexPath.row)

    let indexPaths = [indexPath]
    tableView.deleteRows(at: indexPaths, with: .automatic)
}
```

► Run the app. Now you can add new checklists and delete them again:



Adding new lists

Note: If the app crashes, then go back and make sure you made all the connections properly in Interface Builder. It's really easy to miss just one tiny thing, but even the tiniest of mistakes can bring the app crashing down in flames...

You can't edit the names of existing lists yet. That requires one last addition to the code.

To bring up the Edit Checklist screen, the user taps the blue accessory button. In the `ChecklistViewController` that triggered a segue. You could use a segue here too. (In fact, if you want to go that route, you've already set up a segue named "EditChecklist" on the storyboard that you can use for this purpose.)

But I want to show you another way :]

This time you're not going to use a segue at all, but load the new view controller by hand from the storyboard. Just because you can.

Load a view controller via code

► Add the following `tableView(_:accessoryButtonTappedForRowWith:)` method to **AllListsViewController.swift**. This method comes from the table view delegate protocol and the name is hopefully obvious enough for you to guess what it does.

```
override func tableView(_ tableView: UITableView,
    accessoryButtonTappedForRowWith indexPath: IndexPath) {

    let controller = storyboard!.instantiateViewController(
        withIdentifier: "ListDetailViewController")
        as! ListDetailViewController
```

```

controller.delegate = self

let checklist = lists[indexPath.row]
controller.checklistToEdit = checklist

navigationController?.pushViewController(controller,
                                       animated: true)
}

```

In this method, you create the view controller object for the Add/Edit Checklist screen and push it on to the navigation stack. This is roughly equivalent to what a segue would do behind the scenes. The view controller is embedded in a storyboard and you have to ask the storyboard object to load it.

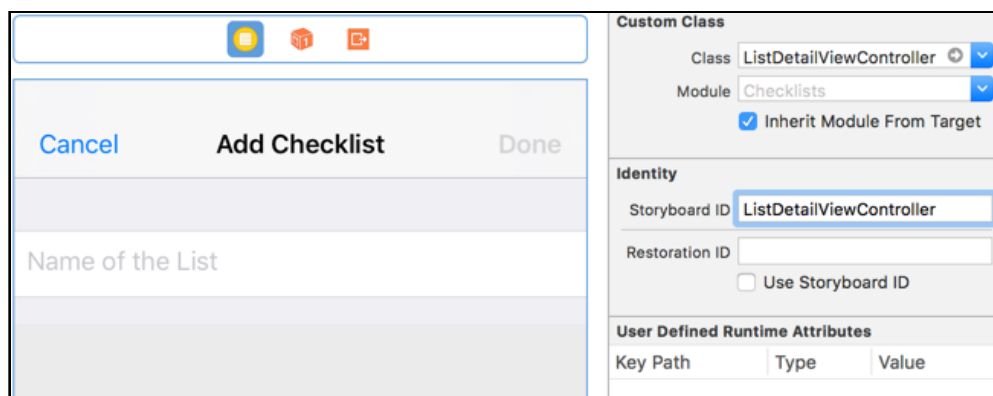
Where did you get that storyboard object? As it happens, each view controller has a storyboard property that refers to the storyboard the view controller was loaded from. You can use that property to do all kinds of things with the storyboard, such as instantiating other view controllers.

The storyboard property is optional because view controllers are not always loaded from a storyboard. But this one is, which is why you can use `!` to *force unwrap* the optional. It's like using `if let`, but because you can safely assume storyboard will not be `nil` in this app, you don't have to unwrap it inside an `if` statement.

The call to `instantiateViewController(withIdentifier:)` takes an identifier string, `ListDetailViewController`. That is how you ask the storyboard to create the new view controller. In your case, this will be the `ListDetailViewController`.

You still have to set this identifier on the navigation controller; otherwise the storyboard won't be able to find it. (And if you try to run the app without setting the identifier, it will crash.)

► Open the storyboard and select the List Detail View Controller. Go to the **Identity inspector** and set **Storyboard ID** to **ListDetailViewController**:



Setting the storyboard identifier

- That should do the trick. Run the app and tap some detail disclosure buttons.
(If the app crashes, make sure the storyboard is saved before you press Run.)

Are you still with me?

If at this point your eyes are glazing over and you feel like giving up: don't.

Learning new things is hard and programming doubly so. Set the book aside, sleep on it, and come back in a few days.

Chances are that in the mean time you'll have an a-ha! moment where the thing that didn't make any sense suddenly becomes clear as day.

If you have specific questions, join us on the forums. I usually check in a few times a day to help people out and so do many members of our community. Don't be embarrassed to ask for help! forums.raywenderlich.com

You can find the project files for the app up to this point under **29 - Lists** in the Source Code folder.

Chapter 30: Improved Data Model

By Fahim Farook and Matthijs Hollemans

Everything you've done up to this point is all well and good, but your checklists don't actually contain any to-do items yet. Or rather, if you select a checklist, you see the same old items for every list! There is no connection between the selected list and the items displayed for that list.

It's time for you to fix that.

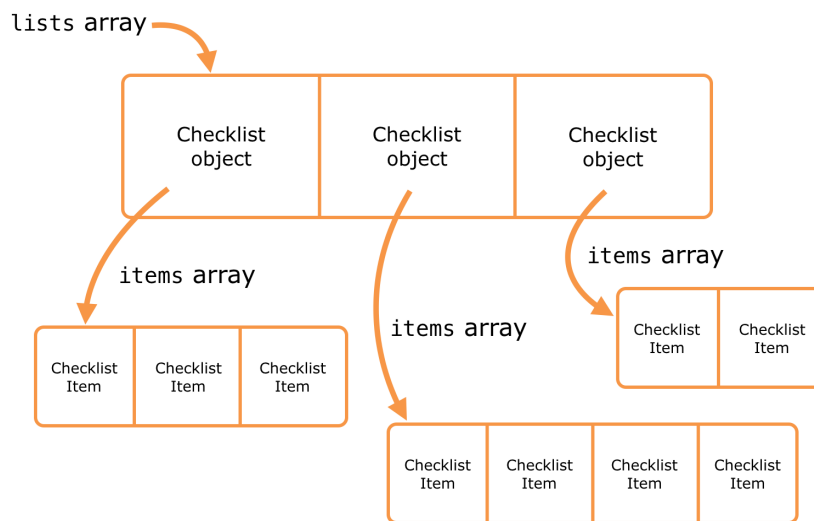
This chapter covers the following steps:

- **The new data model:** Update the data model so that the to-do items for a list are saved along with the list.
- **Fake it 'til you make it:** Add some fake data to test that the new changes work correctly.
- **Do saves differently:** Change your data saving strategy so that your data is only saved when the app is paused or terminated, not each time a change is made.
- **Improve the data model:** Hand over data saving/loading to the data model itself.

The new data model

So far, the list of to-do items and the actual checklists have been separate from each other.

Let's change the data model to look like this:



Each Checklist object has an array of ChecklistItem objects

There will still be the `lists` array that contains all the `Checklist` objects, but each of these `Checklist` instances will have its own array of `ChecklistItem` objects.

The to-do item array

► Add a new property to **Checklist.swift**:

```
class Checklist: NSObject {
    var name = ""
    var items = [ChecklistItem]()    // add this line
    . . .
}
```

This creates a new empty array that can hold `ChecklistItem` objects and assigns it to the `items` property.

If you're a stickler for completeness, you can also write it as follows:

```
var items: [ChecklistItem] = [ChecklistItem]()
```

I personally don't like this way of declaring variables because it violates the "DRY" principle – Don't Repeat Yourself. Fortunately, thanks to Swift's type inference, you can save yourself some keystrokes.

Another way you'll see it written sometimes is:

```
var items: [CheckListItem] = []
```

The notation `[]` means: make an empty array of the specified type. There is no type inference at play there since you have to specify the type explicitly. If you don't specify a type and write the above line as:

```
var items = []
```

You will get an error since the compiler cannot determine the type of the array. That makes sense, right? Regardless of the way you choose to write it, the `Checklist` object now contains an array of `CheckListItem` objects. Initially, that array is empty.

Pass the array

Earlier you fixed `prepare(for:sender:)` in `AllListsViewController.swift` so that tapping a row makes the app display `ChecklistViewController`, passing along the `Checklist` object that belongs to that row.

Currently `ChecklistViewController` still gets the `CheckListItem` objects that it displays from its own private `items` array. You will change that so it reads from the `items` array inside the `Checklist` object instead.

- Remove the `items` instance variable from **`ChecklistViewController.swift`**.
- Then make the following changes in this source file. Anywhere it says `items` you change it to say `checklist.items` instead.

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return checklist.items.count
}
```

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {

    . . .
    let item = checklist.items[indexPath.row]
    . . .
}
```

```
override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {

    . . .
    let item = checklist.items[indexPath.row]
    . . .
}
```

```

override func tableView(
    _ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath) {

    checklist.items.remove(at: indexPath.row)
    . . .
}

```

```

func itemDetailViewController(
    _ controller: ItemDetailViewController,
    didFinishAdding item: ChecklistItem) {
    let newRowIndex = checklist.items.count
    checklist.items.append(item)
    . . .
}

```

```

func itemDetailViewController(
    _ controller: ItemDetailViewController,
    didFinishEditing item: ChecklistItem) {

    if let index = checklist.items.index(of: item) {
        . . .
    }
}

```

```

override func prepare(for segue: UIStoryboardSegue,
    sender: Any?) {

    . . .
    controller.itemToEdit = checklist.items[indexPath.row]
    . . .
}

```

➤ Delete the following methods from **ChecklistViewController.swift**. (Tip: You may want to set aside the code from these methods in a temporary file somewhere; shortly you'll be using them again in a slightly modified form.)

- func documentsDirectory()
- func dataFilePath()
- func saveChecklistItems()
- func loadChecklistItems()

You added these methods to load and save the checklist items from a file. That is no longer the responsibility of this view controller. It is better from a design perspective for the `Checklist` object to do that.

Loading and saving data model objects really belongs in the data model itself, rather than in a controller.

But before you get to that, let's first test whether these changes were successful. Xcode is throwing up a few errors because you still call `saveChecklistItems()` and

`loadChecklistItems()` from several places in the code. You should remove those lines, as you will soon be saving the items from a different place.

- Remove the lines that call `saveChecklistItems()` and `loadChecklistItems()`.
- Press **⌘+B** to make sure the app builds without errors.

Fake it 'til you make it

Let's add some fake data to the various `Checklist` objects so that you can test whether this new design actually works.

Add fake to-do data

In `AllListsViewController`'s `viewDidLoad()` you already put fake `Checklist` objects into the `lists` array. It's time to add something new to this method.

- Add the following to the bottom of `AllListsViewController.swift`'s `viewDidLoad()`:

```
// Add placeholder item data
for list in lists {
    let item = ChecklistItem()
    item.text = "Item for \(list.name)"
    list.items.append(item)
}
```

This introduces something you haven't seen before: the `for in` statement. Like `if`, this is a special language construct.

Programming language constructs

For the sake of review, let's go over the programming language stuff you've already seen. Most modern programming languages offer at least the following basic building blocks:

- The ability to remember values by storing things into variables. Some variables are simple, such as `Int` and `Bool`. Others can store objects (`ChecklistItem`, `UIButton`) or even collections of objects (`Array`).
- The ability to read values from variables and use them for basic arithmetic (multiply, add) and comparisons (greater than, not equals, etc).
- The ability to make decisions. You've already seen the `if` statement, but there is also a `switch` statement that is shorthand for `if` with many `else if`s.

- The ability to group functionality into units such as methods and functions. You can call those methods and receive back a result value that you can then use in further computations.
- The ability to bundle functionality (methods) and data (variables) together into objects.
- The ability to execute a one or more lines of code inside a `do` block and to catch any errors thrown via a `try` statement. (Or, to simply bypass the `do` block by using a `try?` statement instead.)
- The ability to repeat a set of statements more than once. This is what the `for in` statement does. There are other ways to perform repetitions as well: `while` and `repeat`. Endlessly repeating things is what computers are good at.

Everything else is built on top of these building blocks. You've seen most of these already, but repetitions (or **loops** in programmer talk) are new.

If you grok the concepts from this list, you're well on your way to becoming a software developer. And if not, well, just hang in there!

Let's go through that `for` loop line-by-line:

```
for list in lists {  
    . . .  
}
```

This means the following: for every `Checklist` object in the `lists` array, perform the statements between the curly braces.

The first time through the loop, the temporary `list` variable will hold a reference to the Birthdays checklist, as that is the first `Checklist` object that you created and added to the `lists` array.

Inside the loop you do:

```
let item = ChecklistItem()  
item.text = "Item for \(list.name)"  
list.items.append(item)
```

This should be familiar. You first create a new `ChecklistItem` object. Then you set its `text` property to "Item for Birthdays" because the `\(...)` placeholder gets replaced with the name of the `Checklist` object, `list.name`, which is "Birthdays".

Finally, you add this new `ChecklistItem` to the Birthdays `Checklist` object, or rather, to its `items` array.

That concludes the first pass through this loop. Now the `for in` statement will look at the `lists` array again and sees that there are three more `Checklist` objects in that array. So it puts the next one, `Groceries`, into the `list` variable and the process repeats.

This time the text is “Item for Groceries”, which is put into its own `ChecklistItem` object that goes into the `items` array of the `Groceries Checklist` object.

After that, the loop adds a new `ChecklistItem` with the text “Item for Cool Apps” to the `Cool Apps checklist`, and “Item for To Do” to the `To Do checklist`.

Then there are no more objects left to look at in the `lists` array and the loop ends.

Using loops will often save you a lot of time. You could have written this code as follows:

```
var item = ChecklistItem()
item.text = "Item for Birthdays"
lists[0].items.append(item)

item = ChecklistItem()
item.text = "Item for Groceries"
lists[1].items.append(item)

item = ChecklistItem()
item.text = "Item for Cool Apps"
lists[2].items.append(item)

item = ChecklistItem()
item.text = "Item for To Do"
lists[3].items.append(item)
```

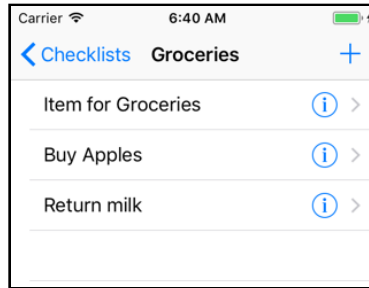
That’s very repetitive, which is a good sign it’s better to use a loop. Imagine if you had 100 `Checklist` objects... would you be willing to copy-paste that code a hundred times? I’d rather use a loop.

Most of the time you won’t even know in advance how many objects you’ll have, so it’s impossible to write it all out by hand. By using a loop you don’t need to worry about that. The loop will work just as well for three items as for three hundred.

As you can imagine, loops and arrays work quite well together.

► Run the app. You’ll see that each checklist now has its own set of items.

Play with it for a minute, remove items, add items, and verify that each list indeed is completely separate from the others.



Each Checklist now has its own items

The new load/save code

Let's put the load/save code back in. This time you'll make `AllListsViewController` do the loading and saving.

► Add the following to `AllListsViewController.swift` (you may want to copy this from that temporary file, but be sure to make the changes mentioned in the comments):

```
func documentsDirectory() -> URL {
    let paths = FileManager.default.urls(for: .documentDirectory,
                                         in: .userDomainMask)
    return paths[0]
}

func dataFilePath() -> URL {
    return documentsDirectory().appendingPathComponent(
        "Checklists.plist")
}

// this method is now called saveChecklists()
func saveChecklists() {
    let encoder = PropertyListEncoder()
    do {
        // You encode lists instead of "items"
        let data = try encoder.encode(lists)
        try data.write(to: dataFilePath(),
                      options: Data.WritingOptions.atomic)
    } catch {
        print("Error encoding item array!")
    }
}

// this method is now called loadChecklists()
func loadChecklists() {
    let path = dataFilePath()
    if let data = try? Data(contentsOf: path) {
        let decoder = PropertyListDecoder()
        do {
            // You decode to an object of [Checklist] type to lists
            lists = try decoder.decode([Checklist].self, from: data)
        } catch {
            print("Error decoding item array!")
        }
    }
}
```

```
}  
}
```

This is mostly identical to what you had before in `ChecklistViewController`, except that you load and save the `lists` array instead of the `items` array. Note that the decode type is now `[Checklist].self` instead of `[ChecklistItem].self`. Also, the names of the methods changed slightly.

➤ Change `viewDidLoad()` to:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    // Enable large titles  
    navigationController?.navigationBar.prefersLargeTitles = true  
    // Load data  
    loadChecklists()  
}
```

This gets rid of the test data you put there earlier and makes the `loadChecklists()` method do all the work.

You also have to make the `Checklist` object support the `Codable` protocol - but that's just a simple change.

➤ Add the `Codable` protocol in **Checklist.swift**:

```
class Checklist: NSObject, Codable {
```

➤ **Important:** Before you run the app, remove the old **Checklists.plist** file from the Simulator's Documents folder.

If you don't, the app will most probably throw up an error message in the Console about the an error decoding because the internal format of the file no longer corresponds to the new data you're loading and saving. This is because the new Swift `Codable` protocol handles data encoding/decoding in a safe fashion.

With older version of this book, where the `Codable` protocol was not available, you had to encode/decode data in a different fashion. That approach used to crash the app if the `Checklists.plist` file was not removed and the data was in a different format.

Weird crashes

When I first wrote this book, I didn't think to remove the `Checklists.plist` file before running the app. That was a mistake, but the app appeared to work fine... until I added a new checklist. At that point the app aborted with a strange error message from `UITableView` that made no sense at all.

I started to wonder whether I tested the code properly. But then I thought of the old file, removed it and ran the app again. It worked perfectly. Just to make sure it was the fault of that file, I put a copy of the old file back and ran the app again. Sure enough, when I tried to add a new checklist it crashed.

The explanation for this kind of error is that somehow the code managed to load the old file, even though its format no longer corresponded to the new data model. This put the table view into a bad state. Any subsequent operations on the table view caused the app to crash.

You'll run into this type of bug every so often, where the crash isn't directly caused by what you're doing but by something that went wrong earlier on. These kinds of bugs can be tricky to solve, because you can't fix them until you find the true cause.

There is a section devoted to debugging techniques towards the end of the book because it's inevitable that you'll introduce bugs in your code. Knowing how to find and eradicate them is an essential skill that any programmer should master – if only to save you a lot of time and aggravation!

- Run the app and add a checklist and a few to-do items.
- Exit the app (with the Stop button) and run it again. You'll see that the list is empty again. All your to-do items are gone!

You can add all the checklists and items you want, but nothing gets saved anymore. What's going on here?

Do saves differently

Previously, you saved the data whenever the user changed something: adding a new item, deleting an item, and toggling a checkmark all caused `Checklists.plist` to be re-saved. That used to happen in `ChecklistViewController`.

However, you just moved the saving logic to `AllListsViewController`. How do you make sure changes to the to-do items get saved now? The `AllListsViewController` doesn't know when a checkmark is toggled on or off.

You could give `ChecklistViewController` a reference to the `AllListsViewController` and have it call its `saveChecklists()` method whenever the user changes something, but that introduces a *child-parent dependency* and you've been trying hard to avoid those (ownership cycles, remember?).

Parents and their children

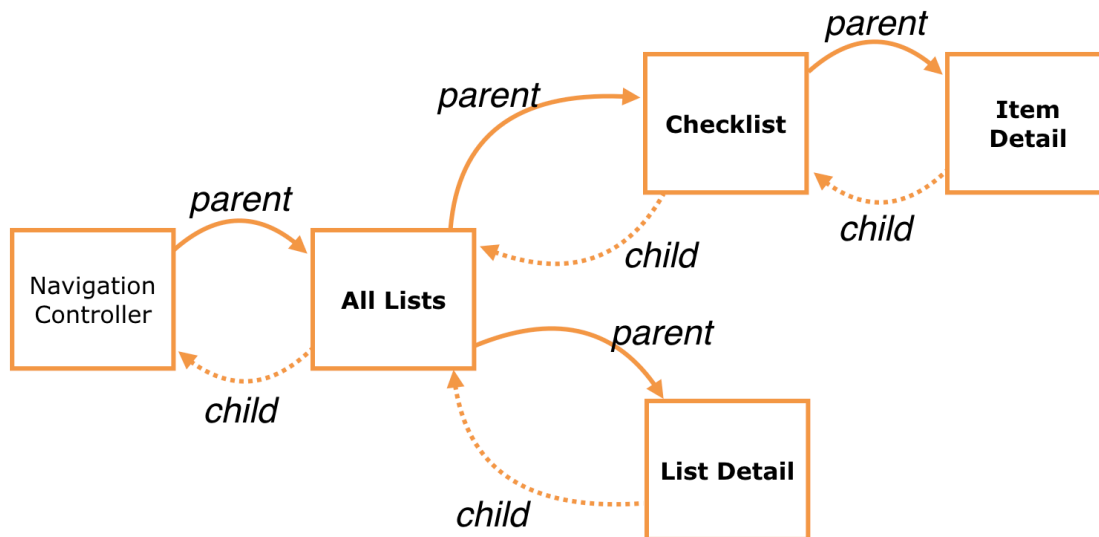
The terms *parent* and *child* are common in software development.

A parent is an object higher up in some hierarchy; a child is an object lower in the hierarchy.

In this case, the “hierarchy” represents the navigation flow between the different screens of the app.

The All Lists screen is the parent of the Checklist screen, because All Lists was “born” first. It creates a new ChecklistViewController “baby” every time the user views the item list for a checklist.

Likewise, All Lists is also the parent of the List Detail screen. The Item Detail screen, however, is the child of the Checklist view controller.



Generally speaking, it’s OK if the parent knows everything about its children, but not the other way around (just like in real life, every parent has horrible secrets they don’t want their kids to know about... or so I’ve been told).

As a result, you don’t want parent objects to be dependent on their child objects, but the other way around is fine. So ChecklistViewController asking AllListsViewController to do things is a big no-no.

The new saving strategy

You may think: ah, I could use a delegate for this. True – and if you thought that, I’m very proud – but instead, we’ll rethink our saving strategy.

Is it really necessary to save changes all the time? While the app is running, the data model sits in working memory and is always up-to-date.

The only time you have to load anything from the file (the long-term storage memory) is when the app first starts up, but never afterwards. From then on you always make the changes to the objects in the working memory.

But when changes are made, the file becomes out-of-date. That is why you save those changes – to keep the file in sync with what is in memory.

The reason you save to a file is so that you can restore the data model in working memory after the app gets terminated. But until that happens, the data in the short-term working memory will do just fine.

You just need to make sure that you save the data to the file just before the app gets terminated. In other words, the only time you save is when you actually need to keep the data safe.

Not only is this more efficient, especially if you have a lot of data, it also is simpler to program. You no longer need to worry about saving every time the user makes a change to the data, only right before the app terminates.

There are three situations in which an app can terminate:

1. While the user is running the app. This doesn't happen very often anymore, but earlier versions of iOS did not support multitasking apps. Receiving an incoming phone call, for example, would kill the currently running app. As of iOS 4, the app will simply be suspended and sent to the background when that happens.

There are still situations where iOS may forcefully terminate a running app, for example, if the app becomes unresponsive or runs out of memory.

2. When the app is suspended. Most of the time iOS keeps running apps around for a long time. Their data is frozen in memory and no computations are taking place. (When you resume a suspended app, it literally continues from where it left off.)

Sometimes the OS needs to make room for an app that requires a lot of working memory – often a game – and then it simply kills the suspended apps and wipes them from memory. The suspended apps are not notified of this.

3. The app crashes. There are ways to detect crashes but handling them can be very tricky. Trying to deal with the crash may actually make things worse. The best way to avoid crashes is to make no programming mistakes! :]

Fortunately for us, iOS will inform the app about significant changes such as, “you are about to be terminated”, and, “you are about to be suspended”. You can listen for these events and save your data at that point. That will ensure the on-file representation of the data model is always up-to-date when the app does terminate.

Save changes on app termination

The ideal place for handling app termination notifications is inside the **application delegate**. You haven’t spent much time with this object before, but every app has one. As its name implies, it is the delegate object for notifications that concern the app as a whole.

This is where you receive the “app will terminate” and “app will be suspended” notifications.

In fact, if you look inside **AppDelegate.swift**, you’ll see the methods:

```
func applicationDidEnterBackground(_ application: UIApplication)
```

And:

```
func applicationWillTerminate(_ application: UIApplication)
```

There are a few others, but these are the ones you need. (The Xcode template put helpful comments inside these methods, so you know what to do with them.)

Now the trick is, how do you call `AllListsViewController`’s `saveChecklists()` method from these delegate methods? The app delegate does not know anything about `AllListsViewController` yet.

You have to use some trickery to find the All Lists View Controller from within the app delegate.

► Add this new method to **AppDelegate.swift**:

```
func saveData() {  
    let navigationController = window!.rootViewController  
                                as! UINavigationController  
    let controller = navigationController.viewControllers[0]  
                                as! AllListsViewController  
    controller.saveChecklists()  
}
```

The `saveData()` method looks at the `window` property to find the `UIWindow` object that contains the storyboard.

UIWindow is the top-level container for all your app's views. There is only one UIWindow object in your app (unlike desktop apps, which usually have multiple windows).

Exercise: Can you explain why you wrote `window!` with an exclamation point?

Unwrapping optionals

At the top of AppDelegate.swift you can see that `window` is declared as an optional:

```
var window: UIWindow?
```

To *unwrap* an optional you normally use the `if let` syntax:

```
if let w = window {  
    // if window is not nil, w is the real UIWindow object  
    let navigationController = w.rootViewController  
}
```

As a shorthand you can use *optional chaining*:

```
let navigationController = window?.rootViewController
```

If `window` is `nil`, then the app won't even bother to look at the rest of the statement and `navigationController` will also be `nil`.

For apps that use a storyboard (and most of them do), you're guaranteed that `window` is never `nil`, even though it is an optional. UIKit promises that it will put a valid reference to the app's UIWindow object inside the `window` variable when the app starts up.

So why is it an optional? There is a brief moment between when the app is launched and the storyboard is loaded where the `window` property does not have a valid value yet. And if a variable can be `nil` – no matter how briefly – then Swift requires it to be an optional.

If you're *sure* an optional will not be `nil` when you're going to use it, you can *force unwrap* it by adding an exclamation point:

```
let navigationController = window!.rootViewController
```

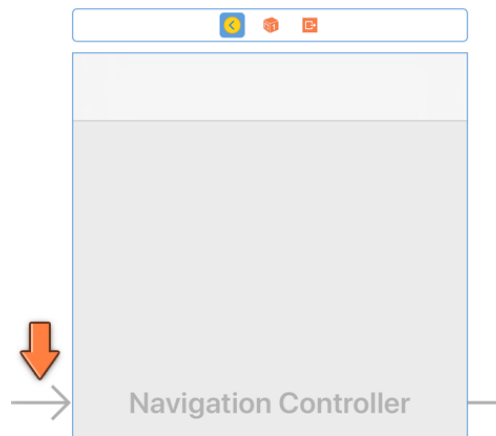
That's exactly what you're doing in the `saveData()` method. Force unwrapping is the simplest way to deal with optionals, but it comes with a danger: if you're wrong and the optional is `nil`, the app will crash. Use with caution!

(You've actually used force unwrapping already when you read the text from the UITextField objects in the Item Detail and List Detail view controllers. The

UITextField text property is an optional String but it will never be nil, which is why you can read it with `textField.text!` – the exclamation point converts the optional String value to a regular String.)

Normally you don't need to do anything with your UIWindow, but in cases such as this you have to ask it for its rootViewController. The “root” or “initial” view controller is the very first scene from the storyboard, the navigation controller all the way over on the left.

You can see this in Interface Builder because this navigation controller has the big arrow pointing at it. This is the one:



The navigation controller is the window's root view controller

(The Attributes inspector for this navigation controller also has the **Is Initial View Controller** box checked, that's the same thing. In the Document Outline it is called the Storyboard Entry Point.)

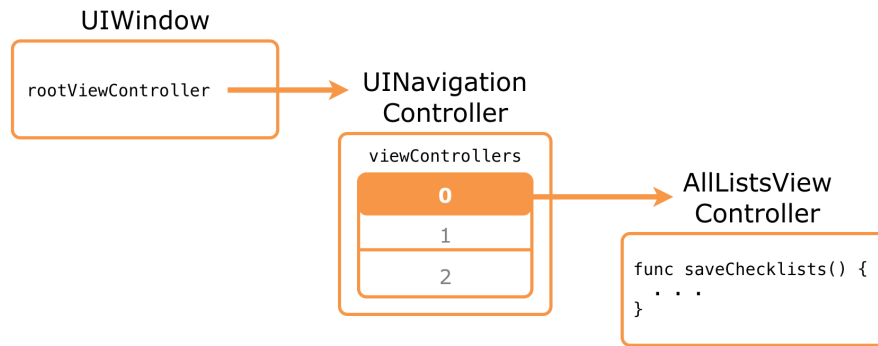
Once you have the navigation controller, you can find the AllListsViewController. After all, that's the view controller that is embedded in the navigation controller.

Unfortunately, the UINavigationController does not have a “rootViewController” property of its own, so you have to look into its viewControllers array to find it:

```
let controller = navigationController.viewControllers[0]
                as! AllListsViewController
```

As usual, a type cast is necessary because the viewControllers array does not know anything about the exact types of your own view controllers. Once you have a reference to AllListsViewController you can call its `saveChecklists()` method.

It's a bit of work to dig through the window and navigation controller to find the view controller you need, but that's life as an iOS developer.



From the root view controller to the AllListsViewController

Note: By the way, the `UINavigationController` does have a `topViewController` property, but you cannot use it here: the “top” view controller is the screen that is currently displaying, which may be the `ChecklistViewController` if the user is looking at to-do items. You don’t want to send the `saveChecklists()` message to that screen – it has no method to handle that message and the app will crash!

► Change the `applicationDidEnterBackground()` and `applicationWillTerminate()` methods to call `saveData()`:

```

func applicationDidEnterBackground(_ application: UIApplication) {
    saveData()
}

func applicationWillTerminate(_ application: UIApplication) {
    saveData()
}

```

► Run the app, add some checklists, add items to those lists, and set some checkmarks.

► Press the Simulator’s home button (or press **Shift+⌘+H**, or pick **Hardware** → **Home** from the Simulator’s menu bar) to make the app go to the background. This simulates what happens when a user taps the home button on their iPhone.

Look inside the app’s Documents folder using Finder. There should be a new `Checklists.plist` file there.

► Press Stop in Xcode to terminate the app. Run the app again and your data should still be there. Awesome!

Xcode's Stop button

Important note: When you press Xcode's Stop button, the application delegate will *not* receive the `applicationWillTerminate(_:)` notification. Xcode kills the app immediately, without mercy.

Therefore, to test the saving behavior, always simulate a tap on the home button to make the app go into the background before you press Stop. If you don't do that, you'll lose your data. *Caveat developer.*

Improve the data model

The above code works, but you can still do a little better. You have made data model objects for `Checklist` and `CheckListItem` but the code for loading and saving the `Checklists.plist` file currently lives in `AllListsViewController`. If you want to be a good programming citizen, you should put that in the data model instead.

The `DataModel` class

I prefer to create a top-level `DataModel` object for many of my apps. For this app, `DataModel` will contain the array of `Checklist` objects. You can move the code for loading and saving data to this new `DataModel` object.

- Add a new file to the project using the **Swift File** template. Save it as **`DataModel.swift`** (you don't need to make this a subclass of anything).
- Change **`DataModel.swift`** to the following:

```
import Foundation

class DataModel {
    var lists = [Checklist]()
}
```

This defines the new `DataModel` object and gives it a `lists` property.

Unlike `Checklist` and `CheckListItem`, `DataModel` does not need to be built on top of `NSObject`. It also does not need to conform to the `Codable` protocol.

`DataModel` will take over the responsibilities for loading and saving the to-do lists from `AllListsViewController`.

► Cut the following methods out of **AllListsViewController.swift** and paste them into **DataModel.swift**:

- `func documentsDirectory()`
- `func dataFilePath()`
- `func saveChecklists()`
- `func loadChecklists()`

► Add an `init()` method to **DataModel.swift**:

```
init() {  
    loadChecklists()  
}
```

This makes sure that as soon as the `DataModel` object is created, it will attempt to load `Checklists.plist`.

You don't have to call `super.init()` because `DataModel` does not have a superclass (it is not built on `NSObject`).

Switch to **AllListsViewController.swift** and make the following changes:

- Remove the `lists` instance variable.
- Remove the call to `loadChecklists()` in `viewDidLoad`.
- Add a new instance variable:

```
var dataModel: DataModel!
```

The `!` is necessary because `dataModel` will temporarily be `nil` when the app starts up. It doesn't have to be a true optional – with `?` – because once `dataModel` is given a value, it will never become `nil` again.

Xcode still finds a number of errors in **AllListsViewController.swift**. You can no longer reference the `lists` variable directly, because it no longer exists. Instead, you'll have to ask the `DataModel` for its `lists` property.

► Wherever the code for `AllListsViewController` says `lists`, replace it with `dataModel.lists`. You need to do this in the following methods:

- `tableView(_:numberOfRowsInSection:)`
- `tableView(_:cellForRowAt:)`
- `tableView(_:didSelectRowAt:)`
- `tableView(_:commit:forRowAt:)`

- `tableView(_:accessoryButtonTappedForRowWith:)`
- `listDetailViewController(_:didFinishAdding:)`
- `listDetailViewController(_:didFinishEditing:)`

Phew, that's a big list! Fortunately, the change is very simple.

To recap, you created a new `DataModel` object that owns the array of `Checklist` objects and knows how to load and save the checklists and their items.

Instead of its own array, the `AllListsViewController` now uses this `DataModel` object, which it accesses through the `dataModel` property.

Create the `DataModel` object

But where does this `DataModel` object get created? There is no place in the code that currently does `dataModel = DataModel()`.

The best place for this is in the app delegate. You can consider the app delegate to be the top-level object in your app. Therefore it makes sense to make it the “owner” of the data model.

The app delegate then passes this `DataModel` object to any view controllers that need to use it.

► In **`AppDelegate.swift`**, add a new property:

```
let dataModel = DataModel()
```

This creates the `DataModel` object and puts it in a constant named `dataModel`.

Even though `AllListsViewController` also has an instance variable named `dataModel`, these two things are totally separate from each other. Here you're only putting the `DataModel` object into `AppDelegate`'s `dataModel` property.

► Simplify the `saveData()` method to just this:

```
func saveData() {  
    dataModel.saveChecklists()  
}
```

If you run the app now, it will crash at once because `AllListsViewController`'s own reference to `DataModel` is still `nil`. I told you those `nil`s were no-gooders!

The best place to share the `DataModel` instance with `AllListsViewController` is in the `application(_:didFinishLaunchingWithOptions:)` method, which gets called as soon as the app starts up.

► Change that method to:

```
func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    let navigationController = window!.rootViewController
        as! UINavigationController
    let controller = navigationController.viewControllers[0]
        as! AllListsViewController
    controller.dataModel = dataModel

    return true
}
```

This finds the `AllListsViewController` by looking in the storyboard (as before) and then sets its `dataModel` property. Now the All Lists screen can access the array of Checklist objects again.

► Do a clean build (**Product** → **Clean**) and run the app. Verify that everything still works. It does? Great!

Still confused about var and let?

If `var` makes a variable and `let` makes a constant, then why were you able to do this in `AppDelegate.swift`:

```
let dataModel = DataModel()
```

You'd think that when something is constant it cannot change, right? Then how come the app lets you add new Checklist objects to `DataModel`? Obviously the `DataModel` object *can* be changed...

Here's the trick: Swift makes a distinction between **value types** and **reference types**, and `let` works a bit differently for both.

An example of a value type is `Int`. Once you create a constant of type `Int` you can never change it afterwards:

```
let i = 100
i = 200      // not allowed
i += 1      // not allowed

var j = 100
j = 200      // allowed
j += 1      // allowed
```

The same goes for other value types such as `Float`, `String`, and even `Array`. They are called value types because the variable or constant directly stores their value.

When you assign the contents of one variable to another, the value is copied into the new variable:

```
var s = "hello"
var u = s        // u has its own copy of "hello"
s += " there"    // s and u are now different
```

But objects that you define with the keyword `class` (such as `DataModel`) are reference types. The variable or constant does not contain the actual object, only a reference to the object.

```
var d = DataModel()
var e = d          // e refers to the same object as d
d.lists.remove(at: 0) // this also changes e
```

You can also write this using `let` and it would do the exact same thing:

```
let d = DataModel()
let e = d          // e refers to the same object as d
d.lists.remove(at: 0) // this also changes e
```

So what is the difference between `var` and `let` for reference types?

When you use `let` it is not the object that is constant but the *reference* to the object. That means you cannot do this:

```
let d = DataModel()
d = someOtherDataModel // error: cannot change the reference
```

The constant `d` can never point to another object, but the object itself can still change.

It's OK if you have trouble wrapping your head around this. The distinction between value types and reference types is an important idea in software development, but it also is something which takes a while to understand.

My suggestion is that you use `let` whenever you can and change to `var` when the compiler complains. Note that optionals always need to be `var`, because being an optional implies that it can change its value at some point.

You can find the project files for the app up to this point under **30 - Improved Data Model** in the Source Code folder.

Chapter 31: User Defaults

By Fahim Farook and Matthijs Hollemans

You now have an app that lets you create lists and add to-do items to those lists. All of this data is saved to long-term storage so that even if the app gets terminated, nothing is lost.

There are some improvements (both to the user interface and to the code) that you can make, though.

This chapter covers the following:

- **Remember the last open list:** Improve the user-experience by remembering the last open list on app re-launch.
- **Defensive programming:** Adding in checks to guard against possible crashes - coding defensively instead of reacting to crashes later.
- **The first-run experience:** Improving the first-run experience for the user so that the app looks more polished and user-friendly.

Remember the last open list

Imagine the user is on the Birthdays checklist and switches to another app. The *Checklists* app is now suspended. It is possible that at some point the app gets terminated and is removed from memory. When the user reopens the app some time later, it no longer is on Birthdays but on the main screen. Because it was terminated, the app didn't simply resume where it left off, but got launched anew.

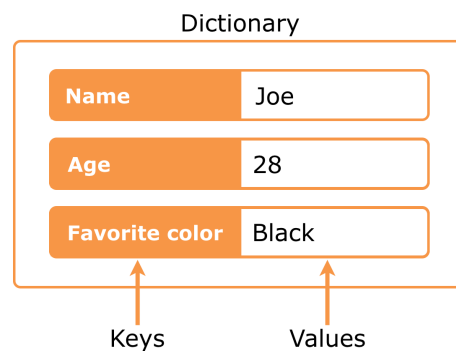
You might be able to get away with this, as apps don't get terminated often (unless your users play a lot of games that eat up memory), but little things like this matter in iOS apps.

Fortunately, it's fairly easy to remember whether the user has opened a checklist and to switch to it when the app starts up.

Use UserDefaults

You could store this information in the Checklists.plist file, but for simple settings such as this, there is the UserDefaults object.

UserDefaults works like a *dictionary*, which is a collection object for storing key-value pairs. You've already seen the array collection, which stores an ordered list of objects. The dictionary is another very common collection that looks like this:



A dictionary is a collection of key-value pairs

Dictionaries in Swift are handled by the Dictionary object (who would've guessed?).

You can put objects into the dictionary under a reference key and then retrieve it later using that key. This is, in fact, how Info.plist works.

The Info.plist file is read into a dictionary and then iOS uses the various keys (on the left hand) to obtain the values (on the right hand). Keys are usually strings but values can be any type of object.

To be accurate, UserDefaults isn't a true dictionary, but it certainly acts like one.

When you insert new values into UserDefaults, they are saved somewhere in your app's sandbox. So, these values persist even after the app terminates.

You don't want to store huge amounts of data inside UserDefaults, but it's ideal for small things like settings – and for remembering what screen the app was on when it closed.

This is what you are going to do:

1. On the segue from the main screen, `AllListsViewController`, to the checklist screen, `ChecklistViewController`, you write the row index of the selected list into `UserDefaults`. This is how you'll remember which checklist was active.

You could have saved the name of the checklist instead of the row index, but what would happen if two checklists have the same name? Unlikely, but not impossible. Using the row index guarantees that you'll always select the proper one.

2. When the user presses the back button to return to the main screen, you have to remove this value from `UserDefaults` again. It is common to set a value such as this to -1 to mean “no value”.

Why -1? You start counting rows at 0, so you can't use 0. Positive numbers are also out of the question, unless you use a huge number such as 1000000 as it's very unlikely the user will make that many checklists. -1 is not a valid row index – and because it's a negative value it looks weird, making it easy to spot during debugging.

(If you're wondering why you're not using an optional for this – good question! – the answer is that `UserDefaults` cannot handle optionals. Sad face.)

3. If the app starts up and the value from `UserDefaults` isn't -1, the user was previously viewing the contents of a checklist and you have to manually perform a segue to the `ChecklistViewController` for the corresponding row.

Phew, it's more work to explain this in English than writing the actual code. ;-)

Let's start with the segue from the main screen. Recall that this segue is triggered from code rather than from the storyboard.

► In `AllListsViewController.swift`, change `tableView(_:didSelectRowAt:)` to the following:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    // add this line:
    UserDefaults.standard.set(indexPath.row, forKey: "ChecklistIndex")

    let checklist = dataModel.lists[indexPath.row]
    performSegue(withIdentifier: "ShowChecklist", sender: checklist)
}
```

In addition to what this method did before, you now store the index of the selected row into `UserDefaults` under the key “`ChecklistIndex`”.

Navigation controller delegate

To be notified when the user presses the back button on the navigation bar, you have to become a delegate of the navigation controller. Being the delegate means that the navigation controller tells you when it pushes or pops view controllers on the navigation stack.

The logical place for this delegate is the `AllListsViewController`.

► Add the delegate protocol to the class line in **AllListsViewController.swift**:

```
class AllListsViewController: UITableViewController,
                             ListDetailViewControllerDelegate,
                             UINavigationControllerDelegate {
```

As you can see, a view controller can be a delegate for many objects at once.

`AllListsViewController` is now the delegate for both the `ListDetailViewController` and the `UINavigationController`, but also implicitly for the `UITableView` (because it is a table view controller).

► Add the following delegate method to **AllListsViewController.swift**:

```
func navigationController(
    _ navigationController: UINavigationController,
    willShow viewController: UIViewController,
    animated: Bool) {

    // Was the back button tapped?
    if viewController === self {
        UserDefaults.standard.set(-1, forKey: "ChecklistIndex")
    }
}
```

This method is called whenever the navigation controller shows a new screen.

If the back button was pressed, the new view controller is `AllListsViewController` itself and you set the “ChecklistIndex” value in `UserDefaults` to -1, meaning that no checklist is currently selected.

Equal or identical

To determine whether the `AllListsViewController` is the newly activated view controller, you wrote:

```
if viewController === self {
```

Yep, it’s not a typo, that’s three equals signs in a row.

Previously to compare objects you used only two equals signs:

```
if segue.identifier == "AddItem" {
```

You may be wondering what the difference is between these two operators. It's a subtle but important question about identity. (Who said programmers couldn't be philosophical?)

If you use `==`, you're checking whether two variables have the same value.

With `===` you're checking whether two variables refer to the exact same object.

Imagine two people who are both called Joe. They're different people who just happen to have the same name.

If we'd compare them using `joe1 === joe2` then the result would be false, as they're not the same person.

But `joe1.name == joe2.name` would be true.

On the other hand, if I'm telling you an amusing (or embarrassing!) story about Joe and this story seems awfully familiar to you, then maybe we happen to know this same Joe.

In that case, `joe1 === joe2` would be true as well.

By the way, the above code would have worked just fine if you had written,

```
if viewController == self
```

with just two equals signs. For objects such as view controllers, equality is tested by comparing the references, just like `===` would do. But technically speaking, `===` is more correct here than `==`.

Show the last open list

The only thing that remains is to check at startup which checklist you need to show and then perform the segue manually. You'll do that in `viewDidAppear()`.

➤ Add the `viewDidAppear()` method to **AllListsViewController.swift**:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    navigationController?.delegate = self

    let index = UserDefaults.standard.integer(
                                                forKey: "ChecklistIndex")
    if index != -1 {
        let checklist = dataModel.lists[index]
    }
}
```

```
performSegue(withIdentifier: "ShowChecklist",
               sender: checklist)
}
```

UIKit automatically calls this method after the view controller has become visible.

First, the view controller makes itself the delegate for the navigation controller.

Every view controller has a built-in `navigationController` property. To access it you use the notation `navigationController?.delegate` because the navigation controller is optional.

(You could also have written `navigationController!` instead of `?`. The difference between the two is that `!` will crash the app if this view controller was ever to be shown outside of a `UINavigationController`, while `?` won't crash but simply ignore the rest of that line. For our app, this does not matter.)

Then it checks `UserDefaults` to see whether it has to perform the segue.

If the value of the “ChecklistIndex” setting is `-1`, then the user was on the app's main screen before the app was terminated, and we don't have to do anything.

However, if the value of the “ChecklistIndex” setting is *not* `-1`, then the user was previously viewing a checklist and the app should segue to that screen. As before, you place the relevant `Checklist` object into the `sender` parameter of `performSegue(withIdentifier:sender:)`.

The `!=` operator means: not equal. It is the opposite of the `==` operator. If you're mathematically-inclined, with some imagination `!=` looks like \neq . (Some languages use `<>` for not equal but that won't work in Swift.)

Note: It may not be immediately obvious what's going on here.

`viewDidAppear()` isn't just called when the app starts up but also every time the navigation controller slides the main screen back into view.

Checking whether to restore the checklist screen needs to happen only once when the app starts, so why did you put this logic in `viewDidAppear()` if it gets called more than once?

Here's the reason:

The very first time `AllListsViewController`'s screen becomes visible, you don't want the `navigationController(_:willShow:animated:)` delegate method to be called yet, as that would always overwrite the old value of “ChecklistIndex” with `-1`, before you've had a chance to restore the old screen.

By waiting to register `AllListsViewController` as the navigation controller delegate until it is visible, you avoid this problem. `viewDidAppear()` is the ideal place for that, so it makes sense to do it from that method.

However, as mentioned, `viewDidAppear()` also gets called after the user presses the back button to return to the All Lists screen. That shouldn't have any unwanted side effects, such as triggering the segue again.

Naturally, the navigation controller calls `navigationController(_:willShow:animated:)` when the back button is pressed, but this happens before `viewDidAppear()`. The delegate method always sets the value of "ChecklistIndex" back to -1, and as a result, `viewDidAppear()` does not trigger a segue again.

And so it all works out... The logic that you added to `viewDidAppear()` only does its job once during app startup. There are other ways to solve this particular issue but this approach is simple, so I like it.

Is all of this going way over your head? Don't fret about it. To get a better idea of what's going on, sprinkle `print()` statements around the various methods to see in which order they get called. Change things around to see what the effect is. Jumping into the code and playing with it is the quickest way to learn!

Double-check that all the lines with `UserDefaults` use the same key name, "ChecklistIndex". If one of them is misspelled, `UserDefaults` is reading from and writing to different items.

► Run the app and go to a checklist screen. Exit to the home screen via the Home button, followed by Stop to quit the app.

Tip: You need to exit to the home screen first because `UserDefaults` may not immediately save its settings to disk, and therefore, you may lose your changes if you kill the app from within Xcode.

Note: Does the app crash for you at this point? That happens if you didn't add any lists or to-do items yet. That's the exact problem we'll solve in the next section. You can either comment out the code in `viewDidAppear()`, add some to-do items, and enable the code again to try it. Or, simply move on to the next section.

► Run the app again and you'll notice that Xcode immediately switches to the screen where you were last at. Cool, huh?

Defensive programming

► Now do the following: stop the app and delete it from the Simulator by holding down on the app icon until it starts to wiggle and then deleting it.

Then, run the app again from within Xcode and watch it crash:

```
fatal error: Index out of range
```

The app crashes in `viewDidAppear()` on the line:

```
let checklist = dataModel.lists[index]
```

What's going on here? Apparently, the value of `index` is not `-1`, because the code entered the `if` statement.

As it turns out `index` is `0`, even though there should be nothing in `UserDefaults` yet because this is a fresh install of the app. The app didn't write anything in the "ChecklistIndex" key yet.

Here's the thing: `UserDefaults`'s `integer(forKey:)` method returns `0` if it cannot find the value for the key you specified. But in this app, `0` is a valid row index.

At this point, the app doesn't have any checklists yet. So, `index 0` does not exist in the `lists` array. That is why the app crashes.

What should happen instead, is that `UserDefaults` returns `-1` if nothing is set yet for "ChecklistIndex", because to your app `-1` means: show the main screen instead of a specific checklist.

Set a default value for a UserDefaults key

Fortunately, `UserDefaults` will let you set default values for the default values. Yep, you read that correctly. Let's do that in the `DataModel` object.

► Add the following method to **DataModel.swift**:

```
func registerDefaults() {  
    let dictionary = [ "ChecklistIndex": -1 ]  
    UserDefaults.standard.register(defaults: dictionary)  
}
```

This creates a new `Dictionary` instance and adds the value `-1` for the key "ChecklistIndex".

The square bracket notation is not only used to make arrays, but also dictionaries. The difference is that for a dictionary it always looks like,

```
[ key1: value1, key2: value2, . . . ]
```

while an array is just:

```
[ value1, value2, value3, . . . ]
```

UserDefaults will use the values from this dictionary if you ask it for a key but it cannot find anything under that key.

► Change **DataModel.swift**'s `init()` to call this new method:

```
init() {  
    loadChecklists()  
    registerDefaults()  
}
```

► Run the app again. Now, it should no longer crash.

Why did you do this in `DataModel`? Well, I don't really like to sprinkle all of these calls to `UserDefaults` throughout the code.

Clean up the code

In fact, let's move all of the `UserDefaults` stuff into `DataModel`.

► Add the following to **DataModel.swift**:

```
var indexOfSelectedChecklist: Int {  
    get {  
        return UserDefaults.standard.integer(  
            forKey: "ChecklistIndex")  
        }  
    set {  
        UserDefaults.standard.set(newValue,  
            forKey: "ChecklistIndex")  
        }  
}
```

This does something you haven't seen before. It appears to declare a new instance variable `indexOfSelectedChecklist` of type `Int`, but what are these `get { }` and `set { }` blocks?

This is an example of a *computed property*.

There isn't any storage allocated for this property (so it's not really a variable). Instead, when the app tries to read the value of `indexOfSelectedChecklist`, the code in the `get` block is performed. And when the app tries to put a new value into `indexOfSelectedChecklist`, the `set` block is performed.

From now on, you can simply use `indexOfSelectedChecklist` and it will automatically update `UserDefaults`. How cool is that?

You're doing this so the rest of the code won't have to worry about `UserDefaults` anymore. The other objects just have to use the `indexOfSelectedChecklist` property on `DataModel`.

Hiding implementation details is an important object-oriented programming principle, and this is one way to do it.

If you decide later that you want to store these settings somewhere else, for example, in a database, or in iCloud, then you only have to change this in one place, in `DataModel`. The rest of the code will be oblivious to these changes. That's a good thing.

► Update the code in **AllListsViewController.swift** to use this new computed property:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    navigationController?.delegate = self

    let index = dataModel.indexOfSelectedChecklist // change this
    if index != -1 {
        let checklist = dataModel.lists[index]
        performSegue(withIdentifier: "ShowChecklist",
                      sender: checklist)
    }
}
```

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    // change this line
    dataModel.indexOfSelectedChecklist = indexPath.row

    let checklist = dataModel.lists[indexPath.row]
    performSegue(withIdentifier: "ShowChecklist",
                  sender: checklist)
}
```

```
func navigationController(
    _ navigationController: UINavigationController,
    willShow viewController: UIViewController,
    animated: Bool) {
    if viewController === self {
```

```
        dataModel.indexOfSelectedChecklist = -1    // change this
    }
}
```

The intent of the code is now much clearer. `AllListsViewController` no longer has to worry about the “how” – storing values in `UserDefaults` – and can simply focus on the “what” – changing the index of the selected checklist.

► Run the app again and make sure everything still works.

A subtle bug

It’s pretty nice that the app now remembers what screen you were on, but this new feature has also introduced a subtle bug in the app. Here’s how to reproduce it:

► Start the app and add a new checklist. Also, add a new to-do item to this list. Now kill the app from within Xcode.

Because you did not exit to the home screen first, the new checklist and its item were not saved to `Checklists.plist`.

However, there is a (small) chance that `UserDefaults` did save its changes to disk and now thinks this new list is selected. That’s a problem because that list doesn’t exist anymore (it never made it into `Checklists.plist`).

`UserDefaults` will save its changes at indeterminate times. So, it could have saved before you terminated the app.

► Run the app again and – if you’re lucky? – it will crash with:

```
fatal error: Index out of range
```

If you can’t get this error to appear, add the following line to the `set` block of `indexOfSelectedChecklist` and try again. This forces `UserDefaults` to save its changes every time `indexOfSelectedChecklist` changes:

```
set {
    UserDefaults.standard.set(newValue,
                              forKey: "ChecklistIndex")
    UserDefaults.standard.synchronize()
}
```

The reason for the crash is that `UserDefaults` and the contents of `Checklists.plist` are out-of-sync. `UserDefaults` thinks the app needs to select a checklist that doesn’t actually exist. Every time you run the app it will now crash. Yikes!

This situation shouldn't really happen during regular usage. It happened because you used the Xcode Stop button to kill the app before it had a chance to save the plist file.

Under normal circumstances, the user would press the home button. As the app goes into the background, it properly saves both `Checklists.plist` and `UserDefaults` and everything is in sync again.

However, the OS can always decide to terminate the app and then this same situation could occur.

Even though there's only a small chance that this can go wrong in practice, you should really protect the app against this. These are the kinds of bug reports you don't want to receive because often, you have no idea what the user did to make it happen.

This is where the practice of *defensive programming* becomes important. Your code should always check for such boundary cases and be able to gracefully handle them even if they are unlikely to occur.

In our case, you can easily fix `AllListsViewController`'s `viewDidAppear()` method to deal with this situation.

► Change the `if` statement in `viewDidAppear()` to:

```
if index >= 0 && index < dataModel.lists.count {
```

Instead of just checking for `index != -1`, you now do a more precise check to determine whether `index` is valid. It should be between 0 and the number of checklists in the data model. If not, then you simply don't segue.

This prevents `dataModel.lists[index]` from asking for an object at an `index` that doesn't exist.

You haven't seen the `&&` operator before. This symbol means "logical and". It is used as follows:

```
if something && somethingElse {  
    // do stuff  
}
```

This reads: if something is true **and** something else is also true, then do stuff.

In `viewDidAppear()` you only perform the segue when `index` is 0 or greater and also less than the number of checklists, which means it's only valid if it lies in between those two values.

With this defensive check in place, you're guaranteed that the app will not try to segue to a checklist that doesn't exist, even if the data is out-of-sync.

Note: Even though the app remembers what checklist the user was on, it won't bother to remember whether the user had the Add/Edit Checklist or Add/Edit Item screen open.

These kinds of data input screens are supposed to be temporary. You open them to make a few changes and then close them again. If the app goes to the background and is terminated, then it's no big deal if the data input screen disappears.

At least, that is true for this app. If you have an app that allows the user to make many complicated edits in an input screen, you may want to persist those changes when the app closes so the user won't lose all their work in case the app is killed.

In this chapter you used `UserDefaults` to remember which screen was open, but iOS actually has a dedicated API for this kind of thing, State Preservation and Restoration. You can read more about this on raywenderlich.com/117471/state-restoration-tutorial.

The first-run experience

Let's use `UserDefaults` for something else. It would be nice if the first time you ran the app it created a default checklist for you, simply named "List", and switched over to that list. This enables you to start adding to-do items right away.

That's how the standard Notes app works too: you can start typing a note right after launching the app for the very first time, but you can also go one level back in the navigation hierarchy to see a list of all notes.

Check for first run

To implement the above feature, you need to keep track in `UserDefaults` whether this is the first time the user runs the app. If it is, you create a new `Checklist` object.

You can perform all of this logic inside `DataModel`.

It's a good idea to add a new default setting to the `registerDefaults()` method. The key for this value is "FirstTime".

► Change the `registerDefaults()` method in **DataModel.swift** (don't miss the comma after the first line of the dictionary):

```
func registerDefaults() {
    let dictionary: [String:Any] = [ "ChecklistIndex": -1,
                                    "FirstTime": true ]

    UserDefaults.standard.register(defaults: dictionary)
}
```

The “FirstTime” setting can be a boolean value because it's either true (this is the first time) or false (this is any other than the first time).

The value of “FirstTime” needs to be true if this is the first launch of the app after a fresh install.

Also, note that there's now a type declaration for `dictionary`. Why was that added? Try removing the type declaration, the `: [String:Any]` bit, and see what happens. Xcode will throw up an error.

This is because originally, there was one value in the dictionary and it was an `Int`. But when you introduced the `FirstTime` key, its corresponding value is a `Bool`. Now your dictionary has a mixed set of values - an `Int` and a `Bool`. So, at this point, the compiler is unsure whether you meant to have a mixed bag of values, or if it was a mistake on your part. So it wants you to explicitly indicate what the dictionary type is, and that's why you declare it as `[String:Any]`, to indicate that the value could indeed be of any type.

► Still in **DataModel.swift**, add a new `handleFirstTime()` method:

```
func handleFirstTime() {
    let userDefaults = UserDefaults.standard
    let firstTime = userDefaults.bool(forKey: "FirstTime")

    if firstTime {
        let checklist = Checklist(name: "List")
        lists.append(checklist)

        indexOfSelectedChecklist = 0
        userDefaults.set(false, forKey: "FirstTime")
        userDefaults.synchronize()
    }
}
```

Here you check `UserDefaults` for the value of the “FirstTime” key. If the value for “FirstTime” is true, then this is the first time the app is being run. In that case, you create a new `Checklist` object and add it to the array.

You also set `indexOfSelectedChecklist` to 0, which is the index of this newly added Checklist object, to make sure the app will automatically segue to the new list in `AllListsViewController`'s `viewDidAppear()` method.

Finally, you set the value of “FirstTime” to false, so this code won't be executed again the next time the app starts up.

► Call this new method from `DataModel`'s `init()`:

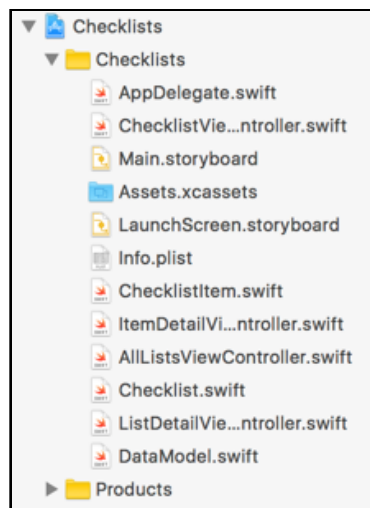
```
init() {  
    loadChecklists()  
    registerDefaults()  
    handleFirstTime()  
}
```

► Remove the app from the Simulator and run it again from Xcode.

Because it's the first time you run the app (at least from the app's perspective) after a fresh install, it will automatically create a new checklist named List and switch to it.

Organizing Source Files

At this point, your Project navigator probably lists your files something like this:



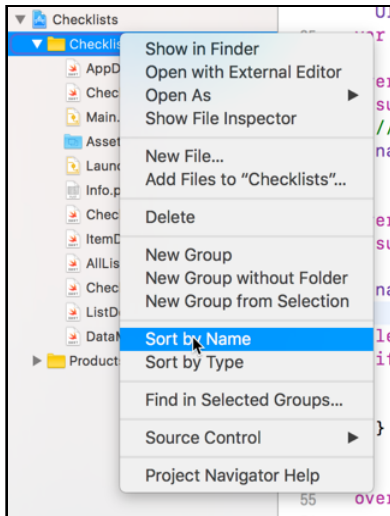
Project navigator file listing

It's a bit messy since it's hard to find where a given file is. Sure, you know exactly where each file is now, but what happens when you have 20 or 30 files in there? Or a hundred?

Xcode does provide a few different ways to organize your files.

The first thing you can do is a simple alphabetical sorting of files so that you can find a given file quickly - since it will be in alphabetical order. That is simple enough to accomplish.

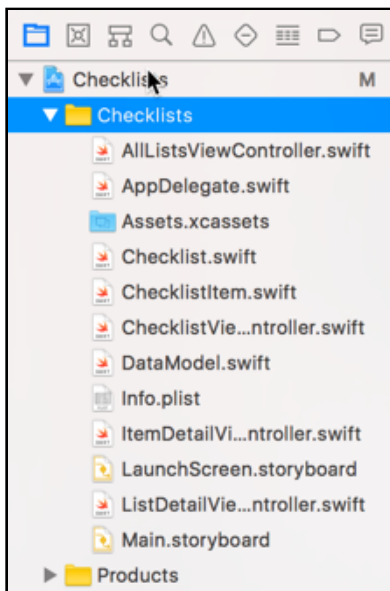
► Right-click (or Control-click) on the yellow **Checklists** folder. A context menu should pop up.



Context menu for folder

► Select **Sort by Name**.

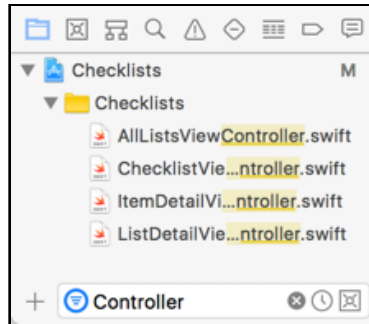
Voila! All the files inside the Checklists folder are now in alphabetical order.



Sorted file listing

That certainly makes finding files a lot easier, but what if you had 20 or 30 files? Or even a hundred? You would still have to do a lot of scrolling around to find the exact file you wanted.

Xcode does provide a filter field at the bottom of the Navigator pane that you can use to filter files in the current list by name. You can type in, for example, "Controller" and it will display only the files with "Controller" in the file name. (You can click the little circle icon with an "x" in the filter field to clear the filter.)

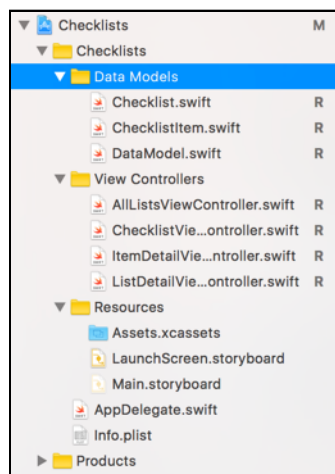


Filter file list by name

But you can do better :) You can also organize your files into virtual folders, called *groups*, so that you can organize the files by functionality. For example, you can put all your view controllers together into a folder called View Controllers, the data models into a Data Models folder and so on ...

You probably noticed the New Group menu option on the folder context menu when you right-clicked on the Checklists folder earlier. That's what you need to use in order to create a new group. Simply create a new group (or three), drag files into the group and you should be set.

You could quite easily organize the file listing from above to look something like this:



Organized file listing

You can find the project for the app up to this point under **31 - UserDefaults** in the Source Code folder.

Chapter 32: UI Improvements

By Fahim Farook and Matthijs Hollemans

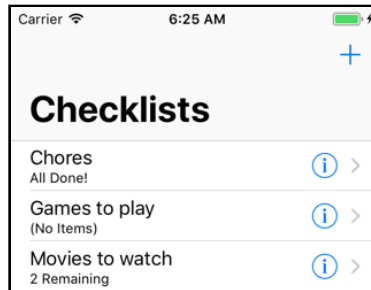
Checklists now has full functionality and is starting to come together. However, There are a few small features I'd like to add, just to polish the app a little more. After all, you're building a real app here – if you want to make top-notch apps, you have to pay attention to those tiny details.

This chapter covers the following:

- **Show counts:** Show the number of to-do items remaining for each list.
- **Sort the lists:** Sort the list of checklist items alphabetically.
- **Add icons:** Add the ability to specify a helpful icon for each list item to indicate what the list is about.
- **Make the app look good:** Improve how the app looks by making a few basic colour changes to give it its own unique style.
- **Support all iOS devices:** Add support for the different screen sizes present on the various iOS devices.

Show counts

On the main screen, for each checklist, the app will show the number of to-do items that do not have checkmarks yet:



Each checklist shows how many items are still left to-do

Count the unchecked items

First, you need a way to count these items.

► Add the following method to **Checklist.swift**:

```
func countUncheckedItems() -> Int {  
    var count = 0  
    for item in items where !item.checked {  
        count += 1  
    }  
    return count  
}
```

With this method you can ask any Checklist object how many of its ChecklistItem objects do not yet have their checkmark set. The method returns this count as an Int value.

You use a `for in` to loop through the ChecklistItem objects from the `items` array. If an `item` object has its `checked` property set to `false`, you increment the local variable `count` by 1.

Remember that the `!` operator negates the result. So if `item.checked` is `true`, then `!item.checked` will make it `false`. You should read it as “where not `item.checked`”.

Note: If the `!` symbol is written in front of something then it is the logical **not** operator, as you see here. When the `!` is written behind something, it’s related to optionals. This is another example of a symbol that has more than one meaning in Swift. The correct interpretation depends on the context where it is being used.

When the loop is over and you've looked at all the objects, you return the total value of the count to the caller.

Exercise: What would happen if you used `let` instead of `var` to make the count variable?

Answer: When count is a constant, Swift won't let you change its value, so the line that does `+= 1` will show an error message.

By the way, you could also have written the loop as follows:

```
for item in items {  
    if !item.checked {  
        count += 1  
    }  
}
```

This uses the more familiar `if` statement instead. Personally, I like the brevity of the `for in` where loop, but using an `if` is just as valid.

Display the unchecked item count

► Go to **AllListsViewController.swift** and in `makeCell(for:)` change `style: .default` to `style: .subtitle`.

The rest of the code stays the same, except you now use `.subtitle` for the cell style instead of `.default`. The “subtitle” cell style adds a second, smaller label below the main label. You can use the cell's `detailTextLabel` property to access this subtitle label.

► That happens in `tableView(_:cellForRowAt)`. Add the following line just before `return cell`:

```
cell.detailTextLabel!.text =  
    "\(checklist.countUncheckedItems()) Remaining"
```

You call the `countUncheckedItems()` method on the `Checklist` object and put the count into a new string that you display using the `detailTextLabel`.

As usual, you use `\(...)` to do the string interpolation. Notice that you can even call methods inside interpolated strings. Sweet!

Force unwrapping

To put text into the cell's labels, you wrote:

```
cell.textLabel!.text = someString
```



```
cell.detailTextLabel!.text = anotherString
```

The `!` is necessary because `textLabel` and `detailTextLabel` are optionals.

The `textLabel` property is only present on table view cells that use one of the built-in cell styles; it is `nil` on custom cell designs. Likewise, not all of the cell styles have a detail label and `detailTextLabel` will be `nil` in those cases.

Here you're using the "Subtitle" cell style, which is guaranteed to have both labels. Because these optionals will never be `nil` for a "Subtitle" cell, you can use `!` to *force unwrap* them. This turns the optional into an actual object that you can use.

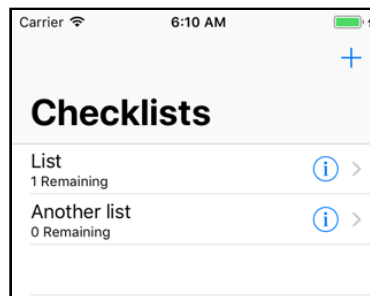
Be careful with this, though... using `!` on an optional that *is* `nil` will crash your app immediately.

You could also have written the above code as:

```
if let label = cell.textLabel {  
    label.text = someString  
}  
if let label = cell.detailTextLabel {  
    label.text = anotherString  
}
```

That is safer – no chance of crashing here – but also a bit more cumbersome. Writing `!` is just more convenient in this case.

► Run the app. For each checklist it will now show how many items still remain unchecked.



The cells now have a subtitle label

Update the unchecked item count on changes

One problem: The to-do count never changes. If you toggle a checkmark on or off, or add new items, the “to do” count remains the same. That’s because you create these table view cells once and never update their labels. (Try it out!)

Exercise: Think of all the situations that will cause this “still to do” count to change.

Answer:

- The user toggles a checkmark on an item. When the checkmark is set, the count goes down. When the checkmark gets removed, the count goes up again.
- The user adds a new item. New items don’t have their checkmark set, so adding a new item should increment the count.
- The user deletes an item. The count should go down but only if that item had no checkmark.

These changes all happen in the `ChecklistViewController` but the “still to do” label is shown in the `AllListsViewController`.

So, how do you let the All Lists View Controller know about this?

If you thought, “That’s easy, let’s use a delegate!”, then you’re starting to get the hang of this. You could make a new `ChecklistViewControllerDelegate` protocol that sends messages when the following things happen:

- The user toggles a checkmark on an item
- The user adds a new item
- The user deletes an item

But what would the delegate – which would be `AllListsViewController` – do in response? It would simply set some new text on the cell’s `detailTextLabel` in all cases.

The delegate approach sounds good, but you’re going to cheat and not use a delegate at all :] There is a simpler solution, and a smart programmer always picks the simplest way to solve a problem.

► Go to **`AllListsViewController.swift`** and add the `viewWillAppear()` method to do the following:

```
override func viewWillAppear(_ animated: Bool) {  
    super.viewWillAppear(animated)  
    tableView.reloadData()  
}
```

Don’t confuse this method with `viewDidAppear()`. The difference is in the verb: *will* versus *did*. `viewWillAppear()` is called before `viewDidAppear()`, when the view is about

to become visible but the animation hasn't started yet. `viewDidAppear()` is called after the view is visible on the screen and the animation has completed. There may be half a second or so difference between them as the animation takes place.

The iOS API often does this: there is a “will” method that is invoked before something happens and a “did” method that is invoked after that something happens. Sometimes you need to do things before, sometimes after, and having two methods gives you the ability to choose whichever situation works best for you.

API (ay-pee-eye) stands for **A**pplication **P**rogramming **I**nterface. When people say “the iOS API” they mean all the frameworks, objects, protocols and functions that are provided by iOS that you as a programmer can use to write apps.

The iOS API consists of everything from UIKit, Foundation, Core Graphics, and so on. Likewise, when people talk about “the Facebook API” or “the Google API”, they mean the services that these companies provide that allow you to write apps for those platforms.

Here, `viewWillAppear()` tells the table view to reload its entire contents. That will cause `tableView(_:cellForRowAt:)` to be called again for every visible row.

When you tap the back button on the `ChecklistViewController`'s navigation bar, the `AllListsViewController` screen will slide back into view. Just before that happens, `viewWillAppear()` is called. Thanks to the call to `tableView.reloadData()` the app will update all of the table cells, including the `detailTextLabels`.

Reloading all of the cells may seem like overkill, but in this situation you can easily get away with it. It's unlikely the All Lists screen will contain many rows (say, less than 100) and only about 14 visible cells, so reloading them is quite fast. And it saves you some work of having to make yet another delegate.

Sometimes a delegate is the best solution; sometimes you just reload the entire table.

➤ Run the app and test that it works!

Display a completion message when all items are done

Exercise. Change the label to read “All Done!” when there are no more to-do items left to check.

Answer: Change the relevant code in `tableView(_:cellForRowAt:)` to:

```
let count = checklist.countUncheckedItems()
if count == 0 {
    cell.detailTextLabel!.text = "All Done!"
} else {
    cell.detailTextLabel!.text = "\(count) Remaining"
}
```

You put the count into a local constant because you refer to it twice. Calculating the count once and storing it into a temporary constant is more optimal than doing the same calculation twice.

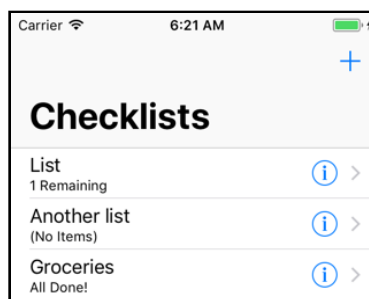
Display an indicator when there are no items in a list

Exercise: Now update the label to say “No Items” when the list is empty.

Answer:

```
let count = checklist.countUncheckedItems()
if checklist.items.count == 0 {
    cell.detailTextLabel!.text = "(No Items)"
} else if count == 0 {
    cell.detailTextLabel!.text = "All Done!"
} else {
    cell.detailTextLabel!.text = "\(count) Remaining"
}
```

Just looking at the result of `countUncheckedItems()` is not enough. If this returns 0, you don’t know whether that means all items are checked off or if the list has no items at all. You also need to look at the total number of items in the checklist, with `checklist.items.count`.



The text in the detail label changes depending on how many items are checked off

Little details like these matter – they make your app more fun to use. Ask yourself, what would make you feel better about having done your chores, the rather bland message “0 Remaining” or the joyous exclamation “All Done!”?

A short diversion into Functional Programming

Swift is primarily an object-oriented language. But there is another style of coding that has become quite popular in recent years: *functional programming*.

The term “functional” means that programs can be expressed purely in terms of mathematical functions that transform data.

Unlike the methods and functions in Swift, these mathematical functions are not allowed to have “side effects”. For any given inputs, a function should always produce the same output. Methods are much less strict.

Even though Swift is not a purely functional language, it does let you use certain functional programming techniques in your apps. They can really make your code a lot shorter.

For example, let’s look at `countUncheckedItems()` again:

```
func countUncheckedItems() -> Int {
    var count = 0
    for item in items where !item.checked {
        count += 1
    }
    return count
}
```

That’s quite a bit of code for something that’s fairly simple. You can actually write this in a single line of code:

```
func countUncheckedItems() -> Int {
    return items.reduce(0) { cnt,
                          item in cnt + (item.checked ? 0 : 1) }
}
```

`reduce()` is a method that looks at each item and performs the code in the `{ }` block. Initially, the `cnt` variable contains the value 0, but after each item it is incremented by either 0 or 1, depending on whether the item was checked.

Incidentally, the `item.checked ? 0 : 1` bit is a simpler way to do an `if ... else` block - the `? .. :..` construct is known as a *ternary conditional operator* - if the first part (the bit before the `?`) evaluates to true, then the result of the expression would be the item after the `?`. Otherwise, the result is the item after the `:`. It can be very handy in a lot of places to write simpler, more succinct code.

When `reduce()` is done, its return value is the total count of unchecked items.

You don’t have to remember any of this for now, but it’s pretty cool to see that Swift allows you to express this kind of algorithm very succinctly.

Sort the lists

Another thing you often need to do with lists is sort them in some particular order.

Let's sort the list of checklists by name. Currently when you add a new checklist it is always appended to the end of the table, regardless of alphabetical order.

When do you do the sorting?

Before we figure out how to sort an array, let's think about when you need to perform this sort:

- When a new checklist is added
- When a checklist is renamed

There is no need to re-sort when a checklist is deleted because that doesn't have any impact on the order of the other objects.

Currently you handle these two situations in `AllListsViewController`'s implementation of `didFinishAdding` and `didFinishEditing`.

► Change these methods to the following:

```
func listDetailViewController(
    _ controller: ListDetailViewController,
    didFinishAdding checklist: Checklist) {
    dataModel.lists.append(checklist)
    dataModel.sortChecklists()
    tableView.reloadData()
    navigationController?.pushViewController(animated: true)
}

func listDetailViewController(
    _ controller: ListDetailViewController,
    didFinishEditing checklist: Checklist) {
    dataModel.sortChecklists()
    tableView.reloadData()
    navigationController?.pushViewController(animated: true)
}
```

You were able to remove a whole bunch of stuff from both methods because you now always do `reloadData()` on the table view.

It is no longer necessary to insert the new row manually, or to update the cell's `textLabel`. Instead you simply call `tableView.reloadData()` to refresh the entire table's contents.

Again, you can get away with this because the table will only hold a handful of rows. If this table had hundreds of rows, a more advanced approach might be necessary. (You could figure out where the new or renamed `Checklist` object should be inserted and just update that row.)

The sorting algorithm

The `sortChecklists()` method on `DataModel` is new and you still need to add it. But before that, we need to have a short discussion about how sorting works.

When you sort a list of items, the app will compare the items one-by-one to figure out what the proper order is. But what does it mean to compare two `Checklist` objects?

In *Checklists* we obviously want to sort them by name, but we need some way to tell the app that's what we mean.

► Add the following method to **DataModel.swift**:

```
func sortChecklists() {
    lists.sort(by: { checklist1, checklist2 in
        return checklist1.name.localizedStandardCompare(
            checklist2.name) == .orderedAscending })
}
```

Here you tell the `lists` array that the `Checklists` it contains should be sorted using some specific logic.

That logic is provided in the shape of a *closure*. You can tell it's a closure by the `{ }` brackets around the sorting code:

```
lists.sort(by: { /* the sorting code goes here */ })
```

You've briefly seen closures with the alert box in the *Bull's Eye* app. They wrap a piece of source code into an anonymous, inline method.

The purpose of the closure is to determine whether one `Checklist` object comes before another, based on our rules for sorting.

The sort algorithm will repeatedly ask one `Checklist` object from the list how it compares to the other `Checklist` objects using the logic from the closure, and then shuffle them around until the array is sorted.

This allows `sort()` to sort the contents of the array in any order you desire. If you wanted to sort on other criteria, all you'd have to do is change the logic inside the closure.

The actual sorting code is this:

```
checklist1.name.localizedStandardCompare(
    checklist2.name) == .orderedAscending
```

To compare these two Checklist objects, you're only looking at their names.

The `localizedStandardCompare(_:)` method compares the two name strings while ignoring lowercase vs. uppercase (so "a" and "A" are considered equal) and taking into consideration the rules of the current *locale*.

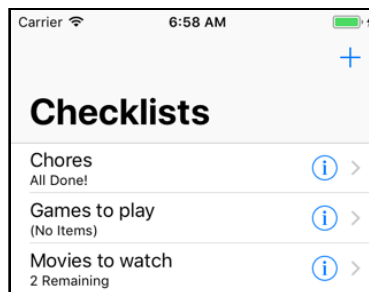
A locale is an object that knows about country and language-specific rules. Sorting in German may be different than sorting in English, for example.

That's all you have to do to sort the array: call `sort()` and give it a closure with the logic that compares two Checklist objects.

► Just to make sure the existing lists are also sorted in the right order, you should also call `sortChecklists()` when the plist file is loaded:

```
func loadChecklists() {
    ...
    lists = try decoder.decode([Checklist].self, from: data)
    sortChecklists() // Add this
} catch {
    ...
}
```

► Run the app and add some new checklists. Change their names and notice that the list is always sorted alphabetically.

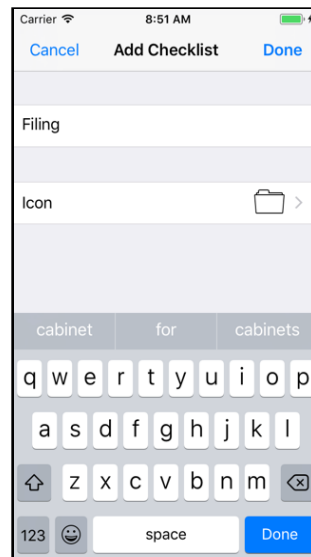


New checklists are always sorted alphabetically

Add icons

Because true iOS developers can't get enough of view controllers and delegates, let's add a new property to the Checklist object that lets you choose an icon. We're really going to cement these principles in your mind!

When you're done, the Add/Edit Checklist screen will look like this:

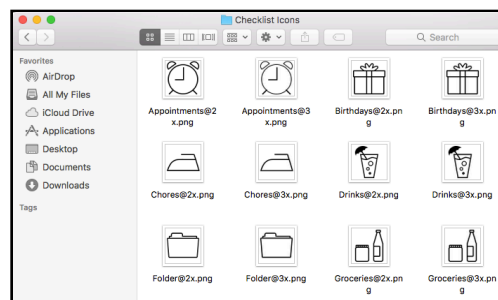


You can assign an icon to a checklist

You are going to add a row to the Add/Edit Checklist screen that opens a new screen for picking an icon. This icon picker is a new view controller and you will show it by pushing it on to the navigation stack, just like your previous view controllers.

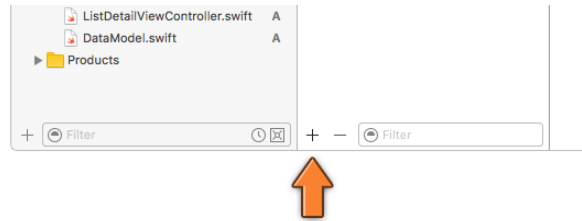
Add the icons to the project

The Resources folder for the book contains a folder named **Checklist Icons** with a selection of PNG images that depict different categories.



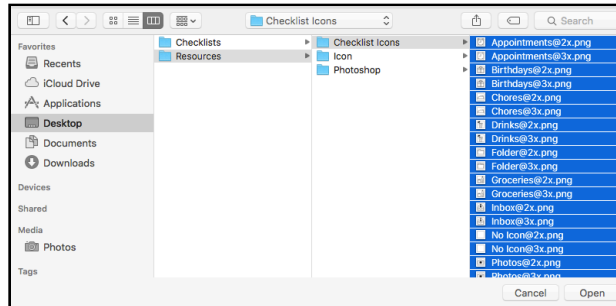
The various checklist icon images

➤ Add the images from this folder to the asset catalog. Select **Assets.xcassets** in the project navigator, click the + button at the bottom and choose **Import...**



Importing new images into the asset catalog

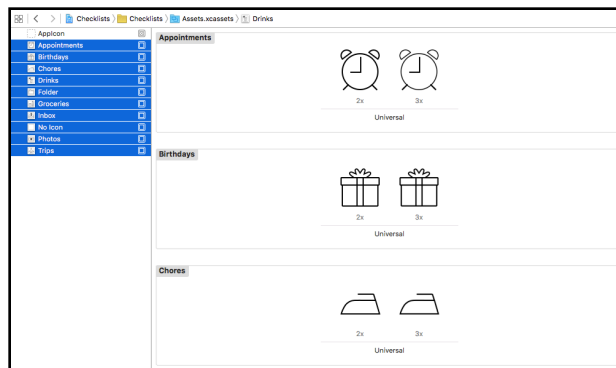
Navigate to the **Checklist Icons** folder and select all the files inside:



Selecting the image files to import

Note: Make sure to select the actual image files, not just the folder.

Click **Open** to import the images. The asset catalog should now look like this:



The asset catalog after importing the checklist icons

Each image comes with a 2x version for Retina devices and a 3x version for the iPhone Plus with the Retina HD screen.

As I pointed out in the previously, you don't need low-resolution 1x graphics anymore. All iPhone, iPad, and iPod touch devices that can run iOS 11 have Retina 2x or 3x screens.

Update the data model

► Add the following property to **Checklist.swift**:

```
var iconName = ""
```

The `iconName` variable holds the filename of the icon image.

The above code initializes `iconName` to have no icon set by default. But what if you actually wanted to create new `Checklist` objects with a default icon set?

It's very easy to implement a default icon. Say, you want all new checklists to have the "Appointments" icon - then change the above line to this:

```
var iconName = "Appointments"
```

And that's all you need to do :]

Display the icon

At this point, you just want to see that you can make an icon – any icon – show up in the table view. When that works, you can worry about letting the user pick their own icons. (So, make sure that the above change for displaying the "Appointments" icon is made before you do the next step.)

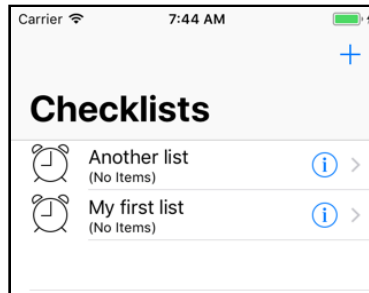
► Change `tableView(_:cellForRowAt:)` in **AllListsViewController.swift** to put the icon into the table view cell:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    . . .
    cell.imageView!.image = UIImage(named: checklist.iconName)
    return cell
}
```

Cells using the standard `.subtitle` cell style come with a built-in `UIImageView` on the left. You can simply pass it an image and it will be displayed automatically. Easy peasy.

Note: When you run the app, you will not see any of your previously saved checklist items. Can you guess why? The addition of `iconName` changed the `Checklist` object and the previously saved information for the object is no longer valid. So, the decoder will run into issues when trying to decode the previously saved file and so, you will end up with no saved items. Sorry.

► Run the app, create a few checklists and now each of them should have an alarm clock icon.



The checklists have an icon

The default icon

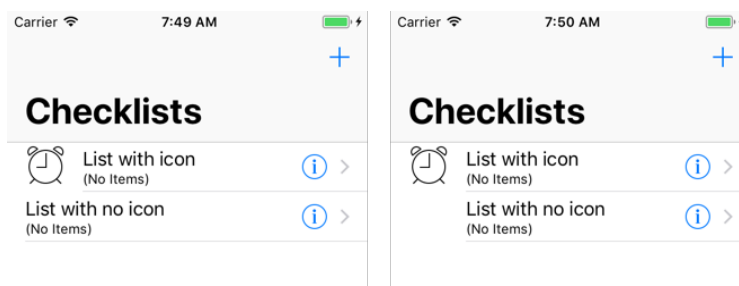
Now that you know it works, you can now change Checklist to give each Checklist object an icon named “No Icon” by default.

► In **Checklist.swift**, change the `iconName` declaration to:

```
var iconName = "No Icon"
```

The “No Icon” image is a fully transparent PNG image with the same dimensions as the other icons. Using a transparent image is necessary to make all the checklists line up properly, even if they have no icon.

If you were to set `iconName` to an empty string instead, the image view in the table view cell would remain empty and the text would align with the left margin of the screen. That looks bad when other cells do have icons:



Using an empty image to properly align the text labels (right)

The icon picker class

Now, let's create the icon picker screen.

► Add a new Swift file to the project. Name it **IconPickerController**.

- Replace the contents of **IconPickerViewController.swift** with:

```
import UIKit

protocol IconPickerViewControllerDelegate: class {
    func iconPicker(_ picker: IconPickerViewController,
                    didPick iconName: String)
}

class IconPickerViewController: UITableViewController {
    weak var delegate: IconPickerViewControllerDelegate?
}
```

This defines the `IconPickerViewController` object, which is a table view controller, and a delegate protocol that it uses to communicate with other objects in the app.

- Add a constant (inside the class brackets) to hold the array of icons:

```
let icons = [ "No Icon", "Appointments", "Birthdays", "Chores",
              "Drinks", "Folder", "Groceries", "Inbox", "Photos", "Trips" ]
```

This is an array that contains a list of icon names. These strings are both the text you will show on the screen and the name of the PNG file inside the asset catalog.

The `icons` array is the data model for this table view. Note that it is a non-mutable array (it is defined with `let` and arrays are “value” types), because the user cannot add or delete icons.

This new view controller is a `UITableViewController`, so you have to implement the data source methods for the table view.

- Add the following methods to the source file:

```
// MARK:- Table View Delegates
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    return icons.count
}
```

This simply returns the number of icons in the array.

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
                        -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(
        withIdentifier: "IconCell",
        for: indexPath)

    let iconName = icons[indexPath.row]
    cell.textLabel!.text = iconName
    cell.imageView!.image = UIImage(named: iconName)
```

```
    return cell  
}
```

Here you obtain a table view cell and give it a title and an image. You will design this cell in the storyboard momentarily. It will be a prototype cell with the “Default” cell style (or “Basic” as it is called in Interface Builder). Cells with this style already contain a text label and an image view, which is very convenient.

The icon picker storyboard changes

► Open the storyboard. Drag a new **Table View Controller** from the Object Library and place it next to the Add Checklist scene.

► In the **Identity inspector**, change the class of this new table view controller to **IconPickerViewController**.

► Select the prototype cell and set its **Style** to **Basic** and its (re-use) **Identifier** to **IconCell**.

That takes care of the design for the icon picker. Now you need to have some place to call it from. To do this, you will add a new row to the Add Checklist screen.

► Go to the **Add Checklist View Controller** and add a new section to the table view. You can do this by changing the **Sections** value in the **Attributes inspector** for the table view from 1 to 2. This will duplicate the existing section.

► Delete the Text Field from the new cell; you don’t need it.

► Add a **Label** to this cell and change its text to **Icon**.

► Set the cell’s **Accessory** to **Disclosure Indicator**. That adds a gray chevron.

► Add an **Image View** to the right of the cell. Make it 36 × 36 points big. (Tip: use the Size inspector for this.)

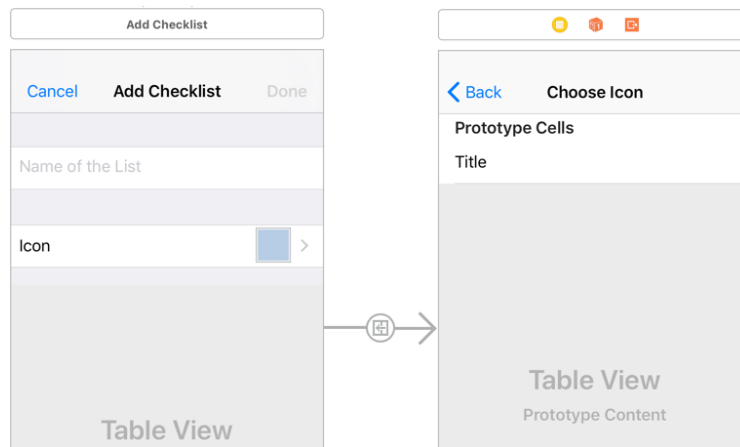
► Use the **Assistant Editor** to add an outlet property for this image view to **ListDetailViewController.swift** and name it **iconImageView**.

Now that you’ve finished the designs for both screens, you can connect them via a segue.

► **Control-drag** from the “Icon” table view cell to the Icon Picker View Controller and add a segue of type **Selection Segue – Show**. (Make sure you’re dragging from the Table View Cell, not its Content View or any of the other subviews. If you are unable to do this accurately from the scene, remember that you can also Control-drag from the Document Outline.)

- Give the segue the identifier **PickIcon**.
- Thanks to the segue, the new view controller has been given a navigation bar. (However, it might not have a Navigation Item - if it doesn't, drag one from the Object Library on to the Icon Picker scene.) Double-click the navigation bar and change its title to **Choose Icon**.

This part of the storyboard should now look like this:



The Icon Picker view controller in the storyboard

Display the icon picker

- In **ListDetailViewController.swift**, change the `willSelectRowAt` table view delegate method to:

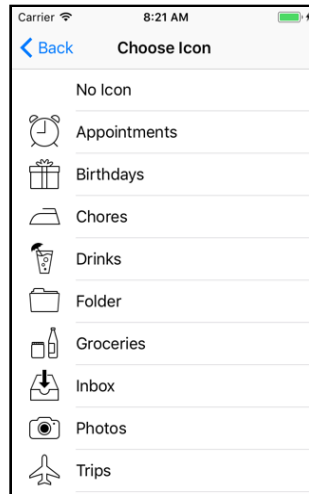
```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath)
    -> IndexPath? {
    if indexPath.section == 1 {
        return indexPath
    } else {
        return nil
    }
}
```

Without this change you cannot tap the “Icon” cell to trigger the segue.

Previously this method always returned `nil`, which meant tapping on rows was not possible. Now, however, you want to allow the user to tap the Icon cell, so this method should return the index-path for that cell.

Because the Icon cell is the only row in the second section, you only have to check `indexPath.section`. There is no need to check the row number too. Users still can't select the cell with the text field (from section 0).

► Run the app and verify that there is now an Icon row in the Add/Edit Checklist screen. Tapping it will open the Choose Icon screen and show a list of icons.



The icon picker screen

Handle icon selection

You can press the back button to go back but selecting an icon doesn't do anything yet. It just colors the row gray but doesn't put the icon into the checklist.

To make this work, you have to hook up the icon picker to the Add/Edit Checklist screen through its own delegate protocol.

► First, add an instance variable in **ListDetailViewController.swift**:

```
var iconName = "Folder"
```

You use this variable to keep track of the chosen icon name.

Even though the Checklist object now has an iconName property, you cannot keep track of the chosen icon in the Checklist object for the simple reason that you may not always have a Checklist object, i.e. when the user is adding a new checklist.

So, you'll store the icon name in a temporary variable and copy that into the Checklist's iconName property at the right time.

You should initialize the `iconName` variable with something reasonable. Let's go with the folder icon. This is only necessary for new Checklists, which get the Folder icon by default.

► Update `viewDidLoad()` to the following:

```
override func viewDidLoad() {  
    · · ·  
    if let checklist = checklistToEdit {  
        · · ·  
        iconName = checklist.iconName           // add this  
    }  
    iconImageView.image = UIImage(named: iconName) // add this  
}
```

This has two new lines: If the `checklistToEdit` optional is not `nil`, then you copy the Checklist object's icon name into the `iconName` instance variable. You also load the icon's image file into a new `UIImage` object and set it on the `iconImageView` so it shows up in the Icon row.

Earlier you created a push segue named "PickIcon". You still need to implement `prepare(for:sender:)` in order to tell the `IconPickerViewController` that this screen is now its delegate.

► First, add the name of that protocol to the class line in **ListDetailViewController.swift**:

```
class ListDetailViewController: UITableViewController,  
    UITextFieldDelegate, IconPickerViewControllerDelegate {
```

► Next, add the implementation of the method from that delegate protocol:

```
// MARK:- Icon Picker View Controller Delegate  
func iconPicker(_ picker: IconPickerViewController,  
    didPick iconName: String) {  
    self.iconName = iconName  
    iconImageView.image = UIImage(named: iconName)  
    navigationController?.popViewController(animated: true)  
}
```

This puts the name of the chosen icon into the `iconName` variable to remember it, and also updates the image view with the new image.

After you do all that, you use `popViewController(animated:)` to "pop" the Icon Picker View Controller off the navigation stack.

Recall that `navigationController` is an optional property of the view controller, so you need to use `?` (or `!`) to access the actual `UINavigationController` object.

► Now, add the following method to **ListDetailViewController.swift**:

```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "PickIcon" {
        let controller = segue.destination
                           as! IconPickerViewController
        controller.delegate = self
    }
}
```

This code should have no big surprises for you.

► Change the `done()` action so that it puts the chosen icon name into the Checklist object when the user closes the screen:

```
@IBAction func done() {
    if let checklist = checklistToEdit {
        checklist.name = textField.text!
        checklist.iconName = iconName // add this
        delegate?.listDetailViewController(self,
                                           didFinishEditing: checklist)
    } else {
        let checklist = Checklist(name: textField.text!)
        checklist.iconName = iconName // add this
        delegate?.listDetailViewController(self,
                                           didFinishAdding: checklist)
    }
}
```

Finally, you must change `IconPickerViewController` to actually call the delegate method when a row is tapped.

► Add the following method to the bottom of **IconPickerViewController.swift**:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    if let delegate = delegate {
        let iconName = icons[indexPath.row]
        delegate.iconPicker(self, didPick: iconName)
    }
}
```

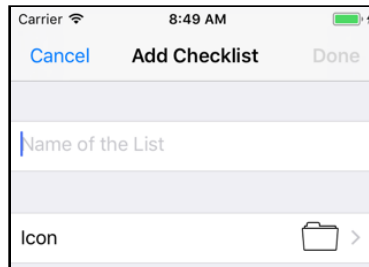
And that's it. You can now set icons on the Checklist objects.

To recap, you:

- Added a new view controller object.
- Designed its user interface in the storyboard editor.
- Hooked it up to the Add/Edit Checklist screen using a segue and a delegate.

Those are the basic steps you need to take with any new screen that you add.

► Run the app to try it out.



You can now give each list its own icon

Achievement unlocked: users can pick icons!

Code refactoring

There's still a small improvement you can make to the code. In `done()`, you currently do this:

```
let checklist = Checklist(name: textField.text!)
checklist.iconName = iconName
```

Setting the icon name can be considered part of the initialization of `Checklist`, so it would be nice if you could pass the icon name to the `Checklist` initializer. And you can :]

► Switch to **Checklist.swift** and modify the `init` method as follows:

```
init(name: String, iconName: String = "No Icon") {
    self.name = name
    self.iconName = iconName
    super.init()
}
```

The modified `init` method looks almost the same as the previous one except for taking a new `iconName` parameter and assigning it to the object's `iconName` property.

But what is the `= "No Icon"` bit after the second parameter? That's called a *default parameter value*. When you specify a default parameter value for a method, when the method is called, you can omit the parameters with default values and the method call would still work, but the default values would be used for the parameters that were omitted. Nifty, huh?

► In `ListDetailViewController.swift`'s `done()` method, replace the code that creates the new `Checklist` object with this (and remove the line after that which sets the `iconName` property):

```
let checklist = Checklist(name: textField.text!,
                          iconName: iconName)
```

► Build the app to verify it still works.

Exercise: Give `CheckListItem` an `init(text:)` method that is used instead of the parameter-less `init()`. Or how about an `init(text:checked:)` method?

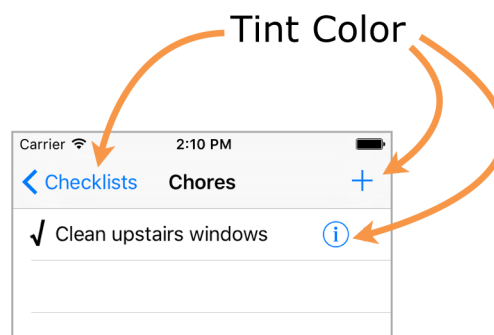
Make the app look good

For *Checklists*, you're going to keep things simple as far as fancying up the graphics goes. The standard look of navigation controllers and table views is perfectly adequate, although a little bland. In the next apps you'll see how you can customize the look of these UI elements.

Change the tint color

Even though this app uses the stock visuals, there is a simple trick to give the app its own personality: changing the **tint color**.

The tint color is what UIKit uses to indicate that things, such as buttons, can be interacted with. The default tint color is a medium blue.

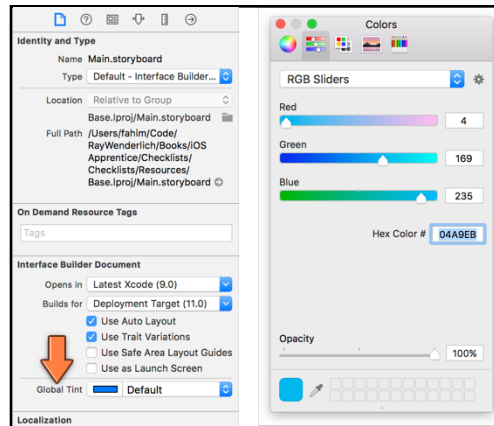


The buttons all use the same tint color

Changing the tint color is pretty easy.

► Open the storyboard and go to the **File inspector** (the first tab).

- Click **Global Tint** to open the color picker and choose Red: 4, Green: 169, Blue: 235. That makes the tint color a lighter shade of blue.



Changing the Global Tint color for the storyboard

Tip: If the color picker only shows a black & white bar, then click the dropdown at the top that says Gray Scale Slider and change it to **RGB Sliders**.

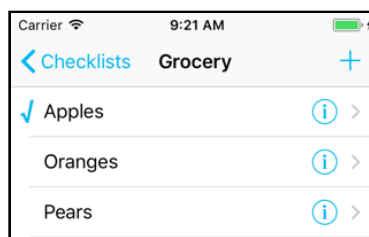
Set the color of the checkmark

It would also look nice if the checkmark wasn't black but used the tint color too.

- To make that happen, add the following line to `configureCheckmark(for:with:)` in **ChecklistViewController.swift**:

```
label.textColor = view.tintColor
```

- Run the app. It already looks a lot more interesting:



The tint color makes the app less plain looking

Add app icons

No app is complete without an icon. The Resources folder for this app contains a folder named **Icon** with the app icon image in various sizes. Notice that it uses the same blue as the tint color.

► Add these icons to the asset catalog (**Assets.xcassets**). Recall that icons go into the **AppIcon** section. Simply drag them from the Finder into the slots.



The app icons in the asset catalog

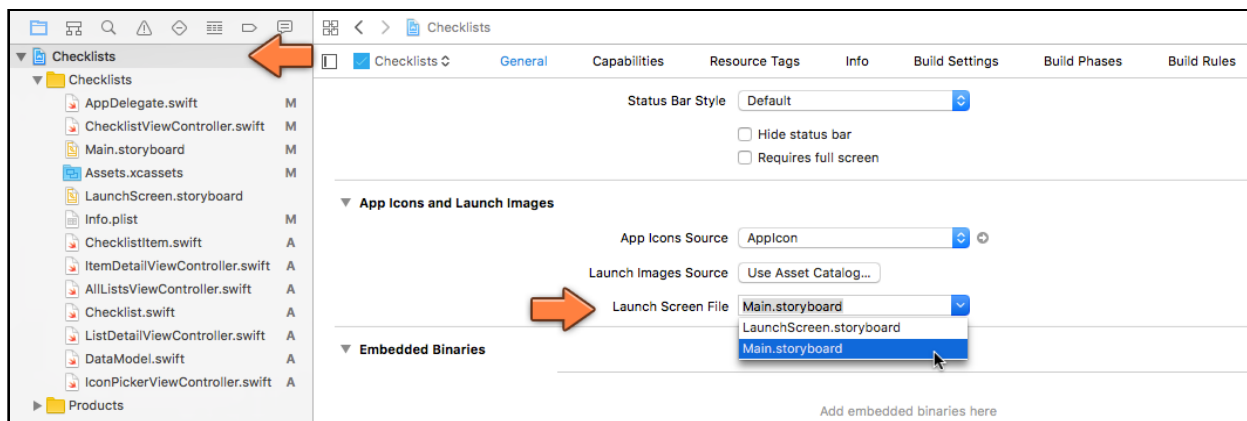
Set the launch image

Apps should also have a launch image or launch file. Showing a static picture of the app's UI will give the illusion of the app is loading faster than it really is. It's all smoke and mirrors.

The Xcode template includes the file **LaunchScreen.storyboard** that is used as the launch file. With some effort you could make this look like the initial screen of the app, but there's an easier solution.

► Open the **Project Settings** screen. In the **General** tab, scroll down to the **App Icons and Launch Images** section.

► In the **Launch Screen File** box, press the arrow and select **Main.storyboard**.



Changing the launch screen file

This tells the app you'll be using the design from the storyboard as the launch file.

Upon startup, the app finds the initial view controller and converts it into a static launch image. For this app that is the All Lists View Controller inside its navigation controller.

- Delete **LaunchScreen.storyboard** from the project.
- From the **Product** menu choose **Clean**. It's also a good idea to delete the app from the Simulator just so it no longer has any copies of the old launch file lying around (hold down on the icon until it starts to wiggle, just like on a real iPhone).
- Run the app. Just before the real UI appears you should briefly see the following launch screen:



The empty launch screen

The launch screen simply has a navigation bar and an empty table view. This gives the illusion the app's UI has already been loaded, though in reality, that the data hasn't been filled in yet.

Using a proper launch screen makes the app look more professional – and faster!

For many apps, you can simply use the main storyboard as the launch file, making it a no-brainer to add.

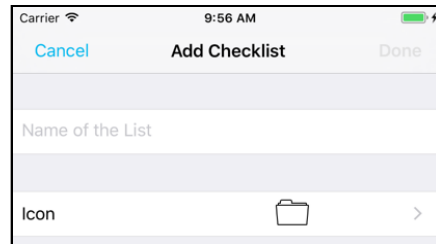
Support all iOS devices

The app should run without major problems on all current iOS devices, from the smallest (iPhone SE) to the largest (iPad Pro). Table view controllers are very flexible and will automatically resize to fit the screen, no matter how large or small. Give it a try in the different Simulators!

Well, I said no *major* problems. But there are still a few tweaks you can make here and there.

Update the Add Checklist rows for larger screens

So far, I've been showing you screenshots of the iPhone SE simulator, and I also designed my screens in Interface Builder using the dimensions of the iPhone SE. But this is what happens when running the app on a larger simulator such as the iPhone 8 Plus or the iPhone X:



The icon is in the wrong place

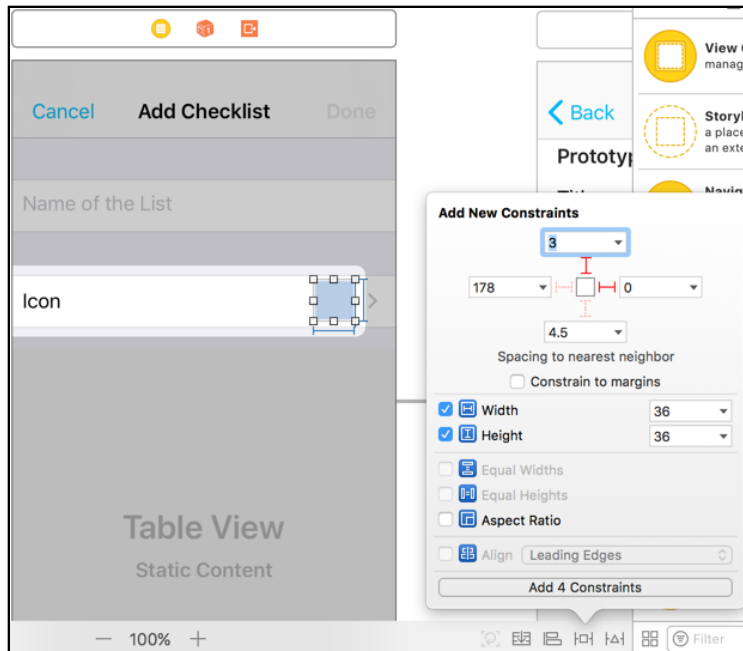
The icon is no longer nicely aligned on the right. Also try typing some text: it gets cut off because the text field is too small. Why does this happen?

When you design the user interface for your app in Interface Builder, it doesn't automatically fit all possible screen sizes, only the one you're designing for. You need to help Interface Builder out and tell it how to adjust your UI for different screen sizes. As you saw before with *Bull's Eye*, that's where Auto Layout comes in.

What you want to happen is that the image view stays glued to the right edge of the screen, always at the same distance from the disclosure indicator. When the view controller grows or shrinks to fit the iPhone screen, the image view should move along with it.

The solution is to add Auto Layout constraints to the image view that tell the app what the relationship is between the image view and the edges of its parent view.

- Select the **Icon Image View**. Bring up the **Add New Constraints** menu using the icon at the bottom of the canvas.
- First, uncheck **Constrain to margins**.
- Activate the bars at the top and the right so they turn red.
- Put checkmarks in front of **Width** and **Height**.



Adding constraints to the Image View

► Finally, click **Add 4 Constraints** to finish.

The image view should now look like this:



The Image View with the constraints

Make sure the bars representing the constraints are blue. If they are orange or red you may have forgotten something in the Add New Constraints menu. (Either try again or use the **Editor** → **Resolve Auto Layout Issues** → **Update Frames** menu item.)

The most important constraint is the one on the right. This tells UIKit that the right-hand side of the image view should always stick to the right-hand edge of the table view cell's content view.

In other words, no matter how wide or narrow the screen is, the image view will always have the same location relative to the disclosure indicator.

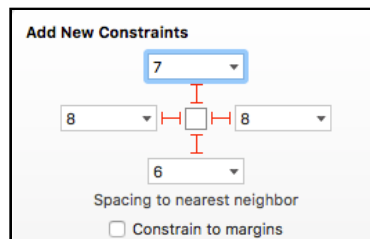
The other three constraints – top, width, and height – were necessary only because all views must always have enough constraints to determine their position and size.

If you don't specify any constraints of your own, Interface Builder will come up with reasonable default constraints. But as soon as you add just one custom constraint, you'll have to add the others too.

► To verify that your changes do the right thing you don't necessarily need to run the app in the simulator. Use the **View as:** panel at the bottom to switch between the different iPhone models right inside Interface Builder. If your constraints are correct, then the icon should always be in the right place.

While you're at it, you might as well fix the text field so that it stretches the entire width of the screen.

► Select the **Text Field** and in the **Add New Constraints** menu activate the four bars so they all become red:

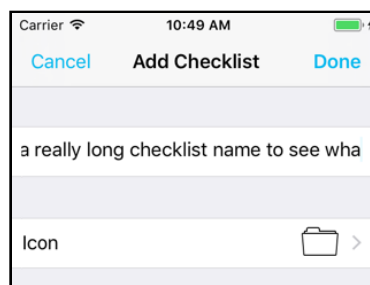


Pinning the text field

These options will make the text field stick to the sides of the table view cell. (The numbers here don't really matter, so it's fine if your numbers are slightly different. The important thing is that there are four red bars indicating the four active constraints.)

► Also do this for the text field on the Add/Edit Item screen.

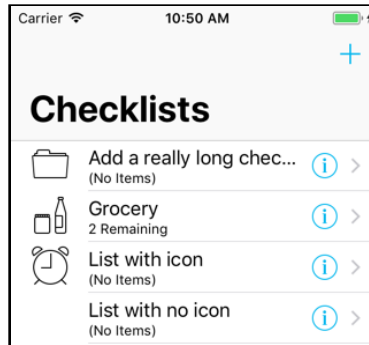
Now you can type all the way to the edge and then the text will start scrolling:



Type to your heart's content

Let's say you enter a long text value. What happens to that text when it gets shown in the other table view?

There is no problem on the All Lists screen:

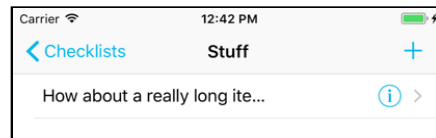


Built-in cell styles automatically resize

This table view uses the built-in “Subtitle” cell style, which automatically resizes to fit the width of the screen. It also truncates the text with ... when it becomes too large.

Update to-do items list for larger screens

For the to-do items table, however, the picture doesn’t look so rosy. The text gets cut off before the end of the screen on larger devices:



The text gets cut off

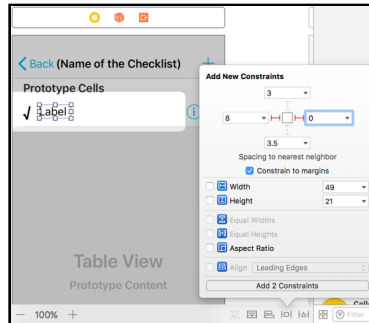
Because this is a custom prototype cell design, you’ll have to add some constraints to stop this from happening.

- In the storyboard, go to the Checklist screen and select the label inside the prototype cell.
- First use **Editor** → **Size to Fit Content** to give the label its ideal size. That makes it a lot smaller, but that’s OK. Without doing this first you may run into issues on the next steps. (Don’t worry if doing this also moves the label.)

You want to pin the label to the right edge of the content view so it sticks to the disclosure button. Let’s make that constraint first.

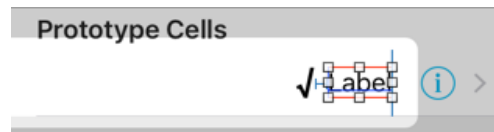
- Open the **Add New Constraints** menu and uncheck **Constrain to margins**.
- Activate the red I-beam on the **left**. Keep the value at what is suggested if you are happy with the spacing between the label and the checkmark before it.

- Activate the red I-beam on the **right**. Give it the value 0 so there is no spacing between the label and the disclosure button.
- Click **Add 1 Constraint** to add the new constraint.



Pinning the label

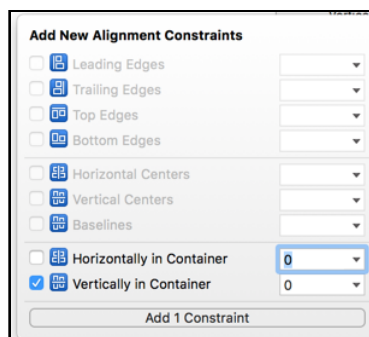
Hmm ... that moves everything right and the label has a red outline around it:



The label doesn't have enough constraints yet

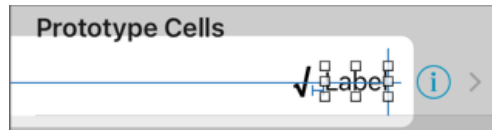
Remember that you always need to specify enough constraints to determine the position and size of a view. Here you only have enough for the label to position itself horizontally, but what about vertically?

- With the label still selected, open the **Align menu** (next to Add New Constraints). Check **Vertically in Container** and click **Add 1 Constraint**.



Centering the label vertically

Now everything turns blue again. The label has a valid position, both X and Y.



All blue bars but still in the wrong place

Note: Even though you didn't specify any constraints for the label's size, the bars are all blue. How come they are not red or orange?

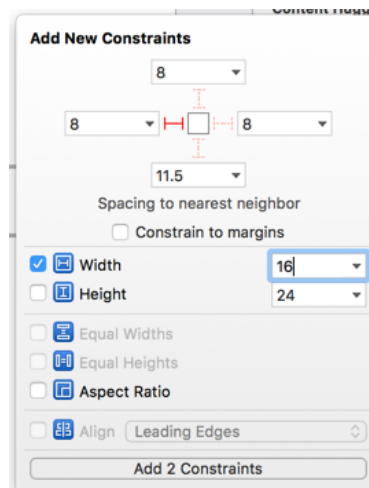
Without size constraints, the label uses its contents – the text and the font – to calculate how big needs to be. This is called the *intrinsic content size*.

UI components with an intrinsic size, such as `UILabel`, don't need to have Width or Height constraints, but this is only valid if you've used Size to Fit Content to reset the label to its intrinsic size first.

Unfortunately, the label is now right aligned. That's not what you wanted... the label should be on the left and just as wide as the cell's content view.

The easiest way to make this happen is to constraints to the checkmark icon to glue it to the left edge of the screen as well. However, do note that the checkmark label changes size depending on whether the checkmark is set or not. So you can't depend on the label's intrinsic content size here. (Otherwise, when the checkmark is not showing, the text for those rows would be slightly shifted to the left.)

► Instead, you'll set the **left** spacing for the checkmark label as well as a specific **width**. Like this:

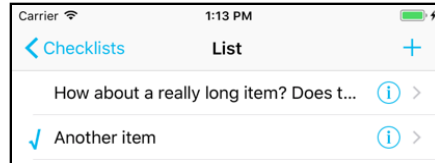


Checkmark label constraints

The label now stretches to be as wide as the table view cell.

There's one more thing to be done - you need to vertically center the checkmark. Easy enough since you are now a master of auto layout, right?

- With the checkmark label still selected, open the **Align menu** (next to Add New Constraints). Check **Vertically in Container** and click **Add 1 Constraint**.
- Run the app and the label should properly truncate:



The label uses as much room as it can

You can find the project for the app up to this point under **32 - UI Improvements** in the Source Code folder.

Chapter 33: Local Notifications

By Fahim Farook and Matthijs Hollemans

I hope you're still with me! We have discussed in great detail view controllers, navigation controllers, storyboards, segues, table views and cells, and the data model. These are all essential topics to master if you want to build iOS apps because almost every app uses these building blocks.

In this chapter you're going to expand the app to add a new feature: **local notifications**, using the iOS User Notifications framework. A local notification allows the app to schedule a reminder to the user that will be displayed even when the app is not running.

You will add a “due date” field to the `CheckListItem` object and then remind the user about this deadline with a local notification.

If this sounds like fun, then keep reading. :-)

The steps for this chapter are as follows:

- **Try it out:** Try out a local notification just to see how it works.
- **Set a due date:** Allow the user to pick a due date for to-do items.
- **Due date UI:** Create a date picker control.
- **Schedule local notifications:** Schedule local notifications for the to-do items, and update them when the user changes the due date.

Try it out

Before you wonder about how to integrate local notifications with *Checklists*, let's just schedule a local notification and see what happens.

By the way, local notifications are different from *push* notifications (also known as *remote* notifications). Push notifications allow your app to receive messages about external events, such as your favorite team winning the World Series.

Local notifications are more similar to an alarm clock: you set a specific time and then it “beeps”.

Get permission to display local notifications

An app is only allowed to show local notifications after it has asked the user for permission. If the user denies permission, then any local notifications for your app simply won't appear. You only need to ask for permission once, so let's do that first.

► Open **AppDelegate.swift** and add an new import to the top of the file:

```
import UserNotifications
```

This tells Xcode that we're going to use the User Notifications framework.

► Add the following to the method `application(_:didFinishLaunchingWithOptions:)`, just before the `return true` line:

```
// Notification authorization
let center = UNUserNotificationCenter.current()
center.requestAuthorization(options: [.alert, .sound]) {
    granted, error in
    if granted {
        print("We have permission")
    } else {
        print("Permission denied")
    }
}
```

Recall that `application(_:didFinishLaunchingWithOptions:)` is called when the app starts up. It is the *entry point* for the app, the first place in the code where you can do something after the app launches.

Because you're just playing with these local notifications now, this is a good place to ask for permission.

You tell iOS that the app wishes to send notifications of type “alert” with a sound effect. Later you'll put this code into a more appropriate place.

Things that start with a dot

Throughout the app you’ve seen things like `.none`, `.checkmark`, and `.subtitle` – and now `.alert` and `.sound`. These are *enumeration* symbols.

An enumeration, or enum for short, is a data type that consists of a list of possible symbols and their values.

For example, the `UNAuthorizationOptions` enum contains the symbols:

```
.badge  
.sound  
.alert  
.carPlay
```

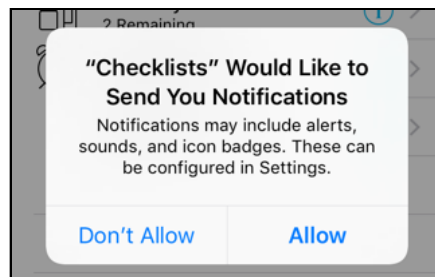
You can combine these names in an array to define what sort of notifications the app will show to the user. Here you’ve chosen the combination of an alert and a sound effect by writing `[.alert, .sound]`.

It’s easy to spot when an enum is being used because of the dot in front of the symbol name. This is actually shorthand notation; you could also have written it like this:

```
center.requestAuthorization(options:  
    [UNAuthorizationOptions.alert, UNAuthorizationOptions.sound]) {  
    . . .  
}
```

Fortunately, Swift is smart enough to realize that `.alert` and `.sound` are from the enum `UNAuthorizationOptions`, so you can save yourself some keystrokes.

➤ Run the app. You should immediately get a popup asking for permission:



The permission dialog

Tap **Allow**. The next time you run the app you won’t be asked again; iOS remembers your choice.

(If you tapped Don’t Allow – naughty! – then you can always reset the Simulator to get the permissions dialog again. You can also change the notification options via the *Settings* app.)

Show a test local notification

► Stop the app and add the following code to the end of `didFinishLaunchingWithOptions` (but before the return):

```
let content = UNMutableNotificationContent()
content.title = "Hello!"
content.body = "I am a local notification"
content.sound = UNNotificationSound.default()

let trigger = UNTimeIntervalNotificationTrigger(
    TimeInterval: 10,
    repeats: false)

let request = UNNotificationRequest(
    identifier: "MyNotification",
    content: content,
    trigger: trigger)

center.add(request)
```

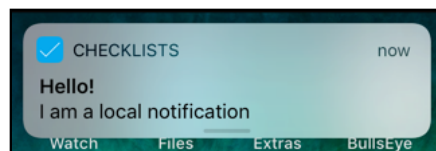
This creates a new local notification. Because you wrote `TimeInterval: 10`, it will fire exactly 10 seconds after the app has started.

The `UNMutableNotificationContent` describes what the local notification will say. Here, you set an alert message to be shown when the notification fires. You also set a sound.

Finally, you add the notification to the `UNUserNotificationCenter`. This object is responsible for keeping track of all the local notifications and displaying them when they are up.

► Run the app. Immediately after it has started, exit to the home screen.

Wait 10 seconds... I know, it seems like an eternity! After an agonizing 10 seconds a message should pop up:



The local notification message

► Tap the notification to go back to the app.

And that's a local notification. Pretty cool, huh?

Why did I want you to exit to the home screen? iOS will only show a notification alert if the app is not currently active.

► Stop the app and run it again. This time don't press Home and just wait.

Well, don't wait too long – nothing will happen. The local notification does get fired, but it is not shown to the user. To handle this situation, we must listen somehow to interesting events that concern these notifications. How? Through a delegate, of course!

Handle local notification events

- Add the following to AppDelegate's class declaration:

```
class AppDelegate: UIResponder, UIApplicationDelegate,  
                  UNUserNotificationCenterDelegate {
```

This makes AppDelegate the delegate for the UNUserNotificationCenter.

- Also add the following method to **AppDelegate.swift**:

```
// MARK:- User Notification Delegates  
func userNotificationCenter(  
    _ center: UNUserNotificationCenter,  
    willPresent notification: UNNotification,  
    withCompletionHandler completionHandler:  
    @escaping (UNNotificationPresentationOptions) -> Void) {  
    print("Received local notification \(notification)")  
}
```

This method will be invoked when the local notification is posted and the app is still running. You won't do anything here except log a message to the debug pane.

When your app is active and in the foreground, it is supposed to handle any fired notifications in its own manner. Depending on the type of app, it may make sense to react to the notification, for example to show a message to the user or to refresh the screen.

- Finally, tell the UNUserNotificationCenter that AppDelegate is now its delegate. You do this in `application(_:didFinishLaunchingWithOptions:)` (add this after you ask for permission - perhaps when permission is granted?):

```
center.delegate = self
```

- Run the app again and just wait (don't press Home).

After 10 seconds you should see a message in the Xcode Console. It displays something like this:

```
Received local notification <UNNotification: 0x7ff54af135e0; date:  
2016-07-11 14:21:27 +0000, request: <UNNotificationRequest: . . .  
identifier: MyNotification, content: <UNNotificationContent: . . .  
title: Hello!, subtitle: (null), body: I am a local notification,  
. . .
```

All right, now you know that it works, you should remove the test code from **AppDelegate.swift** because you don't really want to schedule a new notification every time the user starts the app.

► Remove the the local notification code from `didFinishLaunchingWithOptions`, but keep these lines:

```
let center = UNUserNotificationCenter.current()
center.delegate = self
```

You can also keep the `userNotificationCenter(_:willPresent:withCompletionHandler:)` method, as it will come in handy when debugging the local notifications.

Set a due date

Let's think about how the app will handle these notifications. Each `CheckListItem` will get a due date field (a `Date` object, which specifies a date and time) and a `Bool` that says whether the user wants to be reminded of this item or not.

Users might not want to be reminded of everything, so you shouldn't schedule local notifications unless the user asks for it. Such a `Bool` variable is often called a *flag*. Let's name it `shouldRemind`.

When do you schedule a notification?

First, let's figure out how and when to schedule the notifications. I can think of the following situations:

- When the user adds a new `CheckListItem` object that has the `shouldRemind` flag set, you must schedule a new notification.
- When the user changes the due date on an existing `CheckListItem`, the old notification (if there is one) should be cancelled and a new one scheduled in its place (if `shouldRemind` is still set).
- When the user toggles the `shouldRemind` flag from on to off, the existing notification should be cancelled. The other way around, from off to on, should schedule a new notification.
- When the user deletes a `CheckListItem`, its notification, if it had one, should be cancelled.

- When the user deletes an entire `Checklist`, all the notifications for those items, if there are any, should be cancelled.

This makes it obvious that you don't need just a way to schedule new notifications, but also a way to cancel them.

You should probably also check that you don't create notifications for to-do items whose due dates are in the past. I'm sure iOS is smart enough to ignore those notifications, but let's be good citizens anyway.

Associate to-do items with notifications

We need some way to associate `CheckListItem` objects with their local notifications. This requires some changes to our data model.

When you schedule a local notification, you create a `UNNotificationRequest` object. It is tempting to put the `UNNotificationRequest` object as an instance variable in `CheckListItem`, so you always know what it is. However, this is not the correct approach.

Instead, you'll use an *identifier*. When you create a local notification, you need to give it an identifier, which is just a `String`. It doesn't really matter what is in this string, as long as it is unique for each notification.

To cancel a notification at a later point, you don't use the `UNNotificationRequest` object but the identifier you gave it. The right approach is to store this identifier in the `CheckListItem` object.

Even though the identifier for the local notification is a `String`, you'll give each `CheckListItem` an identifier that is simply a number. You'll also save this item ID in the `Checklists.plist` file. When it's time to schedule or cancel a local notification, you'll turn that number into a string. Then, you can easily find the notification when you have the `CheckListItem` object, or the `CheckListItem` object when you have the notification object.

Assigning numeric IDs to objects is a common approach when creating data models – it is very similar to giving records in a relational database a numeric primary key, if you're familiar with that sort of thing.

► Add these properties to **`CheckListItem.swift`**:

```
var dueDate = Date()
var shouldRemind = false
var itemID: Int
```

Note that you called the last variable `itemID` and not simply “id”. The reason is that `id` is a special keyword in Objective-C, and this could cause trouble if you ever wanted to mix your Swift code with Objective-C code.

The `dueDate` and `shouldRemind` variables have initial values, but `itemID` does not. That’s why you had to specify the type for `itemID` – it’s an `Int` – but not for the other two variables since Swift will infer the type for those based on the initial value.

Xcode will complain at this point since `Checklist` has no initializer which sets up `itemID` and it has no initial value. In order to correct this, you need to add a new method to `DataModel` to generate a unique item ID.

► Hop on over to **DataModel.swift** and add a new method:

```
class func nextChecklistItemID() -> Int {
    let userDefaults = UserDefaults.standard
    let itemID = userDefaults.integer(forKey: "ChecklistItemID")
    userDefaults.set(itemID + 1, forKey: "ChecklistItemID")
    userDefaults.synchronize()
    return itemID
}
```

You’re using your old friend `UserDefaults` again.

This method gets the current “`ChecklistItemID`” value from `UserDefaults`, adds 1 to it, and writes it back to `UserDefaults`. It returns the previous value to the caller.

The method also does `userDefaults.synchronize()` to force `UserDefaults` to write these changes to disk immediately, so they won’t get lost if you kill the app from Xcode before it had a chance to save.

This is important because you never want two or more `ChecklistItems` to get the same ID.

You could add a default value for “`ChecklistItemID`” to the `registerDefaults()` method so as to customize the start value for the item ID, but you really don’t have to in this case :] Remember that if there is no existing value for “`ChecklistItemID`”, you’d get 0 back from a call to `UserDefaults` (if you didn’t provide a default value via `registerDefaults()`). That is good enough for your use since your IDs would then start at 0 and count up.

The first time `nextChecklistItemID()` is called, it will return the ID 0. The second time it is called it will return the ID 1, the third time it will return the ID 2, and so on. The number is incremented by one each time. You can call this method a few billion times before you run out of unique IDs.

Class methods vs. instance methods

If you are wondering why you wrote,

```
class func nextChecklistItemID()
```

and not just:

```
func nextChecklistItemID()
```

then I'm glad you're paying attention. :-)

Adding the `class` keyword means that you can call this method without having a reference to an instance of the `DataModel` object.

With a class method, you do:

```
itemID = DataModel.nextChecklistItemID()
```

Instead of:

```
itemID = dataModel.nextChecklistItemID()
```

This is because `ChecklistItem` objects do not have a `dataModel` property with a reference to a `DataModel` object. You could certainly pass them such a reference, but I decided that using a *class method* was easier.

The declaration of a class method begins with `class func`. This kind of method applies to the class as a whole.

So far you've been using *instance methods*. They just have the word `func` (without `class`) and work only on a specific instance of that class.

We haven't discussed the difference between classes and instances before, and you'll get into that in more detail later in the book. For now, just remember that a method starting with `class func` allows you to call methods on an object even when you don't have a reference to that object.

I had to make a trade-off: is it worth giving each `ChecklistItem` object a reference to the `DataModel` object, or can I get away with a simple class method? To keep things simple, I chose the latter. It's certainly possible that, if you were to develop this app further, it would make more sense to give `ChecklistItem` a `dataModel` property instead.

► Now, switch back to **CheckListItem.swift** and add an `init()` method to fix the initial Xcode error:

```
override init() {  
    itemID = DataModel.nextCheckListItemID()  
    super.init()  
}
```

This asks the `DataModel` object for a new item ID whenever the app creates a new `CheckListItem` object.

Display the new IDs

For a quick test to see if assigning these IDs works, you can add them to the text that is shown in the `CheckListItem` cell label. This is just a temporary thing for testing purposes, as users couldn't care less about the internal identifier of these objects.

► In **ChecklistViewController.swift**, change the `configureText(for:with:)` method to:

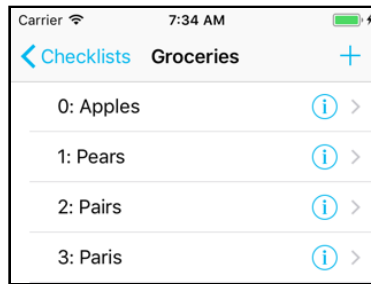
```
func configureText(for cell: UITableViewCell,  
                  with item: CheckListItem) {  
    let label = cell.viewWithTag(1000) as! UILabel  
    //label.text = item.text  
    label.text = "\(item.itemID): \(item.text)"  
}
```

I have commented out the original line because you want to reuse it later. The new one uses `\(...)` to add the to-do item's `itemID` property to the text.

Before you run the app, do note that you have changed the format of the `CheckListItem` (and thus, by extension the `Checklists.plist` file) and so your existing data will not display when you run the app.

► Run the app and add some checklist items. Each new item should get a unique identifier. Exit to the home screen (to make sure everything is saved properly) and stop the app.

Run the app again and add some new items; the IDs for these new items should start counting at where they left off.



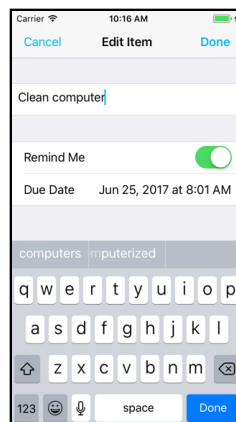
The items with their IDs. Note that the item with ID 3 was deleted in this example.

OK, that takes care of the IDs. Now let's add the “due date” and “should remind” fields to the Add/Edit Item screen.

(Keep `configureText(for:with:)` the way it is for the time being; that will come in handy with testing the notifications.)

Due date UI

You will add settings for the two new fields to the Add/Edit Item screen and make it look like this:



The Add/Edit Item screen now has Remind Me and Due Date fields

The due date field will require some sort of date picker control. iOS comes with a cool date picker view that you'll add into the table view.

The UI changes

➤ Add the following outlets to **ItemDetailViewController.swift**:

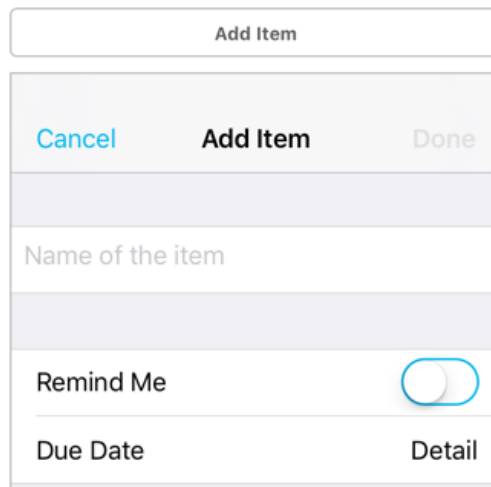
```
@IBOutlet weak var shouldRemindSwitch: UISwitch!
@IBOutlet weak var dueDateLabel: UILabel!
```

➤ Open the storyboard and select the Table View in the Add Item scene.

► Add a new section to the table. The easiest way to do this is to increment the **Sections** field in the **Attributes inspector**. This duplicates the existing section and cell.

► Remove the Text Field from the new cell. Select the new section in the Document Outline and then increment its **Rows** value to 2 in the **Attributes inspector**.

You will now design the new cells to look as follows:



The new design of the Add/Edit Item screen

- Add a **Label** to the first cell and set its text to **Remind Me**. Set the font to **System**, size **17**.
- Also drag a **Switch** control into the cell. Hook it up to the `shouldRemindSwitch` outlet on the view controller. In the Attributes inspector, set its Value to **Off** so it is no longer green.
- Pin the Switch to the **top** and **right** edges of the table view cell. This makes sure the control will be visible regardless of the width of the device's screen.
- The third cell has two labels: Due Date on the left and the label that will hold the actual chosen date on the right. You don't have to add these labels yourself: simply set the **Style** of the cell to **Right Detail** and rename Title to **Due Date**.
- The label on the right should be hooked up to the `dueDateLabel` outlet.

You may need to move the Remind Me label and the switch around a bit to align them nicely with the labels from the “due date” cell. Tip: select the “Due Date” and “Detail” labels and look in the Size inspector what their margins are (should be 16 points from the edges).

Display the due date

Let's write the code for displaying the due date.

- Add a new `dueDate` instance variable to **ItemDetailViewController.swift**:

```
var dueDate = Date()
```

For a new `CheckListItem` item, the due date is right now, i.e. `Date()`. That sounds reasonable but by the time the user has filled in the rest of the fields and pressed Done, that due date will be in the past.

But you do have to suggest something here. An alternative default value could be this time tomorrow, or ten minutes from now, but in most cases the user will have to pick their own due date anyway.

- Add a new `updateDueDateLabel()` method to the file:

```
func updateDueDateLabel() {  
    let formatter = DateFormatter()  
    formatter.dateStyle = .medium  
    formatter.timeStyle = .short  
    dueDateLabel.text = formatter.string(from: dueDate)  
}
```

To convert the `Date` value to text, you use a `DateFormatter` object.

The way it works is very straightforward: you give it a style for the date component and a separate style for the time component, and then ask it to format the `Date` object.

You can play with different styles here, but space in the label is limited. So, you can't fit in the full month name, for example.

The cool thing about `DateFormatter` is that it takes the current locale into consideration so the time will look good to the user no matter where they are on the globe.

- Change `viewDidLoad()` as follows:

```
override func viewDidLoad() {  
    . . .  
    if let item = itemToEdit {  
        . . .  
        shouldRemindSwitch.isOn = item.shouldRemind // add this  
        dueDate = item.dueDate                       // add this  
    }  
  
    updateDueDateLabel() // add this  
}
```

If there already is an existing `CheckListItem` object, you set the switch control to on or off, depending on the value of the object's `shouldRemind` property. If the user is adding a new item, the switch is initially off (you did that in the storyboard).

You also get the due date from the `CheckListItem`.

Update edited values

► The last thing to change in this file is the `done()` action. Replace the current code with:

```
@IBAction func done() {
    if let item = itemToEdit {
        item.text = textField.text!

        item.shouldRemind = shouldRemindSwitch.isOn // add this
        item.dueDate = dueDate                       // add this

        delegate?.itemDetailViewController(self,
                                           didFinishEditing: item)
    } else {
        let item = CheckListItem()
        item.text = textField.text!
        item.checked = false

        item.shouldRemind = shouldRemindSwitch.isOn // add this
        item.dueDate = dueDate                       // add this

        delegate?.itemDetailViewController(self,
                                           didFinishAdding: item)
    }
}
```

Here you put the value of the switch control and the `dueDate` instance variable back into the `CheckListItem` object when the user presses the Done button.

► Run the app and change the position of the switch control. The app will remember this setting when you terminate it (but be sure to exit to the home screen first).

The due date row doesn't really do anything yet, however. In order to make that work, you first have to create a date picker.

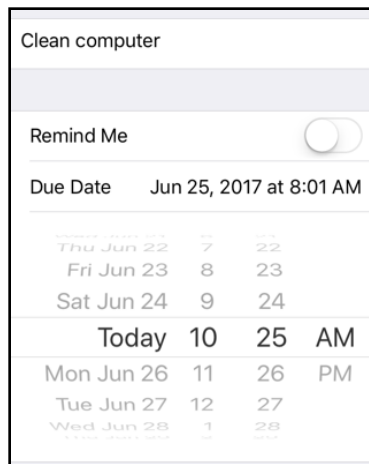
Note: Maybe you're wondering why you're using an instance variable for the `dueDate` but not for `shouldRemind`.

You don't need one for `shouldRemind` because it's easy to get the state of the switch control: you just look at its `isOn` property, which is either `true` or `false`.

However, it is hard to read the chosen date back out of the `dueDateLabel` because the label stores text (a `String`), not a `Date`. So it's easier to keep track of the chosen date separately in a `Date` instance variable.

The date picker

You will not create a new view controller for the date picker. Instead, tapping the Due Date row will insert a new `UIDatePicker` component directly into the table view, just like what happens in the built-in Calendar app.



The date picker in the Add Item screen

► Add a new instance variable to **`ItemDetailViewController.swift`**, to keep track of whether the date picker is currently visible:

```
var datePickerVisible = false
```

► Add the `showDatePicker()` method:

```
func showDatePicker() {
    datePickerVisible = true

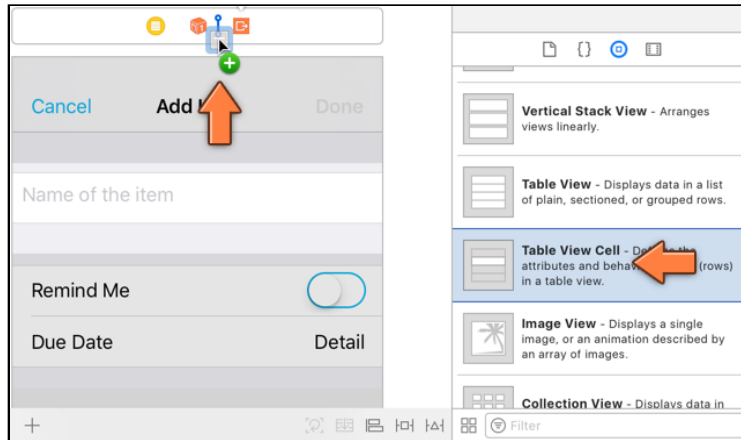
    let indexPathDatePicker = IndexPath(row: 2, section: 1)
    tableView.insertRows(at: [indexPathDatePicker], with: .fade)
}
```

This sets the new instance variable to `true`, and tells the table view to insert a new row below the Due Date cell. This new row will contain the `UIDatePicker`.

The question is: where does the cell for this new date picker row come from? You can't put it into the table view as a static cell already because then it would always be visible. You only want to show it after the user taps the Due Date row.

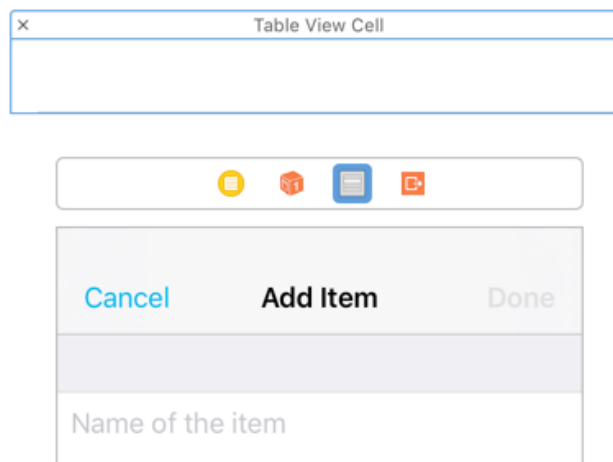
Xcode has a feature where you can add additional views to a scene that are not immediately visible. That's a great solution to this problem!

► Open the storyboard and go to the **Add Item** scene. From the Object Library, pick up a new **Table View Cell**. Don't drag it on to the view controller itself but on to the scene dock at the top:



Dragging a table view cell into the scene dock

Now, the storyboard should look like this:



The new table view cell sits in its own area

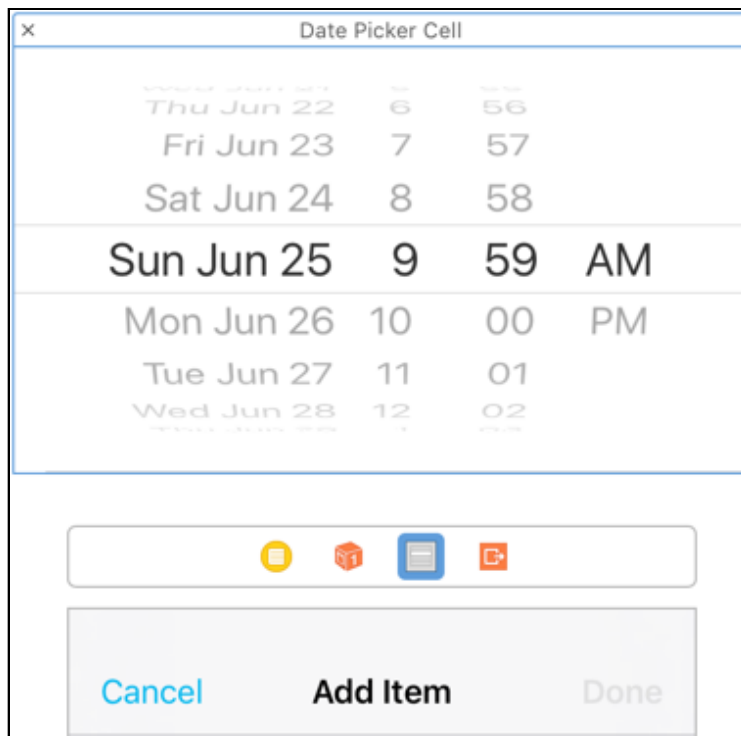
The new Table View Cell object belongs to the scene but it is not (yet) part of the scene's table view.

The cell is a bit too small to fit a date picker, so first you'll make it bigger.

► Select the Table View Cell and in the **Size inspector** set the **Height** to 217. The date picker is 216 points tall, plus one point for the separator line at the bottom of the cell.

- In the **Attributes inspector**, set **Selection** to **None** so this cell won't turn gray when you tap on it.
- From the Object Library, drag a **Date Picker** into the cell. It should fit exactly.
- Use the **Add New Constraints** menu to glue the Date Picker to the four sides of the cell. Turn off **Constrain to margins** and then select the four I-beams to make them red (they all should be 0).

When you're done, the new cell looks like this:



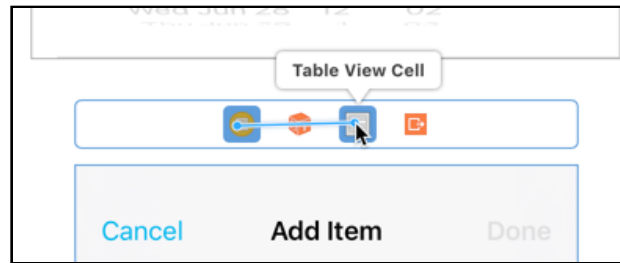
The finished date picker cell

So how do you get this cell into the table view? First, make two new outlets and connect them to the cell and the date picker, respectively. That way you can refer to these views from code.

- Add these lines to **ItemDetailViewController.swift**:

```
@IBOutlet weak var datePickerCell: UITableViewCell!
@IBOutlet weak var datePicker: UIDatePicker!
```

- Switch back to the storyboard and simply Control-drag from the yellow circle icon for the view controller to the gray icon for the Table View Cell, and select the **datePickerCell** outlet.



Control-drag between the icons in the scene dock

► To connect the date picker, Control-drag from the yellow icon to the big Date Picker above it and select the **datePicker** outlet.

Display the date picker

Great! Now that you have outlets for the cell and the date picker inside it, you can write the code to add them to the table view.

Normally, you would implement the `tableView(_:cellForRowAt:)` method, but remember that this screen uses a table view with static cells. Such a table view does not have a data source and therefore does not use `cellForRowAt`.

If you look in **ItemDetailViewController.swift** you won't find that method anywhere. However, with a bit of trickery you can override the data source for a static table view and provide your own methods.

► Add the `tableView(_:cellForRowAt:)` method to **ItemDetailViewController.swift**:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    if indexPath.section == 1 && indexPath.row == 2 {
        return datePickerCell
    } else {
        return super.tableView(tableView, cellForRowAt: indexPath)
    }
}
```

Danger: You shouldn't really mess around too much with this method when it's being used by a static table view, because it may interfere with the inner workings of those static cells. But if you're careful you can get away with it.

The `if` statement checks whether `cellForRowAt` is being called with the index-path for the date picker row. If so, it returns the new `datePickerCell` that you just designed. This is safe to do because the table view from the storyboard doesn't know anything about row 2 in section 1, so you're not interfering with an existing static cell.

For any index-paths that are not the date picker cell, this method will call through to super (which is UITableViewController). This is the trick that makes sure the other static cells still work.

► You also need to override `tableView(_:numberOfRowsInSection:)`:

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    if section == 1 && datePickerVisible {
        return 3
    } else {
        return super.tableView(tableView,
            numberOfRowsInSection: section)
    }
}
```

If the date picker is visible, then section 1 has three rows. If the date picker isn't visible, you can simply pass through to the original data source.

► Likewise, you also need to provide the `tableView(_:heightForRowAt:)` method:

```
override func tableView(_ tableView: UITableView,
    heightForRowAt indexPath: IndexPath) -> CGFloat {
    if indexPath.section == 1 && indexPath.row == 2 {
        return 217
    } else {
        return super.tableView(tableView, heightForRowAt: indexPath)
    }
}
```

So far the cells in your table views all had the same height (44 points), but this is not a hard requirement. By providing the `heightForRowAt` method you can give each cell its own height.

The `UIDatePicker` component is 216 points tall, plus 1 point for the separator line, making for a total row height of 217 points.

The date picker is only made visible after the user taps the Due Date cell, which happens in `tableView(_:didSelectRowAt:)`.

► Add that method:

```
override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
    textField.resignFirstResponder()

    if indexPath.section == 1 && indexPath.row == 1 {
        showDatePicker()
    }
}
```

This calls `showDatePicker()` when the index-path indicates that the Due Date row was tapped. It also hides the on-screen keyboard if that was visible.

Make the Due Date row tappable

At this point you have most of the pieces in place, but the Due Date row isn't actually tap-able yet. That's because `ItemDetailViewController.swift` already has a `willSelectRowAt` method that always returns `nil`, causing taps on all rows to be ignored.

► Change `tableView(_:willSelectRowAt:)` to:

```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    if indexPath.section == 1 && indexPath.row == 1 {
        return indexPath
    } else {
        return nil
    }
}
```

Now the Due Date row responds to taps, but the other rows don't.

► Run the app to try it out. Add a new checklist item and tap the Due Date row.

Oop!. The app crashes. After some investigating, I found that when you override the data source for a static table view cell, you also need to provide the delegate method `tableView(_:indentationLevelForRowAt:)`.

That's not a method you'd typically use, but because you're messing with the data source for a static table view, you do need to override it. I told you this was a little tricky.

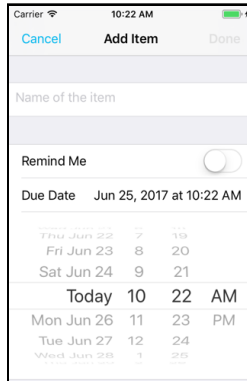
► Add the new delegate method:

```
override func tableView(_ tableView: UITableView,
                        indentationLevelForRowAt indexPath: IndexPath) -> Int {
    var newIndexPath = indexPath
    if indexPath.section == 1 && indexPath.row == 2 {
        newIndexPath = IndexPath(row: 0, section: indexPath.section)
    }
    return super.tableView(tableView,
                          indentationLevelForRowAt: newIndexPath)
}
```

The reason the app crashed on this method was that the standard data source doesn't know anything about the cell at row 2 in section 1 (the one with the date picker), because that cell isn't part of the table view's design in the storyboard.

So after inserting the new date picker cell, the data source gets confused and it crashes the app. To fix this, you have to trick the data source into believing there really are three rows in that section when the date picker is visible.

► Run the app again. This time the date picker cell shows up where it should:



The date picker appears in a new cell

Listen for date picker events

Interacting with the date picker *should* change the date in the Due Date row, but currently this has no effect whatsoever on the Due Date row (try it out: spin the wheels).

You have to listen to the date picker’s “Value Changed” event. That event gets sent whenever the picker wheels settle on a new value. For that, you need to add a new action method.

► Add the action method to **ItemDetailViewController.swift**:

```
@IBAction func dateChanged(_ datePicker: UIDatePicker) {  
    dueDate = datePicker.date  
    updateDueDateLabel()  
}
```

This is pretty simple. It updates the `dueDate` instance variable with the new date and then updates the text on the Due Date label.

► In the storyboard, Control-drag from the Date Picker to the view controller and select the **dateChanged:** action method. Now everything is properly hooked up. (You can verify that the action method is indeed connected to the date picker’s Value Changed event by looking at the Connections inspector.)

► Run the app to try it out. When you turn the wheels on the date picker, the text in the Due Date row updates too. Cool.

However, when you edit an existing to-do item, the date picker does not show the date from that item. It always starts on the current date and time.

➤ Add the following line to the bottom of `showDatePicker()`:

```
datePicker.setDate(dueDate, animated: false)
```

This passes the proper date to the `UIDatePicker` component.

➤ Verify that it works: click on the ⓘ button from an existing to-do item, preferably one you made a while ago, and confirm that the date picker shows the same date and time as the Due Date label. Excellent!

Change the date label color when the date picker is active

Speaking of the label, it would be nice if this becomes highlighted when the date picker is active. You can use the tint color for this (that's also what the Calendar app does).

➤ Replace the contents of `showDatePicker()` with this:

```
func showDatePicker() {
    datePickerVisible = true

    let indexPathDataRow = IndexPath(row: 1, section: 1)
    let indexPathDatePicker = IndexPath(row: 2, section: 1)

    if let dateCell = tableView.cellForRow(at: indexPathDataRow) {
        dateCell.detailTextLabel!.textColor =
            dateCell.detailTextLabel!.tintColor
    }

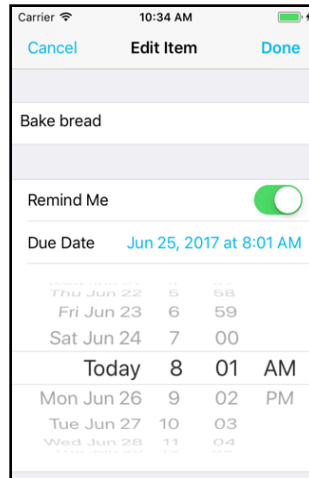
    tableView.beginUpdates()
    tableView.insertRows(at: [indexPathDatePicker], with: .fade)
    tableView.reloadRows(at: [indexPathDataRow], with: .none)
    tableView.endUpdates()

    datePicker.setDate(dueDate, animated: false)
}
```

This sets the `textColor` of the `detailTextLabel` to the tint color. It also tells the table view to reload the Due Date row. Without that, the separator lines between the cells don't update properly.

Because you're doing two operations on the table view at the same time – inserting a new row and reloading another – you need to put that code in between calls to `beginUpdates()` and `endUpdates()`, so that the table view can animate everything at the same time.

► Run the app. The date now appears in blue:



The date label appears in the tint color while the date picker is visible

Hide the date picker

When the user taps the Due Date row again, the date picker should disappear. If you try that right now the app will crash – what did you expect? This won't win it many favorable reviews.

► Add a new `hideDatePicker()` method:

```
func hideDatePicker() {
    if datePickerVisible {
        datePickerVisible = false

        let indexPathDataRow = IndexPath(row: 1, section: 1)
        let indexPathDatePicker = IndexPath(row: 2, section: 1)

        if let cell = tableView.cellForRow(at: indexPathDataRow) {
            cell.detailTextLabel!.textColor = UIColor.black
        }
        tableView.beginUpdates()
        tableView.reloadRows(at: [indexPathDataRow], with: .none)
        tableView.deleteRows(at: [indexPathDatePicker], with: .fade)
        tableView.endUpdates()
    }
}
```

This does the opposite of `showDatePicker()`. It deletes the date picker cell from the table view and restores the color of the date label to the original color.

► Change `tableView(_:didSelectRowAt:)` to toggle between the visible and hidden states:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
```

```
        if indexPath.section == 1 && indexPath.row == 1 {  
            if !datePickerVisible {  
                showDatePicker()  
            } else {  
                hideDatePicker()  
            }  
        }  
    }  
}
```

There is another situation where it's a good idea to hide the date picker: when the user taps inside the text field.

It won't look very nice if the keyboard partially overlaps the date picker, so you might as well hide it. The view controller is already the delegate for the text field, making this easy.

► Add the `textFieldDidBeginEditing(_:)` method:

```
func textFieldDidBeginEditing(_ textField: UITextField) {  
    hideDatePicker()  
}
```

And with that you have a cool inline date picker!

► Run the app and verify that hiding the date picker works for both scenarios.

Schedule local notifications

One of the principles of object-oriented programming is that objects should do as much as possible themselves. Therefore, it makes sense that the `CheckListItem` object should schedule its own notifications.

Schedule notifications

► Add the following method to **CheckListItem.swift**:

```
func scheduleNotification() {  
    if shouldRemind && dueDate > Date() {  
        print("We should schedule a notification!")  
    }  
}
```

This compares the due date on the item with the current date. You can always get the current time by making a new `Date` object.

The statement `dueDate > Date()` compares the two dates and returns `true` if `dueDate` is in the future and `false` if it is in the past.

If the due date is in the past, the `print()` will not be performed.

Note the use of the `&&` “and” operator. You only print the text when the Remind Me switch is set to “on” *and* the due date is in the future.

You will call this method when the user presses the Done button after adding or editing a to-do item.

► In the `done()` action in **ItemDetailViewController.swift**, add the following line just before the call to `didFinishEditing` and also before `didFinishAdding`:

```
item.scheduleNotification()
```

► Run the app and try it out. Add a new item, set the switch to ON but don’t change the due date. Press Done.

There should be no message in the Console because the due date has already passed (it is several seconds in the past by the time you press Done).

► Add another item, set the switch to ON, and choose a due date in the future.

When you press Done now, the text “We should schedule a notification!” should appear in the Console.

Now that you’ve verified the method is called in the proper place, let’s actually schedule a new local notification object for the following three scenarios: adding a to-do item, editing a to-do item, deleting a to-do item.

Add a to-do item

► In **ChecklistItem.swift**, change `scheduleNotification()` to:

```
func scheduleNotification() {
    if shouldRemind && dueDate > Date() {
        // 1
        let content = UNMutableNotificationContent()
        content.title = "Reminder:"
        content.body = text
        content.sound = UNNotificationSound.default()

        // 2
        let calendar = Calendar(identifier: .gregorian)
        let components = calendar.dateComponents(
            [.month, .day, .hour, .minute],
            from: dueDate)

        // 3
        let trigger = UNCalendarNotificationTrigger(
            dateMatching: components,
            repeats: false)

        // 4
```

```
let request = UNNotificationRequest(
    identifier: "\(itemID)", content: content,
    trigger: trigger)
// 5
let center = UNUserNotificationCenter.current()
center.add(request)

print("Scheduled: \(request) for itemID: \(itemID)")
}
```

You’ve seen this code before when you tried out local notifications for the first time, but there are a few differences.

1. Put the item’s text into the notification message.
2. Extract the month, day, hour, and minute from the `dueDate`. We don’t care about the year or the number of seconds – the notification doesn’t need to be scheduled with millisecond precision, on the minute is precise enough.
3. To test local notifications you used a `UNTimeIntervalNotificationTrigger`, which scheduled the notification to appear after a number of seconds. Here, you’re using a `UNCalendarNotificationTrigger`, which shows the notification at the specified date.
4. Create the `UNNotificationRequest` object. Important here is that we convert the item’s numeric ID into a `String` and use it to identify the notification. That is how you’ll be able to find this notification later in case you need to cancel it.
5. Add the new notification to the `UNUserNotificationCenter`.

Xcode is not so impressed with this new code and gives a bunch of error messages.

What is wrong here? `UNUserNotificationCenter` and the other objects are provided by the User Notifications framework – you can tell by the “UN” prefix in their names.

However, `CheckListItem` hasn’t used any code from that framework until now. The only framework objects it has used, `NSObject` and `Codable`, came from another framework, `Foundation`.

► To tell `CheckListItem` about the User Notifications framework, you need to add the following line to the top of the file, below the other import:

```
import UserNotifications
```

Now the errors disappear like snow in the sun.

There’s another small problem, though. If you’ve reset the Simulator recently, then the app no longer has permission to send local notifications.

► Try it out. Run the app, add a new checklist item, set the due date a minute into the future, and press Done. You might not see a notification.

Even if you do see a notification, since the authorization request code is no longer there, *Checklists* certainly won't have permission on your user devices.

When you were just messing around at the beginning of this chapter, you placed the permission request code in the AppDelegate and ran it immediately upon launch. That's not recommended.

Don't you just hate those apps that prompt you for ten different things before you've even had a chance to properly look at them? Let's be a bit more user friendly with our own app!

► Add the following method to **ItemDetailViewController.swift**:

```
@IBAction func shouldRemindToggled(_ switchControl: UISwitch) {
    textField.resignFirstResponder()

    if switchControl.isOn {
        let center = UNUserNotificationCenter.current()
        center.requestAuthorization(options: [.alert, .sound]) {
            granted, error in
                // do nothing
        }
    }
}
```

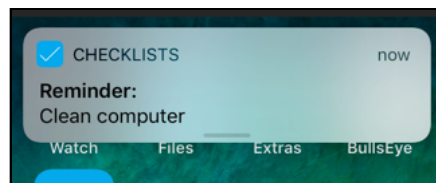
When the switch is toggled to ON, this prompts the user for permission to send local notifications. Once the user has given permission, the app won't put up a prompt again.

► Also add an import `UserNotifications` or the above method won't compile.

► Open the storyboard and connect the **shouldRemindToggled:** action to the switch control.

► Test it out. Run the app, add a new checklist item, set the due date a minute into the future, press Done and exit to the home screen.

Wait one minute (patience...) and the notification should appear. Pretty cool!



The local notification when the app is in the background

That takes care of the adding a new item scenario. There are two others left.

Edit an existing item

When the user edits an item, the following situations can occur with the Remind Me switch:

- Remind Me was switched off and is now switched on. You have to schedule a new notification.
- Remind Me was switched on and is now switched off. You have to cancel the existing notification.
- Remind Me stays switched on but the due date changes. You have to cancel the existing notification and schedule a new one.
- Remind Me stays switched on but the due date doesn't change. You don't have to do anything.
- Remind Me stays switched off. Here you also don't have to do anything.

Of course, in all those situations you'll only schedule the notification if the due date is in the future.

Phew, that's quite a list. It's always a good idea to take stock of all possible scenarios before you start programming because this gives you a clear picture of everything you need to tackle.

It may seem like you need to write a lot of logic here to deal with all these situations, but actually it turns out to be quite simple.

First you'll look if there is an existing notification for this to-do item. If there is, you simply cancel it. Then you determine whether the item should have a notification and if so, you schedule a new one.

That should take care of all the above situations, even if sometimes you simply could have left the existing notification alone. The algorithm is crude, but effective.

► Add the following method to **ChecklistItem.swift**:

```
func removeNotification() {  
    let center = UNUserNotificationCenter.current()  
    center.removePendingNotificationRequests(  
        withIdentifiers: ["\(itemID)"]  
    )  
}
```

This removes the local notification for this `ChecklistItem`, if it exists. Note that `removePendingNotificationRequests()` requires an array of identifiers, so we first put our `itemID` into a string with `\(...)` and then into an array using `[]`.

► Call this new method from to the top of `scheduleNotification()`:

```
func scheduleNotification() {  
    removeNotification()  
    . . .  
}
```

Let's try it out.

- Run the app and add a to-do item with a due date two minutes into the future. A new notification will be scheduled. Go to the home screen and wait until it shows up.
- Edit the item and change the due date to three minutes into the future. The old notification will be removed and a new one scheduled for the new time.
- Add a new to-do item with a due date two minutes into the future. Edit the to-do item but now set the switch to OFF. The old notification will be removed and no new notification will be scheduled.
- Edit again and put the time a few minutes into the future but don't change anything else; no new notification will be scheduled because the switch is still off.

These tests should also work if you terminate the app in between.

Delete a to-do item

There is one last case to handle: deletion of a `ChecklistItem`. This can happen in two ways:

1. The user can delete an individual item using swipe-to-delete.
2. The user can delete an entire checklist, in which case all its `ChecklistItem` objects are also deleted.

An object is notified when it is about to be deleted using the `deinit` message. You can simply implement this method, look if there is a scheduled notification for this item, and then cancel it.

► Add the following to **`ChecklistItem.swift`**:

```
deinit {  
    removeNotification()  
}
```

That's all you have to do. The special `deinit` method will be invoked when you delete an individual `ChecklistItem` but also when you delete a whole `Checklist` – because all its `ChecklistItems` will be destroyed as well, as the array they are in is deallocated.

► Run the app and try it out. First, schedule some notifications a minute or so into the future and then remove that to-do item or its entire checklist. Wait until the due date comes and you shouldn't get a notification.

Once you're convinced everything works, you can remove the `print()` statements. They are only temporary for debugging purposes. You probably don't want to leave them in the final app. The `print()` statements won't hurt, but the end user can't see those messages anyway.

► Also remove the item ID from the label in the `ChecklistViewController` – that was only used for debugging.

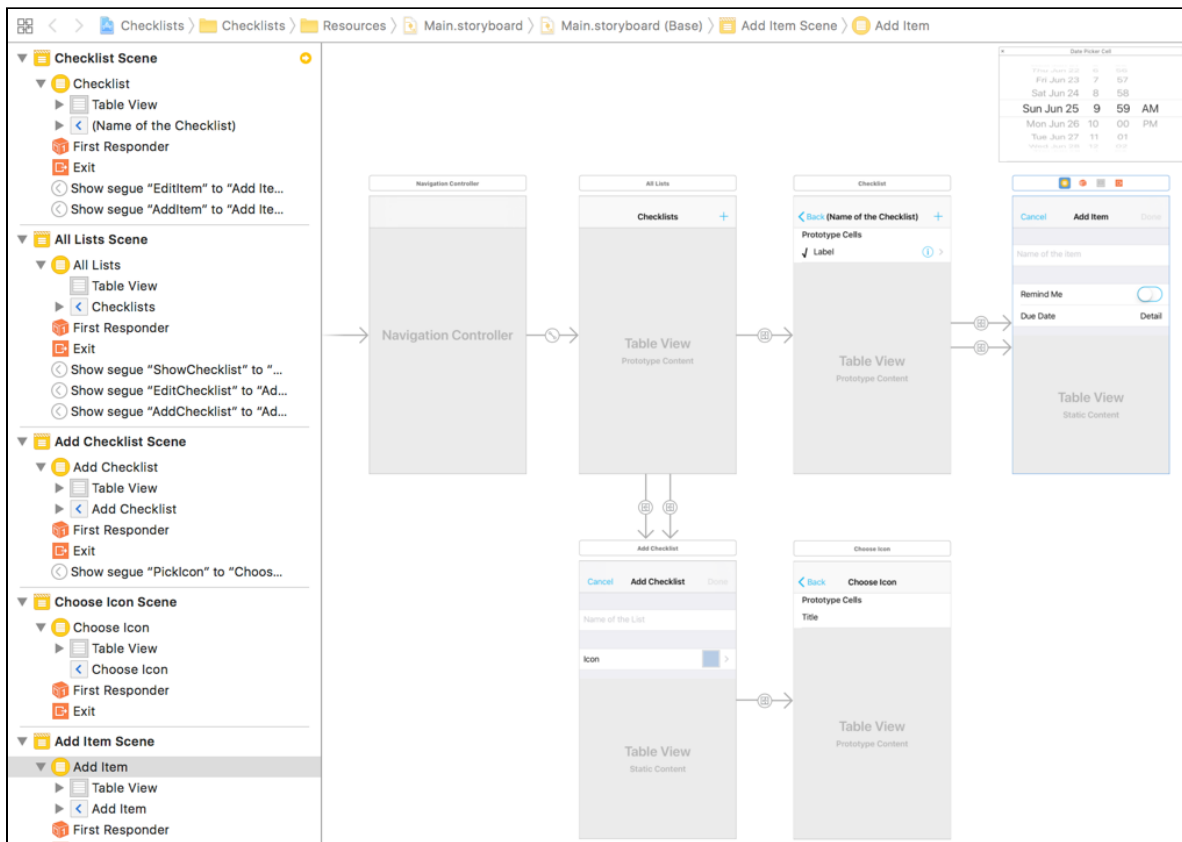
That's a wrap!

Things should be starting to make sense by now. I've thrown you into the deep end by writing an entire app from scratch. We've touched on a number of advanced topics already, but hopefully you were able to follow along quite well with what we've been doing. Kudos for sticking with it until the end!

It's OK 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 do this app again from the start – it will make more sense the second time around!

This section focused mainly on UIKit and its most important controls and patterns. In the next section we'll take a few steps back to talk more about the Swift language itself. And of course, you'll build another cool app.

Here is the final storyboard for *Checklists*:



The final storyboard

Completing all of that is pretty impressive! Give yourself a well-deserved pat on the back :]

Take a break, and when you're ready, continue on to the next section, where you'll make an app that knows its place! :-)

Haven't had enough yet? Here are some challenges to sink your teeth into:

Exercise: Display the due date in the table view cells, under the text of the to-do item.

Exercise: Sort the to-do items list based on the due date. This is similar to what you did with the list of Checklists except that now you're sorting ChecklistItem objects and you'll be comparing Date objects instead of strings.

You can find the final project files for the Checklists app under **33 - Local Notifications** in the Source Code folder.

Conclusion

Congratulations on making it through the first part of our iOS 11 and Swift 4 for Beginners course!

At this point, you have built two complete iOS apps from scratch, and you've learned the basics of programming in Swift. Your iOS adventure is well underway.

But don't worry — there's more! The course covers additional topics, including Auto Layout, Collection Views, Scroll Views, networking, firebase, git, Xcode Tips & Tricks, and lots of other technologies you'll master on your way to becoming a knowledgeable developer.

Also, if you enjoyed this book, you might want to get the full copies of **The iOS Apprentice** and **The Swift Apprentice**, which go far beyond the scope of this course into more advanced concepts.

The iOS Apprentice

[The iOS Apprentice](#) covers how to build 4 complete iOS apps via step-by-step tutorials.



The full version of the iOS Apprentice builds on what you've learned to build even more exciting iOS apps:

- Learn how to build a GPS-based mapping application that saves a user's favorite locations, displays them on a map, uses the iPhone's camera to take pictures and save all this information to a database.
- Then, build an app that uses live data from the iTunes store to help the user browse music selections. You'll learn how to work with remote APIs, how to interpret the data sent back from network requests, and how to display that information back to the user.
- You'll also learn how to make your apps work in multiple languages around the world, how to adopt your apps for the iPad, and even how to submit your apps to the App Store for sale!

You can get the full version of the book here:

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

The Swift Apprentice

[The Swift Apprentice](#): Covers the Swift 4 language in detail, from tutorials on fundamental programming concepts.



The full version of the Swift Apprentice goes even deeper with the language to help you become an expert at one of the most modern and fastest-growing programming languages today. You'll learn about advanced language features including the following:

- Enumerations, protocols and generics
- Pattern matching

- Advanced error handling
- Encoding and decoding information
- And much more!

You can get the full version of the Swift Apprentice here:

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

You can find both of these books, along with books on animations, developing for the Apple TV and the Apple Watch, as well as game development, at our official store:

- <http://store.raywenderlich.com>

Thank you again for enrolling in our course. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

— Ray, Fahim, Matt, Eli, Matthijs, Ben, Cosmin, and Steven

The Course Companion Book team