The Pragmatic Programmers

Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson
edited by Tammy Coron

What Readers Are Saying About
Xcode Treasures

No matter how long you've been writing iOS and Mac apps using Xcode, you will find a ton of treasures in this book. Some of them will be things that you've used in the past and forgotten, but many will have you saying "Wow, I didn't know Xcode could do that."

➤ **Daniel H. Steinberg**
Storyteller, Dim Sum Thinking

Xcode is a fairly deep program. Most developers barely scrape the surface and only occasionally go down the rabbit hole. Whether you're brand new to Apple platforms development or a seasoned code slinger, this book has something for you. You'll discover insights on app versioning, alternate views on Storyboard, and a wealth of debugging and provisioning tips too. Xcode Treasures will make a valuable addition to your tool belt.

➤ **Tim Mitra**
President, iT Guy Technologies

We've left this page blank to make the page numbers the same in the electronic and paper books.

We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say “hello.”

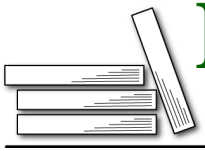
Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Development Editor: Tammy Coron
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-586-3
Book version: P1.0—October 2018

Contents

Acknowledgments

This book is entirely different than all the others I've worked on. Instead of walking through the essential APIs of an SDK such as Cocoa Touch or Core Audio, this book digs into the development tool itself, and the techniques that go with using it. That's required a very different approach to writing and producing the book, something I couldn't have done without a lot of help.

So first off, thanks to the Pragmatic Bookshelf for pushing me to roll the dice on something different in the first place. It might have been easy enough just to continue polishing the beginners' SDK book, but instead, they asked for ideas that would get a little further off that well-trod path. By trying something different, we've created something that will be useful to the intermediate or advanced developer who doesn't need to be reintroduced to UIViewController every year. Within the company, thanks as always to Andy and the now-retired Dave for running the show and creating a publishing system that helps authors focus on prose and code, not formatting. Susannah Davidson Pfalzer signed the book, and Brian MacDonald kept it going, so thanks to both of them. And thanks to Janet Furlow, who I badgered for years on a weekly basis to update the web page with my latest code-along livestream video from this book or the previous one.

Big thanks to my editor, Tammy Coron, who had to deal with this book being written in a random-access fashion, as I started on whatever chapter I felt I was ready for at any particular moment. As a result, she had to have faith that the prerequisites would eventually be satisfied for a chapter near the end of the book but written early in the process. She also pushed me to reorganize the chapters to make for a more sensible whole, caught me when I tried to hand-wave around tricky parts and made sure the whole thing lives up to what it claims to be. It's been more rigorous than another run through the beginner title would have been, but that rigor has made for a better book. Lucky for me she's so well organized.

Thank you to the technical reviewers who provided early feedback on the nitty-gritty details: Mark Dalrymple, Steve Hill, Daniel Jalkut, Jeff Kelley, Erica Sadun, and Daniel Steinberg. Also, thanks to everyone who submitted suggestions and mistakes on the book's errata page, https://pragprog.com/titles/caxcode/errata.

Finally, I want to reserve the last bit of thanks for the Xcode team at Apple. The motivation for this book came in part due to my frustration with the endless potshots, and hot takes hurled at them on social media, which I'm pretty sure they're not actually allowed to respond to publicly. It takes professionalism to look past that kind of abuse and keep at it. And the more you code, especially as you code bigger and harder things, the more you can and should appreciate what Xcode has accomplished. When I start to think of how one would even build something like a visual storyboard editor, and then realize that's just one of many features in Xcode, I can't help but be a little awed.

Obligatory end-of-book music check: this time it was X Japan, Coeur de Pirate, Kate Nash, Bob Seger, Sarah Slean, and the soundtracks to *Schwarzesmarken* and *Kakegurui*. Current musical stats at http://www.last.fm/user/invalidname.

Introduction

A good craftsperson doesn't blame their tools… but you wouldn't know this from the way some developers talk.

Hop on Twitter or stop by your local CocoaHeads meeting, and someone's sure to be complaining about *something*. A minor annoyance here, a show-stopper there, a grudge from years ago that needs to be nursed now… it all leads to a constant sense of negativity about the work we do as developers.

A lot of that griping is directed at Xcode, the tool that's more or less indispensable when developing for Apple platforms. Just search for `xcode` on Twitter and you'll see screeds and tweetstorms damning Xcode for every sin a developer tool can commit—as if the app had been developed by the Marquis de Sade himself, for the sole purpose of torturing iOS developers.

But… come closer… let me blow your mind for a minute.

Xcode is good, actually.

Been There, Done That

As a longtime developer, I can be pretty jaded about things. But when it comes to developer tools, it's not the tools I've grown weary of, it's the complaining.

Before getting into Apple development, I used to edit a developer website for Sun Microsystems and O'Reilly Media called `java.net`. This was a site with feature articles, news items, and a repository of open source projects. Because Sun's stuff was on there, we naturally did a lot of coverage of their IDE, which was called NetBeans. But within the Java community at the time, there was a holy war between NetBeans and another IDE named Eclipse.

The Eclipse fans were not always diplomatic in their distaste for NetBeans, which is a diplomatic way of saying they were vulgar, toxic zealots about it. They didn't just dislike NetBeans, they *hated* it. They hated the application, hated the people who worked on it, and hated anyone who said they liked it. They made everything personal. It wasn't about the difference between

lightweight and heavyweight rendering of UI components, it was about US VERSUS THEM.

But over the years, a funny thing happened. Android hit the scene and initially adopted Eclipse as its IDE of choice. Yet, in recent years Google has pushed it aside in favor of a new IDE, "Android Studio." And all of a sudden, I see developers on that side of the fence saying how great Studio is compared to Eclipse. And how Eclipse sucks. And how Eclipse has *always* sucked.

To which I'm like… "Really? Because you pretty much used to say that I'd be committing professional malpractice if I *didn't* use Eclipse."

This was the point at which I officially stopped listening to the mob when it comes to the relative merits of IDEs.

What's in an IDE?

I keep using that acronym: IDE. Let's stop for a second and look at what we're even talking about. An IDE is an *integrated development environment*. In other words, it brings together multiple tools for developers and integrates them so they work well together, often all within a single application.

For the most part, you don't actually have to use an IDE at all. Look at what goes into an iOS or macOS app:

- You edit source code.
- You build UIs with a visual editor, although you can also build UIs in code.
- You compile the source into executable code, and you link it with libraries and frameworks.
- You collect the executables, along with resource files like graphics, sound, localizations, metadata, etc., into an app bundle, which is just a folder that adheres to a known format.
- You use a code-signing tool to cryptographically prove your identity as the creator of the app.
- You distribute this to users or upload it to Apple for review and publication on their App Store.

The thing is, pretty much every one of these steps can be performed individually, with separate applications. You can write your source in any number of popular text editors, compile it on the command line with `clang` and `swiftc`, use scripts to assemble the bundle, and use Apple's `codesign` tool to sign the

result and ready it for Gatekeeper or the App Store. Then you upload it with a browser to submit it for Apple's review.

And yet, almost nobody does this. You can own your workflow, but it's way too much work for way too little benefit.

In fact, consider the unappreciated benefits of combining these into a single application:

- When the UI builder and the source editor are in the same app, the UI builder can insert code to create your IBActions and IBOutlets for you.
- When the compiler gets a live look at the code, it can offer accurate auto-completion as you type, not based merely on the matching string fragments, but by figuring out what will and will not compile at your insertion point. As a bonus, the compiler can also show you warnings and errors as you type.
- To top it off, the code-signing process, which used to be a manual mess of tools, can be made automatic and actually do the right thing… most of the time, anyway.

This isn't just more convenient, it's more powerful, in substantial and interesting ways.

No Regrets

Now think about how you use Xcode. There are a ton of menu items you've never used, right? What about those project settings that get filled in for you? Have you ever really given them a second look? The scheme selector that's never been anything more than a way to switch simulators for build-and-run; what more can it do? Those weird little inspectors all over the right pane in storyboards; what's their purpose?

Isn't it possible there's a whole lot of power to this app that you've never used, because you didn't know it was there, and never needed to look for it?

Isn't it possible that at some point you've said "I wish Xcode did *X*", and that maybe Xcode actually *does* do *X*, and you just didn't know?

That's the purpose of this book: to go beyond the basics; to take an in-depth look at the many different areas of Xcode and show you the neat stuff you may have missed. It's a collection of handy tricks, clever customizations, and power techniques that will leave you wondering how you ever overlooked them or lived without them.

About This Book

This is a book about the tool Xcode, not about the software development kits (SDKs) for the various Apple platforms. That means you should *not* make this your first book for learning iOS or macOS programming. This book is for developers who know the SDKs, at least a little, and want to get better at making apps.

Notice that the preceding paragraph said SDKs… plural. Xcode supports development for four platforms—iOS, macOS, tvOS, and watchOS—and this book embraces that diversity. You'll find examples for all the Apple platforms, most obviously in Chapter 11, *Platform Specifics*, on page 207, but also wherever it makes the most sense for the material. For example, in Chapter 2, *Storyboards: Appearance*, on page 23, the examples use Mac storyboards to work with Auto Layout because it's easier to see layout changes by resizing windows than by running on differently sized iPhone simulators. Don't worry; Apple platforms share so much code in their SDKs, it's not that hard for an iOS developer to get the hang of macOS, and vice versa.

Similarly, the book will use whatever Xcode-supported programming language best illustrates the material. While Swift is the default for most examples, some will use Objective-C or C, as is the case in Chapter 6, *Debugging Code*, on page 109 when inspecting memory buffers directly in the debugger, or catching errors that are only possible in those languages.

Each chapter takes on a more or less distinct topic, with little or no dependency on earlier material, so feel free to hop to whatever catches your interest from the table of contents. You don't have to read the whole thing in order if you don't want to.

Expectations and Technical Requirements

This book assumes you have some familiarity with developing for one or more of the Apple platforms: macOS, iOS, tvOS, or watchOS. Presumably, you've at least finished an introductory book; if not, may I recommend *iOS 10 SDK Development [Cla17]* to learn the iOS SDK. If you'd also like to take a deep dive into the Swift language itself, in a more platform-agnostic form, take a look at *A Swift Kickstart [Ste17]*.

The book uses the current version of Xcode, which at the time of this writing is Xcode 9. Historically, Xcode only runs on the current version of macOS

(and for a limited time, on the previous version as well), which at the time of this writing is macOS High Sierra (10.13).

Online Resources

You may find the code for this book on the book's Pragmatic Bookshelf website.[1] If you find a problem with the text, please report it using the errata submission form.

1. https://pragprog.com/book/caxcode

CHAPTER 1

Projects

The journey of a thousand five-star reviews starts with a single command-shift-N… or something like that.

When you start development on an app, you start with a project file. From there, you usually dive right in and start throwing down storyboards or view controller code. But what if you slow down a second and take a look around?

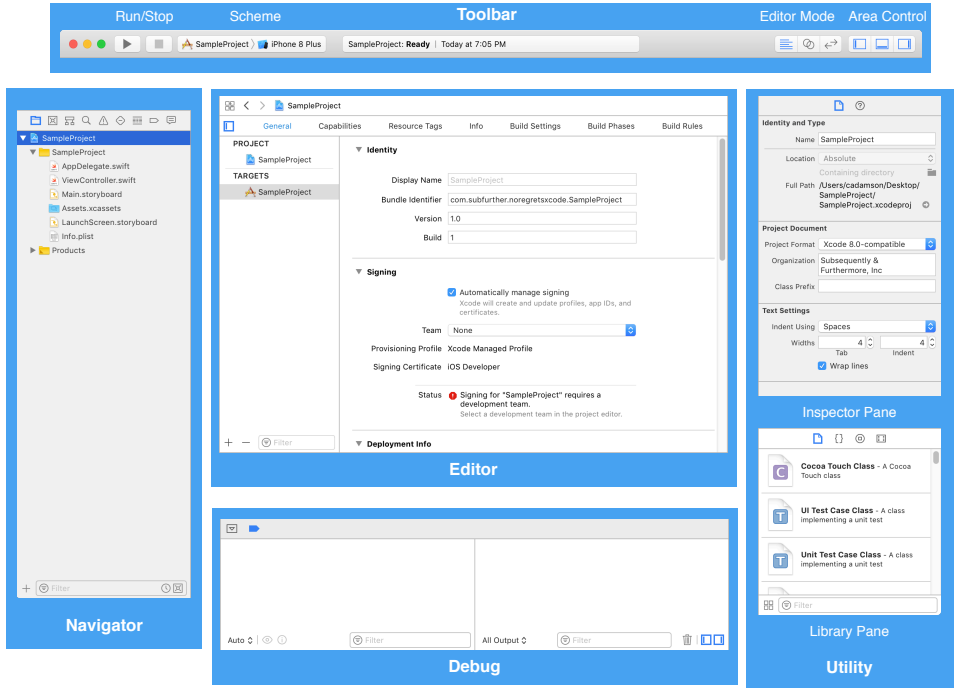
Believe it or not, the Xcode project is doing a lot for us. If you take some time now to understand where the various settings are—and what defaults have already been set—you can avoid getting mad later when Xcode does something stupid and you don't understand why.

Understanding Projects and Files

When you create or open a project, Xcode displays the project in a single window view. This view has panes on the right, left, and bottom, which you can choose to show or hide. It also has an always-visible content area at the center, and all of the essential tools for building, running, and managing the window are located in a toolbar at the top. The figure on page 2 shows this arrangement.

To quickly recap, here are the major regions of the Xcode project window:

- *Toolbar*: Contains build and run buttons, the scheme selector—used to determine which target to build, and for what destination (e.g., a device or the simulator), an iTunes-like status display, an editor mode switch (regular, assistant, or SCM), and controls to show or hide the left, bottom, and right panes.
- *Navigator*: Shows top-level navigation of project contents. Making a selection here resets the contents of the editor in the center. A small toolbar switches between navigators. By default, you use the *File Navigator* (⌘1)

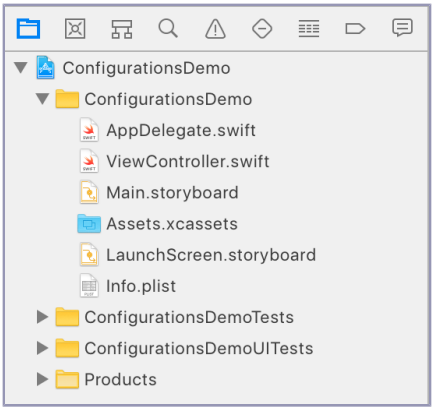


to choose files, but there are also navigators to browse breakpoints and unit tests, examine build errors and warnings, etc.

- *Editor*: A content-specific editing UI for whatever is selected in the navigator. For example, if you select a source file, this now becomes a source editor. If you select a build from the Report Navigator, the editor shows the build log.
- *Utilities*: This area at the right is split into two sub-panes. At the top, the *Inspectors* let you inspect information and settings for source files, user-interface elements, storyboard connections, etc., given the current selection in the editor. At the bottom, *Utility Pane* offers reusable code snippets, user-interface objects, and other conveniences.
- *Debug Area*: The bottom pane is also split into two panes (with controls to show either or both). The left side is the *Variables View*, which shows variables in scope when you're stopped on a breakpoint, and is empty otherwise. The right is the *Console*, which shows log output, and accepts debugger commands when stopped on a breakpoint.

Looking at the big picture, an Xcode project keeps track of two things:

1. What files are used to create the app (or framework, command-line utility, iMessage sticker pack, what have you…)
2. What Xcode is supposed to *do* with those files.

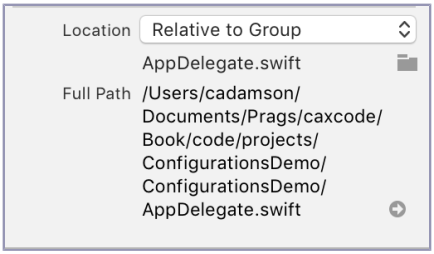


The contents of the project are shown in the File Navigator in the left pane, which is surely familiar to all Xcode users. It's a tree-structure list view, whose root is an icon for the Xcode project itself. Under the root, there are child groups for the source files for the app and its tests. Then, at the bottom, there's a "Products" folder representing the locations of files that have been built.

If you've ever had an Xcode project under source control—something we'll talk about more in Chapter 10, *Source Control Management*, on page 189—you may have noticed that adding, deleting, or re-organizing files dirties the project; that is to say, it changes the project in a way that is different from what was last saved. You can tell, because these actions put an "M" (for "modified") next to the project icon at the top of the File Navigator. And that makes sense, based on the first principle above: you've changed the content of the project, and that has to be stored somewhere.

The project's ..xcodeproj "file" itself is actually a *bundle*: a directory with an extension that tells the Finder to act like it's a single file. You can browse into it with the command line, or the Finder's "Show Bundle Contents" command, and you'll see that it keeps track of files, what groups they're in, what targets they're part of, and so on.

To show details about any file in the project, select it in the File Navigator, and then bring up the File Inspector (⌥⌘1) in the right pane. This inspector starts with the file's type, which has important consequences for source files that you'll look at in Chapter 5, *Building Projects*, on page 89.



Right below the file type, there's a "Location" pop-up and the full path of the file. There are also two small icon buttons: a folder icon that lets you reset the file location by browsing to choose a different file, and a circle-arrow that reveals the file in the Finder. The pop-up lets you change

how Xcode finds the file: whether the filename is relative to its group, relative to something else (like an SDK), or an absolute path.

"Relative to Group" is almost always the right option, especially if you share your projects with other developers. After all, an absolute path that starts /Users/cadamson/Documents might work for me, but probably won't for you.

When Absolute Paths Make Sense

Absolute paths aren't *always* wrong. A few years ago, Apple told Core Audio developers to put some helper files in a known location under /Developer. In that scenario, an absolute path made sense, because anyone using the project would have the same helper files in the same place on his or her machine. I also recently wrote a demo project that grabs the Mac system sounds that are always installed in /System/Library/Sounds. But scenarios like these are really rare.

It's possible to damage a project by renaming or moving files in the Finder. Doing so will break the relative path that Xcode expects, and Xcode will start showing the filename with a red label in the File Navigator. To fix this problem, click the File Inspector's folder icon to show a file-navigator dialog. Then, re-select the moved or renamed file.

Working with Project and Target Settings

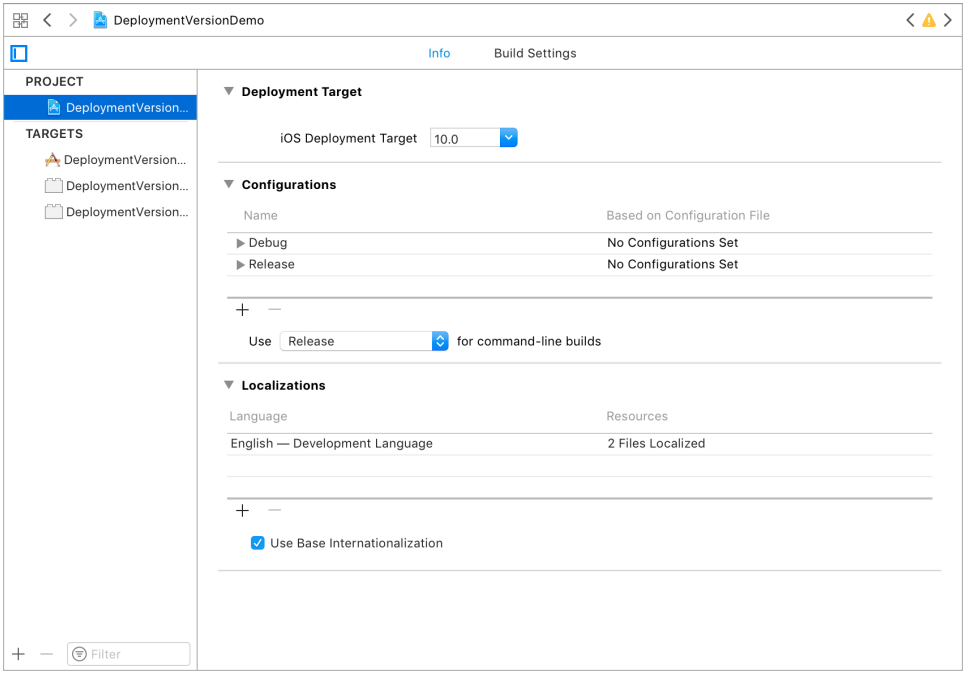
So yeah, Xcode can keep track of the files in your project, but how does it *use* them? To find out, you can click the project icon in the File Navigator, which brings up the settings UI in the Editor, as seen in the figure on page 5.

This editor can show a list of the project and its targets on the left (as shown in the figure), or expose just a pop-up to switch between them; toggle the two with the icon shown at the right if you need to. Personally, I prefer to always show the list, though this might not be practical if you have a very small display. Selecting the project or one of the targets will fill the editor with a tabbed editor of many, many, *many* properties and settings.

The project and the targets have different tabs of settings you can change. The project is simpler, showing a fairly simple Info tab, and a more detailed Build Settings tab. The app target has many more tabs, which you'll get into shortly.

Setting a Deployment Target and Base SDK

The project's Info tab shows one of the most important settings of all: the *deployment target*. This is the earliest version of the OS that you intend for



your app to support. By default, it'll be the version of iOS, macOS, tvOS, or watchOS that was current when this version of Xcode was released. So, for Xcode 9, it will default to iOS 11 or macOS 10.13.

Is that the right choice for your app? Maybe. But you might be leaving behind users who haven't updated their devices, and users with really old devices that *can't* update. That can be a good thing: if your app has significant CPU or memory needs, dropping a small number of very old devices may be worth it, if it means you can better support the majority of your users. Version targeting is an art, not a science.

But this isn't the only place where you need to think about the users' OS version. Switch over to the Build Settings tab, and find the setting for "Base SDK". The list of settings is long, so you may want to type `Base` in the filter field at the upper right to find it. By default, this will also be the current version of the OS as of this Xcode version.

These are two different things, which is why they are two different settings. Base SDK indicates what version of the SDK you build your code for, while Deployment Target is what version of the OS you expect to run on.

One common strategy that works well on iOS is to support the current version, plus one version behind that. In this example, that means targeting iOS 10.

But what about all the users who update to iOS 11? There are new features they'll miss out on if you limit yourself to the iOS 10 SDK.

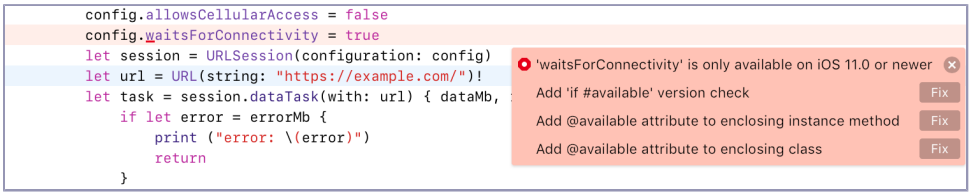
For example, consider the following code, which uses a feature that's new in iOS 11:

```
Line 1  let config = URLSessionConfiguration.default
    2  config.allowsCellularAccess = false
    3  config.waitsForConnectivity = true
    4  let session = URLSession(configuration: config)
```

The new API is on line 3. waitsForConnectivity is a new property that allows a URLSession to wait for a suitable connection to become available, rather than failing immediately. In this case, line 2 says we're not willing to use the cellular modem; so if that's the only connection, this code will just wait until you get on Wi-Fi. (By the way, if the session does need to wait, it calls the URLSessionTaskDelegate method urlSession(_:taskIsWaitingForConnectivity:), so you could use that callback to tell the user that they need to get on Wi-Fi, or let them cancel the request now.)

This code is fine if you only want to support iOS 11. But if you want to support iOS 10? Problems appear, because waitsForConnectivity doesn't exist in iOS 10. That means you can't just compile it for iOS 10, since the compiler would see waitsForConnectivity as an undefined symbol. Also, this strategy wouldn't work anyway, because Xcode only ever has two Base SDK settings: a specific version number like iOS 11, and "Latest iOS". You almost always want the latter, because it will keep working when you open your project on next year's version of Xcode.

Instead, you can go to the Info tab and change the Deployment Version to iOS 10. Now the code will produce a build error: 'waitsForConnectivity' is only available on iOS 11.0 or newer. Notice in the figure how the error icon is a circle? That means there's an instant-fix available:



In Swift, you can use the availability syntax to handle the possibility of running this code on a version of iOS that doesn't have waitsForConnectivity. The instant-fix offers us three options: you can put an #if available test inside the method itself to avoid calling just a short section of code that uses an undefined symbol; you can put an @available on the method to hide the whole method from iOS 10; or you can use @available on the entire class.

You do need to think carefully about your paths through the code and how you want to handle each OS, but this case is pretty simple: the app can just let the connection fail on iOS 10, which is what has always happened in this scenario prior to iOS 11. So, accept the #if defined fix and rewrite the code like this:

```
let config = URLSessionConfiguration.default
config.allowsCellularAccess = false
if #available(iOS 11.0, *) {
    config.waitsForConnectivity = true
} else {
    // let connection fail if cellular-only
}
let session = URLSession(configuration: config)
```

Now you have code that is built with the iOS 11 SDK, but can run on a Deployment Target of iOS 10. And, as a bonus, it will use this iOS 11–only feature for users who are running the latest version of the OS.

Using Target Settings

Along with the Project's overall settings, there are settings for each target. A target, you'll recall, is something you *do* with the files in a project, like producing an app or building and running tests.

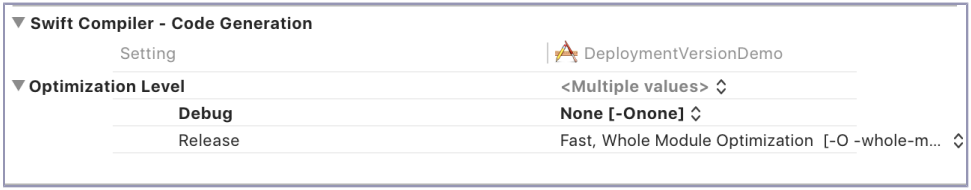
The target settings are more involved than the top-level project. While the project had only two tabs, an iOS app target will have seven: General, Capabilities, Resource Tags, Info, Build Settings, Build Phases, and Build Rules. Some of these are offered as conveniences: the Info tab puts some of the most important settings front-and-center, duplicating and finessing material from other tabs. For example, in an iOS project, the pop-up for Devices (with the options "iPhone", "iPad", and "Universal") actually duplicates a build setting called "Targeted Device Family", which has the possible values `1`, `2`, or `1, 2`.

Similarly, the Capabilities tab offers simplified configuration of features like Siri, HomeKit, and Apple Pay. Turning on one of these features will edit settings files, create and/or edit entitlements files, link frameworks, and even edit your app's metadata on Apple's backend (something we'll look at much later, in Chapter 9, *Security*, on page 171).

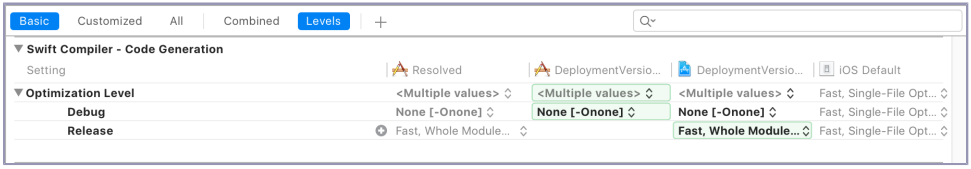
As for the rest of the tabs: Resource Tags are used for on-demand resources, which are of particular interest to tvOS developers (which we'll visit in Chapter 11, *Platform Specifics*, on page 207). The Info tab contains a duplicate of the Info.plist file that provides the metadata for the target. It also offers the ability to enter *Uniform Type Identifiers* (UTIs) for file types that your app declares

or opens from others, as well as URL protocol strings that can be used to open your app. Finally, the various Build tabs require a chapter of their own, Chapter 5, *Building Projects*, on page 89, which we'll visit after we've created some more code and storyboards.

But what can you *do* with this stuff? Let's look at a simple example. Look around the Build Settings tab and you'll notice that many values can be expanded with a disclosure triangle, to show different settings for `Debug` and `Release`. These are the two *configurations* every project starts with. If you look at "Optimization Level" under "Swift Compiler - Code Optimization", you'll notice that Debug gets no optimization, while Release is set to "Fast, Whole Module Optimization". These values are just strings, so if you somehow have some other -O option you want to send to the compiler, you can enter it here.



Notice in the figure that the Debug setting is shown in bold. This indicates that this setting has been customized by the target. You can see that by going to the header of the table. There are two sets of labels here that act as segmented controls (or radio buttons, if you remember those). The first three filter what to show: basic settings, basic plus any settings you've customized, or all settings. The next two buttons are for display mode: combined or levels. The default is combined, but click "Levels" to see a UI like the following figure.

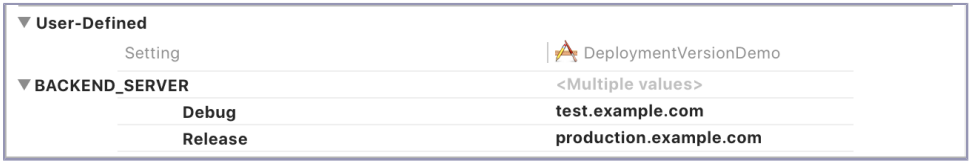


The Levels view reveals the priority by which settings are resolved. Reading right-to-left (what is this, a Japanese manga?), it shows the platform default, the project setting, the target setting, and the final resolved value (which is what the "Combined" shows). As you can see, the target defines a different compiler optimization—none—for the Debug configuration.

You can also add your own settings, and that can be really handy. Consider this common scenario: you have separate backends for testing and production. Because, you know, having testers banging on production is a *really bad idea*.

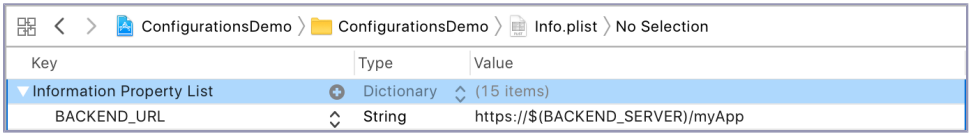
So, you can use the difference between the Debug and Release configurations to set different servers.

You do this by selecting the Build Settings tab and using the menu item Editor > Add Build Setting > Add User-Defined Setting. At the very bottom of the table, in the section "User-Defined Settings", this creates an entry called NEW_SETTING. You can type over the name, and either type a value to use everywhere, or expand the disclosure triangle and enter different values for Debug and Release. In the following figure, the setting name is changed to BACKEND_SERVER, and uses different host names for Debug and Release.



So now you have two different host names, but how do you get these into your code? After all, these are just build settings, so they'll be used at compile time and then forgotten. You need to deliver them to your code somehow. The best way to do that is with the Info.plist file. By looking at that file—either in the Info tab, or by selecting Info.plist from the File Navigator—you can see that many of the values in this file are captured from existing build properties using the syntax $(PROPERTY_NAME).

You can add to this list by selecting the top row, "Information Property List" and clicking the circular plus (+) button that appears on its right side. This adds a new property, selects its name for editing, and puts an auto-complete box under it with the symbolic names of common iOS properties. You can just type in BACKEND_URL for the name, leave the type as String, and for the value enter https://$(BACKEND_SERVER)/myApp. This will expand the build-setting host name, and give you a complete URL.



Now that BACKEND_URL is in the Info.plist, you can access it at runtime by using the Bundle API, specifically the object(forInfoDictionaryKey:) method, like this:

```
guard let urlString = Bundle.main.object(
    forInfoDictionaryKey: "BACKEND_URL") else {
    print ("couldn't get BACKEND_URL")
    return
}
print ("Backend url is: \(urlString)")
```

If you do a local build-and-run, the default scheme uses the Default configuration, so the output in the console is:

```
Backend url is: https://beta.example.com/myApp
```

Finding stuff in the Info.plist is just one of several nifty uses of the Bundle API. We'll see more a little bit later.

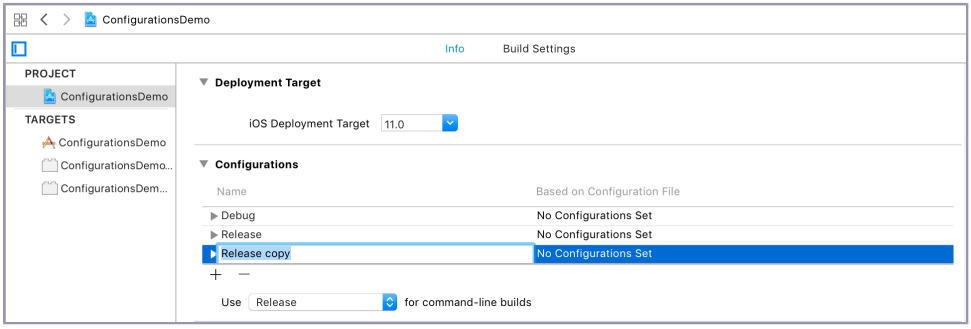
Using Configurations

So you're probably feeling good about the clever use of build settings to keep test runs out of the production backend… until someone decides there needs to also be a proper beta test, with outside testers.

Well, now what do you do? The testers should get a release build, so that it will be as much like the build that ships to the App Store as possible. That is to say, the compiler should use all the performance and size optimizations that a release build would have. However, you want testers to use the test backend, or maybe some beta-only backend, and not the production server. Suddenly, having a distinction between just Debug and Release doesn't cut it anymore.

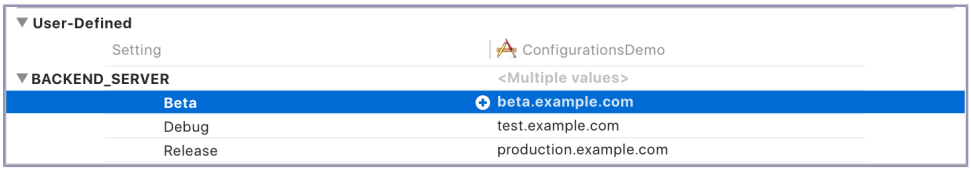
What you need now is a *third* configuration, separate from both Debug and Release. That way, you can build it like Release, but give it a non-production value for your BACKEND_SERVER setting.

From the projects and targets list, select the project, and use the menu command Editor > Add Configuration > Duplicate "Release" Configuration. This creates a new configuration called Release Copy, and selects it for you, as shown in the figure. Next, change the name to Beta.



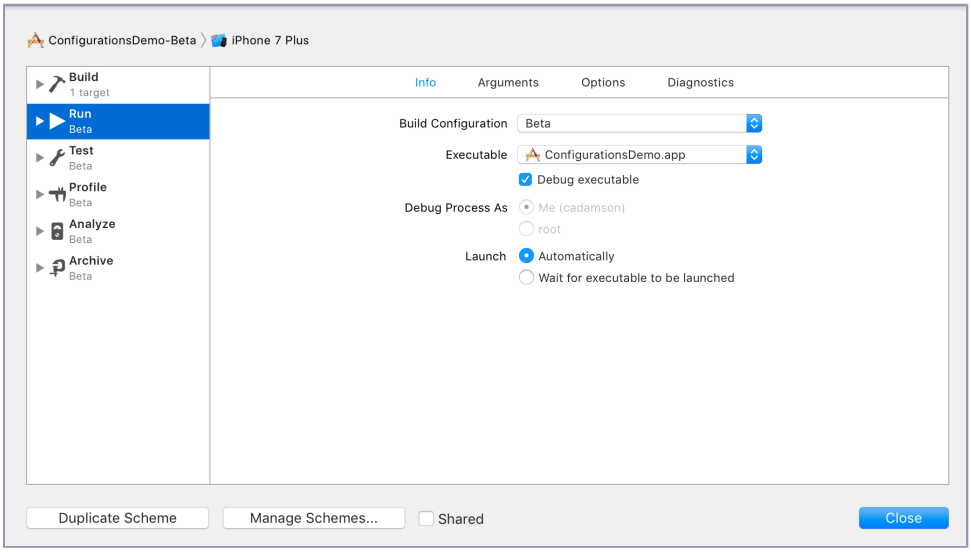
Now if you select the app target, select the Build Settings tab, and scroll down to the Swift Compiler settings, you'll notice that the Beta configuration has the same optimizations as Release does. Then scroll further down to the user-defined setting, BACKEND_SERVER, and notice that the disclosure triangle shows three values: test.example.com for Debug, and production.example.com for Release and

Beta. Now, you can give your new configuration its own value: beta.example.com, just like in the following figure:



So, this is great, but it does make you wonder: how do you actually *use* this new configuration? For that, you need to go up to the scheme selector at the upper left of the Xcode window. A scheme represents how you use the available configurations. For example, the default scheme for the app uses the Debug scheme for building and running locally, but the Release scheme for archiving, since archiving is the first step of distributing to the App Store.

You'll need a scheme that knows to use the Beta configuration, so you might as well just create a new scheme for Beta. From the scheme selector, choose "Manage Schemes...", which slides out a sheet showing a list of current schemes, which is just the default scheme. If you click the plus (+) button at the bottom of this sheet, you can create a new scheme. It asks which current scheme to base it on, and lets you enter a new name, like MyAppName-Beta. This adds the new scheme to the list, and you can click the Edit button to start editing the scheme, as seen in the following figure:



The left side of the scheme editor shows six commands: Build, Run, Test, Profile, Analyze, and Archive. All but the first lets you select a configuration to work with, so you can just change them from Release to Beta. This means

that running locally will use the Beta configuration, so if you build and run this scheme, the output is now:

```
Backend url is: https://test.example.com/myApp
```

Managing Your App's Version Numbers

The versioning of your app is another important setting managed by the project, but it's not as simple as just slapping a 1.0 in a text field once you're ready to ship.

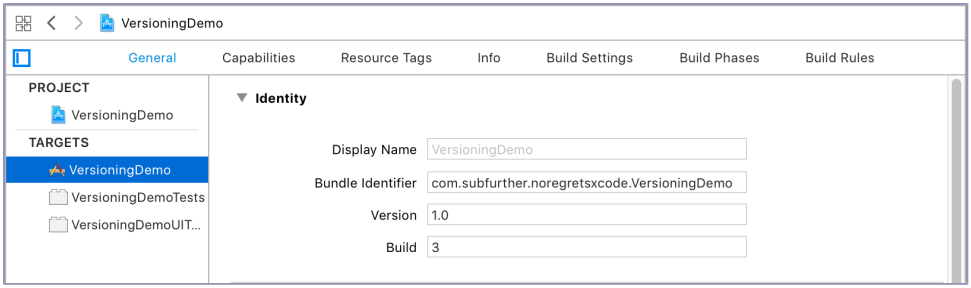
For starters, apps within the Apple ecosystem have two different concepts of a "version", and they sometimes go by different names. Let's look at Xcode's own info box to make sense of this:

The first version number shown, 9.0, is the *marketing version*. This is the version number that users will see on the App Store or in iTunes, and in the About Box of a Mac app.

The number in parentheses is the *build number*. This is a number that's meaningful to the developer, as it represents one specific build of the app. An app will go through many more build numbers than marketing versions. You've seen this if you've ever followed along with a beta release of an app from Apple: the build number changes with every release, but the marketing version remains the same. You'll want to do the same with your betas: version 1.0 / build 1, version 1.0 / build 2, and so on.

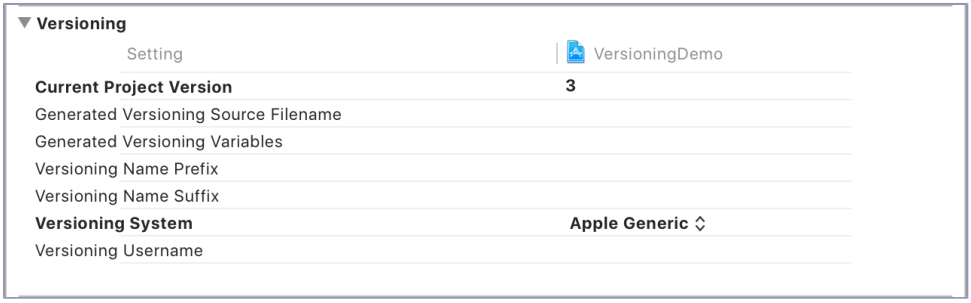
The simple way to manage version numbers is in the app target's General tab (shown in the figure on page 13) where there are text fields for the version (that is, the marketing version) and the build number. You can just update these prior to any build you're going to send to testers or submit to the App Store.

The version is actually stored in the Info.plist, and this is where it gets a little confusing. If you look at either the target's Info tab or navigate to the Info.plist tab, you'll see that the marketing version is called "Bundle versions string, short", and the build number is "Bundle version". But they're really the same things. If you edit one of those values here, the change will show up back in the General tab.



Each target in your project also has entries for these versions in its Info.plist, although for a simple app project, that just means your test targets (if you have them). It gets more interesting once your project has targets like frameworks and app extensions, which we'll be talking about in *Creating App Extensions and Frameworks*, on page 18.

Apple also has a command-line tool for managing your versions. It's called *agvtool*, for "Apple Generic Versioning Tool", and it's installed as part of Xcode's command-line tools. If you want to use it, you need to go to the Build Settings for your project (not the app target), find the "Versioning" section, and set "Versioning System" to "Apple Generic", as shown in the following screenshot:



Once you do this, you can use the agvtool to move the versions of all the targets, and the project itself, in lock-step. In Terminal, cd (change directory) to the project's folder, and then set the marketing version with:

```
agvtool new-marketing-version 0.3
```

You can also bump the build version—that is to say, increment it by one—of all targets with this command:

```
agvtool next-version -all
```

There is more information about agvtool in its Unix man page (type man agvtool on the command line) and online in the Apple Q&A "QA1827 Automating Version and Build Numbers Using agvtool". It is most useful if you set up a

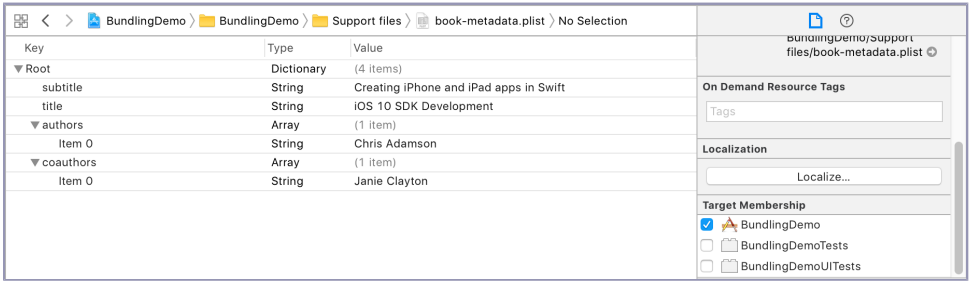
continuous-integration server to do your builds, where you can configure agvtool to bump the build number automatically whenever you send builds to testers or the App Store, something we'll talk about in Chapter 5, *Building Projects*, on page 89.

Adding Images and Other Files to the App

In the previous section, you saw how the Bundle class lets you read values from the app's Info.plist. Actually, it offers much more than that. You can use the Xcode project to include any kind of file inside your app bundle, then find and read it at runtime.

Finding Arbitrary Files in the App Bundle

For example, let's say you add a new file to a project, using the Property List template. In the File Inspector (⌥⌘) on the right, under "Target Membership", the check box for the app target will be checked, meaning this file will automatically be copied into the app at runtime. The following figure shows the plist's target membership:



The way you find it is with the Bundle class. The app itself is the "main" bundle, represented by Bundle's class property main. You can ask the bundle to search for a file by name and extension and either return a String (with path(forResorce:ofType:) and its variants) path or a URL (with url(forResource:withExtension:), etc.).

In iOS 11, the new PropertyListDecoder class makes bundling a .plist particularly appealing. Given the structure of the property list in the above screenshot—Strings for title and subtitle, arrays of Strings for authors and coauthors—you can represent the contents as a Swift struct like this:

projects/BundlingDemo/BundlingDemo/ViewController.swift

```
struct BookMetadata: Decodable {
    let title: String
    let subtitle: String
    let authors: [String]
    let coauthors: [String]
}
```

Since the struct is marked as implementing the Decodable protocol, you can use it with the PropertyListDecoder. The trick is, how do you get the property list as Data to feed to the decoder? The answer, of course, is to have the main Bundle find the file and load its contents as Data. Here's the recipe for that:

projects/BundlingDemo/BundlingDemo/ViewController.swift

```
do {
    if let metadataURL = Bundle.main.url(forResource: "book-metadata",
                                         withExtension: "plist") {
        let metadataData = try Data(contentsOf: metadataURL)
        let decoder = PropertyListDecoder()
        let metadata = try decoder.decode(BookMetadata.self,
                                          from: metadataData)
        titleLabel.text = metadata.title
    }
} catch {
    print ("couldn't get metadata: \(error)")
}
```

So, the technique here is that you just add a non-source file to the project, ensure that its Target Membership check box is set, and then you can find it at runtime, using the methods in Bundle.

Using Images in Apps

Simple enough, right? Now, what if you want to do the same thing with images? Obviously, you could use the same technique, finding image files in the bundle and loading their data into UIImage instances, like this:

projects/BundlingDemo/BundlingDemo/ViewController.swift

```
if let imageURL = Bundle.main.url(forResource: "original-cover",
                                  withExtension: ".jpg"),
    let imageData = try? Data(contentsOf: imageURL) {
    let image = UIImage(data: imageData)
    imageView.image = image
} else {
    print ("didn't find")
}
```

This will work, but it's a *really* bad idea. Don't do this! The most obvious problem here—there are several, but this is the worst—is that this doesn't account for the different resolutions of iOS devices. No one resolution is ideal for all devices; a given image either has less resolution than a top-of-the-line device like an iPhone X wants, or more resolution than the lesser devices can use. Maybe that can't be helped for images you download at runtime, but for images supplied with the app, there's no excuse not to tailor each exactly to the target devices.

The right way to bundle images with your app is to use an *asset catalog*. An asset catalog can hold the same image at multiple resolutions. By default, an app project will have an asset catalog named *Assets.xcassets*, which contains an empty entry for the app's icon. You can add more assets of various types (3D textures, sticker packs, Apple TV image stacks, etc.). For an image asset, you're expected to provide the image at "normal" resolution (one on-screen pixel per virtual point), and then double and triple sizes (usually denoted with a `@2x` and `@3x` in their filenames). In the following screenshot, we've added appropriately scaled images for the cover of the *iOS 10 SDK Development* book, giving this asset the name `adios4-cover`:

The advantage of this approach is that thanks to *app slicing*, only the resources relevant to a given device will be sent to that device when it downloads your app from the App Store. That means that the "Plus" phones will only get the triple-size image, while smaller phones will only get the double-size. Either way, neither device wastes bandwidth or storage on resources it doesn't need.

As a bonus to you, the developer, loading the image gets easier, because you can use the name from the asset catalog as the argument to the `UIImage(named:)` initializer:

projects/BundlingDemo/BundlingDemo/ViewController.swift

```
if let image = UIImage(named: "adios4-cover") {
    imageView.image = image
}
```

In fact, even that isn't the best way to load an image. After all, you might misspell the name (which is why the initializer is failable, and you have to use an `if let`). Now, this will really blow your mind, but if you just start typing the name of the image from the asset catalog, it will be offered as an autocomplete as shown in the figure on page 17.

```
imageView.image = adios4
                  adios4-cover
```

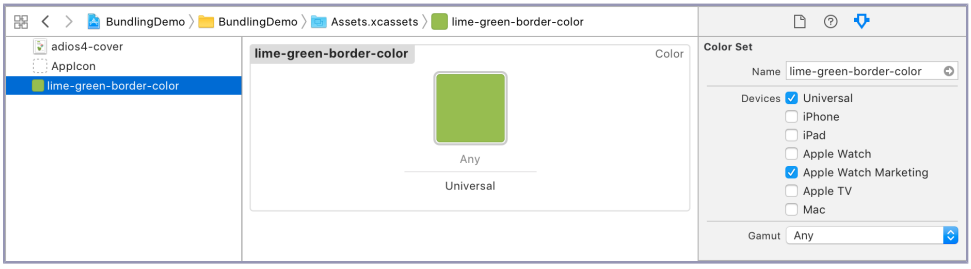
Accept the autocomplete with the return key, and an *image literal* will be added to your source. You're literally saying "set the image property of imageView to this specific image from the asset catalog. And thus, a tiny thumbnail of the image will be shown, inline, in your source code:

```
imageView.image =
```

This seems super crazy at first, to have an image sitting in the middle of your code, but it's actually great. Because Xcode knows what's in your asset library, it can guarantee that an image can be loaded at runtime, so it can assign it directly like this, eliminating the if let dance required for UIImage(named:).

Using Colors in Apps

In a similar way, you can use the asset catalog to define colors. For example, you can choose a color with the standard macOS color pickers and define it as lime-green-border-color in the asset catalog, as shown in the following figure:



Then you can load it by name with the initializer UIColor(named:), like this:

projects/BundlingDemo/BundlingDemo/ViewController.swift

```
if let borderColor = UIColor(named: "lime-green-border-color") {
    imageView.layer.borderColor = borderColor.cgColor
    imageView.layer.borderWidth = 10.0
    imageView.layer.cornerRadius = 10.0
}
```

However, color literals in code work differently than images. Xcode doesn't let you just type lime-green-border-color and autocomplete it to your color. You can autocomplete "Color literal" to get an editable color inline with your code, and then set it to any color you like, but it has no relationship to the asset catalog.

That's kind of a bummer, because between images, colors, and storyboards (which we'll visit in the next chapter), it's possible (at least in theory) to send the Xcode project to a non-developer designer, and have them do most or all of their work directly in Xcode, rather than sending Photoshop files to developers to implement. But for such a workflow to be dependable, you really want the designer to be working with named colors in the asset catalog, not individual color splotches in source files.

Still, compared to the old ways of doing things—writing "theme" classes with dozens or hundreds of class methods to create and return each color or image used in the app—asset catalogs are a big win.

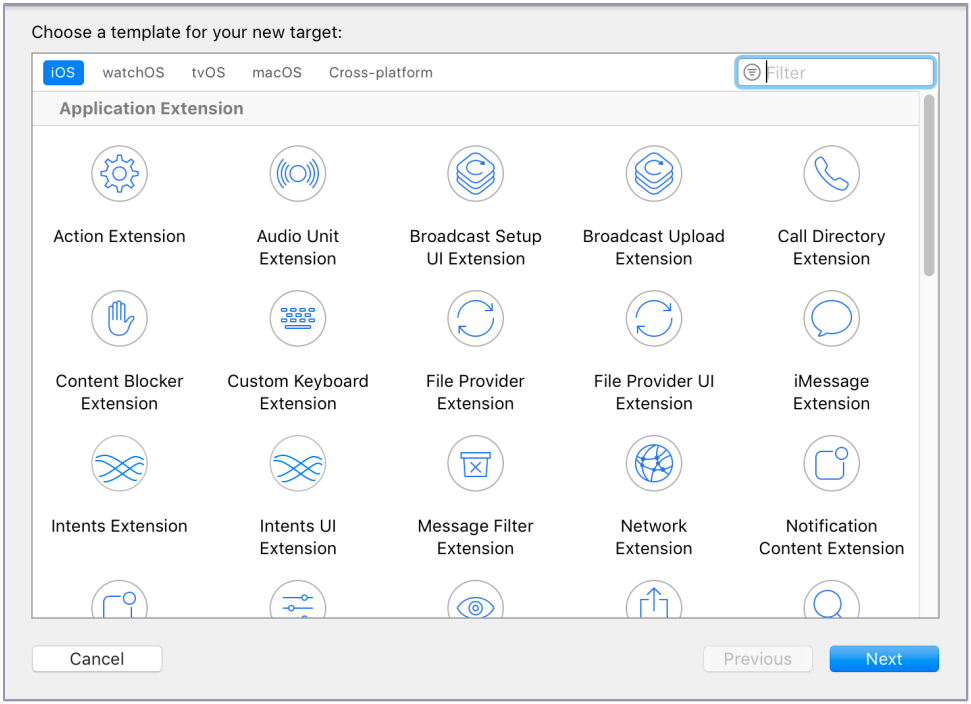
Creating App Extensions and Frameworks

So far, we've been talking about how projects manage settings and contents with the simple case of a single app target and possibly targets for unit tests and UI tests. As it turns out, projects can get a lot more complex than this.

More and more, apps are choosing to create *app extensions*, separate executables that have a relationship back to their host app. The first few extensions were for things like custom keyboards and notification center widgets, but with Apple adding more extension types every year, there are now more than 20 extension types. Some of these might not even initially appear to be app extensions, like providing custom actions for handling push notifications, intents to handle Siri voice commands, or audio units to process streams of audio.

So, don't be surprised when you find you need to add an app extension to your project. To do so, you select the project in the File Navigator, which brings up the project and target settings in the editor area. If you're showing the projects and targets list on the left side of this area, press the plus (+) button at the bottom of the projects/targets list. If you're not showing the projects/targets list, you'll have a pop-up menu of projects and targets, with "Add target…" as its last menu item. Either way, this brings up all the templates for new targets to add to your project, starting with app extensions. The figure on page 19 shows this sheet.

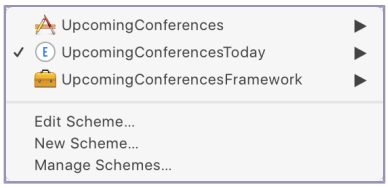
Once you add an app extension target, Xcode adds it as a new choice in the scheme selector, and slides out a sheet asking if you want to make it the current scheme. From this point on, you'll want to start paying attention to the currently selected scheme—are you building the extension, or its host app? Xcode will manage the dependencies too, so it will build the app extension before it builds the app itself.



Also keep in mind that some extensions, by their nature, are run by a host app. This might not be your app, especially if all your functionality is in the extension, and your main app doesn't really do anything. For example, a custom keyboard extension probably makes more sense to run with the Contacts app or some other text-input app, and an audio unit might want to run with Garage Band. With these kinds of extensions, when you use the Run command (⌘R), Xcode will slide in a sheet asking which host app you want to launch your app with.

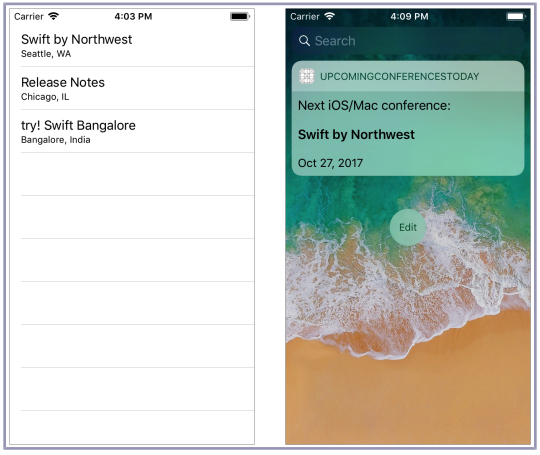
Another thing to keep in mind with App Extensions is code reuse; if your app extension and main app use some of the same logic, you obviously don't want to have separate source files in each group to do the same thing. One way—the bad way—to deal with this is to use the Target Membership check boxes in the File Inspector to compile a given source file for each target that needs it. This is bad, because you do more work redundantly rebuilding the same code, and you leave multiple copies of it on the user's device.

The right way to deal with this is to use an *embedded framework*. Put all the data structures and logic that the extension and app both need in one place, and then have both of them make use of it.



Creating a framework is similar to creating an app extension: click the "Add target..." button, then scroll down to find the "Cocoa Touch Framework". Adding this will create a new target (or two, if you include a test target for the framework). It will also add another scheme (as shown in the figure)—basically, any target that can be built will create a new scheme.

As an example, in the download code, you'll find an app that shows a table of upcoming iOS developer conferences, and a Today extension that shows the first of them in the notifications center. These two executables are shown in the following figure:



To do this, you can create a `struct` in the framework, to represent an `UpcomingConference`. Notice how the structure and all its members are marked `public`. You need to do this because the framework is a different module than the app or the extension, and Swift's default access level is to make symbols only visible within the same module:

projects/UpcomingConferences/UpcomingConferencesFramework/UpcomingConference.swift

```
public struct UpcomingConference {
    public let name: String
    public let location: String
    public let startDate: Date
    public let url: URL
}
```

We'll also extend the structure with a `public static func allConferences()` that returns an array of `UpcomingConference`, but its implementation just returns canned values and isn't worth taking up space here.

What's interesting is that it's quite easy to use this class from the extension and the app. All you need to do is add an `import UpcomingConferencesFramework` to the top of any file that uses the framework. At least it's easy in Swift; for Objective-C, the framework target has a `.h` header file that you need to edit and manually add `#import` commands for all the headers you want to expose.

But once you've imported your framework, you can use the framework's types and methods as you would if they were in the current target. For example, here's all you need to do to refresh the Today extension with the first entry in the conferences array (if there is one):

projects/UpcomingConferences/UpcomingConferencesToday/TodayViewController.swift

```
func widgetPerformUpdate(completionHandler: (
    @escaping (NCUpdateResult) -> Void)) {
    let conferences = UpcomingConference.allConferences()
    guard let nextConference = conferences.first else {
        completionHandler(NCUpdateResult.noData)
        return
    }

    conferenceLabel.text = nextConference.name
    dateLabel.text = formatter.string(from: nextConference.startDate)
    completionHandler(NCUpdateResult.newData)
}
```

Working with Workspaces

There's one level of complexity beyond the multi-target project: the *workspace*. A workspace is a container for many projects, which allows you to build one project and then use its contents in building another.

Despite the added size, the workspace window looks and works for the most part like a project window: you browse the projects and their files in the file navigator, and choose targets to build and run with the scheme selector.

There probably aren't that many developers who have deliberately created Xcode workspaces. What's much more common is the use of CocoaPods,[a] a dependency manager for publicly available Xcode projects, most of them open source. Many useful frameworks are exposed as Cocoapods, and millions of apps make use of them.

When you first integrate CocoaPods into your project, the command-line `pod` tool creates a workspace containing your project and all the "pods" it uses. From this point on, you *have* to use the workspace file—your project can no longer build by itself, because it won't be able to access the frameworks it now depends on.

a. https://cocoapods.org

The app extension's syntax is a little funky, but the basic idea is that the extension gets this widgetPerformUpdate(completionHandler:) callback to let it update the widget. It updates the UI and then calls the completion handler, passing in one of three values to indicate whether it updated the contents (.newData), didn't update (.noData), or encountered an error (.failed). The app extension template gave you a stub for this method; all you needed to do was fill it in. And the way we did that was with a trivial call to your framework: try to get the array of conferences, take the first one, and use its fields to populate the labels.

So, embedded frameworks are nothing to be afraid of—they're just another kind of target. Add them to your project, import as needed, and build away.

Wrap-Up

Who knew that just doing File > New > Project did so much for us? In this chapter, you looked at how projects are organized, and how they work. Projects are containers for the files that make up your app, plus all the information needed to build those files into useful products. We've looked at how to make sense of the hundreds of project settings, like how to build for the current version of the OS but still deploy to earlier versions. We've seen how to create settings that are specific to Debug or Release builds, and create configurations that let you create purpose-built products, like beta builds that target test servers. And you learned how to manage the version number and build number of your app that will be seen by the App Store.

Then you dug into what's actually *in* a build, by looking at how arbitrary files and images can be bundled with an app, and all the advantages you get by using asset catalogs for things like images. Finally, as your needs grow, you discovered how to create app extensions and frameworks, and how to work with the targets and schemes they create.

You played with code, property lists, and image files in this chapter, but there's one crucial kind of file that a project manages: the storyboard. This is where you build the user interface of your apps, and it's a topic so big, it'll take at least one chapter to master its strengths and secrets. That's where we're going next.

CHAPTER 2

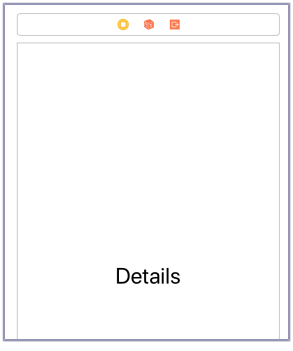
Storyboards: Appearance

One of the things that makes developing for Apple's platforms different from other styles of programming is where you begin. You don't start with data models, architecture, logic, or what have you; you start with the user interface. This has been a guiding principle of Mac development for decades, and it was inherited by iOS, tvOS, and watchOS. Think about what the user sees and does, *then* figure out how to make that happen. This is one of the secrets why people like apps on these platforms—thinking in terms of the user experience helps developers always look at the app through the user's eyes.

For developers, that means opening `Main.storyboard`, and working in the *Interface Builder*, Xcode's user interface editor. This is where you create the app's storyboards. In this chapter, you'll work with a single storyboard; in the next chapter, you'll learn how to add more. With Interface Builder, we can visually sketch out how the app is going to look, what the user will see, and even how they'll move around between scenes. With the UI prepared, you can move on to coding application logic in view controllers, data models, helper functions, and so on.

Working with Scenes

Each scene in a storyboard represents one view controller (either a `UIViewController`, or one of its subclasses, like `UINavigationController`, `UITabBarController`, etc.). The scene in the figure is a typical bare-bones example: the view has a single "Details" label as its subview.



Above the scene, there are three icons in a control strip, named the *Dock*. It shows the top-level objects in the scene: View Controller object, First Responder object, and an Exit object. It can have more, but at

Joe asks:
Can I Build My UI in Code?

Yes, you can. Also, please don't.

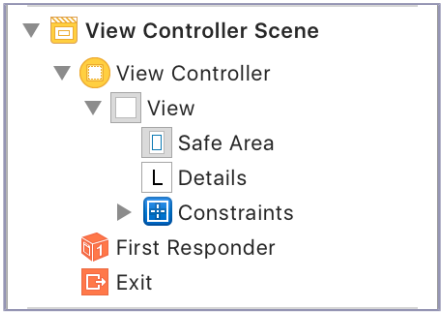
It's a somewhat common sight on Twitter to see developers trashing Interface Builder, asserting that building their UI programmatically is more performant, less buggy, or that they have some other compelling reason to do so. And it is indeed possible. After all, what IB actually does is create and configure instances of the various UI classes and stores them in XML, so they can be loaded at runtime. A UI built entirely in code simply moves this step from design-time to runtime. That's totally legal.

Thing is, IB does a *lot* for you: it lets you build your UIs visually (instead of choosing fonts and colors in code), it lets you preview your UIs on the entire range of devices, evaluates your auto layout constraints and finds problems at design-time rather than runtime, it makes it easier to internationalize by letting you totally rework your UI for specific localizations if needed, and so on.

There are times when building some part of your UI in code is necessary, since there are things that IB can't do well (or at all). But these are uncommon. And with the exception of some games—those which don't use UIKit and instead build an entire UI layer of their own in Open GL or Metal—it's unlikely that you'll be well-served by abandoning storyboards and IB entirely.

…and *always* be suspicious of self-appointed expert developers on Twitter and Stack Overflow.

a minimum, this is what most scenes will have. If you expand the Document Outline on the left side of Interface Builder—and I recommend always having it expanded—you'll see the scene presented as a hierarchy, as shown in the following figure:



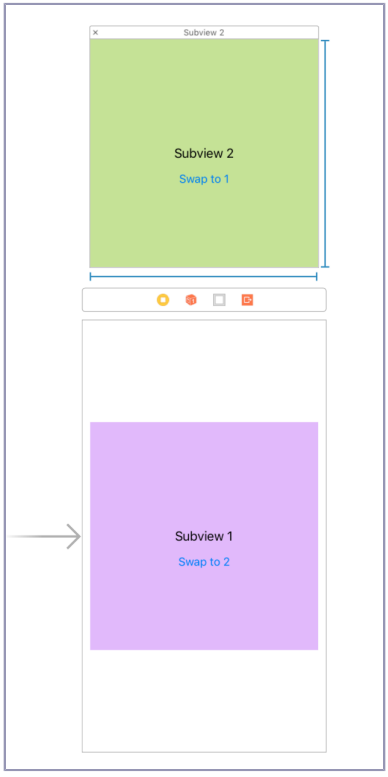
By opening the disclosure indicators in this tree view, you can see the main view (the top-level child of the view controller), the "safe area" within it, all its subviews (just the "Details" label), and the auto layout constraints that hold them all together.

After the View Controller, there are two curious top-level icons in both the Document Outline and the Dock above the scene. The *First Responder* is a *proxy object*, meaning that it's not an actual object in the scene, per se. Instead, it represents whatever object at runtime will be the first to receive messages from one of the scene's controls.

That begs the question of how messages from views are handled. Both UIKit (on iOS and tvOS) and AppKit (on macOS) have a concept of a *responder chain*, meaning that a message like a menu command will be sent to a subview or its controller. If handled, the message processing ends there. Otherwise, a parent view or view controller is asked to handle the message, and so on up through the containment hierarchy. First Responder is rarely useful on iOS, but on macOS, it's how menu commands are handled—the `paste:` message might go to a text view, then its superview, then its containing window, then its window controller, looking for any of these objects to implement the `copy:` method.

The other top-level icon is the "Exit", which will be covered later in *Segues*, on page 43.

Alternate Views



But that's not the whole story of the Document Outline. Imagine if you were to drag a second `UIView` into the scene… not as a subview of the main view, but as a sibling to it. The Dock would show the view as a top-level icon. If you then click on this icon, IB will create a new region for editing this view, separate from the rest of the scene, like in the figure.

This seems weird, but it makes sense: it's not part of the subviews of the main view that's connected to the view controller, so it doesn't appear in the scene itself.

Now, why would you do this? One use for this technique is that it allows you to swap subviews in and out at runtime. Consider the scene in the figure, where you have a subview centered within the main view and an alternate view with different contents. You can connect each of these views to the code as `IBOutlets` named `subview1` and `subview2`. Each has

a button that can be connected with an IBAction back to our code. Then swapping them in and out at runtime looks like this:

storyboards-appearance/SubviewSwapDemo/SubviewSwapDemo/ViewController.swift

```
@IBAction func swapTo1Tapped(_ sender: Any) {
    guard !view.subviews.contains(subview1) else { return }
    subview2.removeFromSuperview()
    view.addSubview(subview1)
    addSubview1CenteringConstraints()
}

@IBAction func swapTo2Tapped(_ sender: Any) {
    guard !view.subviews.contains(subview2) else { return }
    subview1.removeFromSuperview()
    view.addSubview(subview2)
    addSubview2CenteringConstraints()
}
```

Since we haven't covered auto layout yet, the listings in this code for addSubview1CenteringConstraints() and addSubview2CenteringConstraints() are omitted. But there's a downside of this trick: for the main view to center the subviews, you have to rebuild that stuff at runtime. Otherwise, the main view has no idea what to do with the subview you just added to it.

Arbitrary Objects

There's more stuff you can add to the scene. In fact, you can literally add *any* object type. In the Object Library, notice the cube-shaped icon. That's a generic NSObject, which you can drag into any scene. When you do, it will show in the Document Outline and the Dock as the cube icon, and you can create an IBOutlet to it and use it in your view controller.

OK, obvious question: *why would you do this?* Well, a generic NSObject doesn't do much good, but you can use the Identity Inspector to assign it to a different class. Like anything else in a storyboard, this will be a real, live instance of that class when the storyboard loads. So, if this object were, say, a table model, you can wire it up to the table and not have to involve the view controller at all. It will just work!

There are downsides to this approach, however, which explains why it's rarely seen in practice. The object has to be able to fully create itself from whatever data is available within the class, so that can hamper its usefulness. It has a further hindrance in Swift: like the subviews, it can't be created in the view controller's init()—since it is loaded from the filesystem later—so in code it

must be declared as an optional, or at least an implicitly unwrapped optional (as indicated by the ! character after its type).

Honestly, every time I've tried to use this approach, it has struck colleagues as too exotic, risky, or just plain wrong, and we've ended up moving the generic object's code into the view controller, or instances created by the view controller. But who knows, maybe you'll find a good use for this crazy technique!

Inspecting the Inspectors

While a fair amount of storyboard time is spent in the content area at the center of the window, the inspector area on the right is often where the action is. Whatever is selected in the content area—a view controller in the document outline, a view, a button, an auto layout constraint, etc.—its properties will be shown in the various inspectors on the right.

For iOS, there are six inspectors, indicated by a toolbar atop the utilities area, which itself can be shown or hidden by a button in the main toolbar. In order, the inspectors are: File, Quick Help, Identity, Attribute, Size, and Connection. macOS adds two additional inspectors: Bindings and View Effects. The inspectors can be shown (un-hiding the utilities area if necessary) with keyboard shortcuts ⌥⌘1 through ⌥⌘6 (or through ⌥⌘8 on macOS).

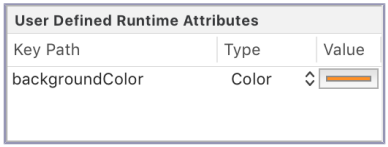
When you're editing code, the inspectors don't do much for you, but when building UIs, they're crucial. In fact, they have lots of neat abilities that many developers don't know exist.

Identity Inspector

The first two inspectors aren't used often. You first saw the File Inspector back in *Understanding Projects and Files*, on page 1, where you used it to view or edit the path to a file in the project. With storyboards, there are check boxes to opt-in to new Xcode features like the "Safe Area" that keeps the UI away from the iPhone X's sensor assembly (aka, "the notch"). If you're internationalizing a project, the localization section here lets you decide if this file will have language-specific variants. And then the second inspector, Quick Help, offers a top-level view of the documentation for the current selection in the storyboard.

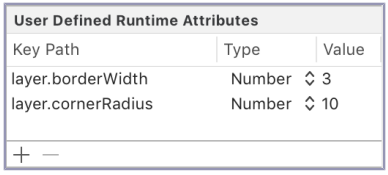
Where things get really interesting is in the third inspector, the Identity Inspector (⌥⌘3). The first field in this inspector is well-known to all iOS

developers: the "Class" field is where you can change the class of an object in the storyboard. The most common use is to change the generic UIViewController to a class of your own making; this is how you tie the storyboard to the specific behavior of the app code. If you were to create a subclass of common views like UIButton or UILabel, you can use this same technique to have the storyboard create an instance of your custom subclass.



A little further down in the Identity Inspector, there is an empty table called "User Defined Runtime Attributes", which is secretly one of the most awesome things in Xcode. This allows you to set any non-private property of the selected object. Here's a silly example: if you select a view, you can press the plus button (+) to create a new entry in the table, use the Key Path backgroundColor, change the type to "Color" and pick a color. Run the app, and *bam!*—the view now has the specified background color. What's happening is that backgroundColor is a property of UIView, and this lets you set that value.

OK, not that big of a deal, since views have color controls in the next inspector. The big win of the User Defined Runtime Attributes table is that it lets you edit any non-private property, even those that aren't otherwise editable here in Interface Builder. Better yet, the use of the term "Key Path" is your hint that this uses key-value coding, which means you can set properties of properties, by using a dot-separator syntax.



Here's a useful example of this: every UIView has a CALayer to show its contents. The CALayer class has some interesting properties for managing its border, such as borderWidth and cornerRadius. Those values don't have dedicated editors—in fact, Interface Builder has no representation of the layer at all. But you *can* set those properties of the layer with this table, using key paths on the view: layer.borderWidth, and layer.cornerRadius.

As it turns out, this is the common recipe for creating rounded borders on iOS, as seen in the figure. The one catch is that the editor can only handle certain types for the values of the properties set via the table. These include common types like booleans, numbers, and strings, as well as essential geometric types like point, size, and rectangle. You can also specify a color. But it only gets you so far. For example, CALayer also has a borderColor property, that

would tempt you to add layer.borderColor to your table. But that won't actually work, since the type of that property is actually a CGColor, and the table can only set a UIColor. So if you want a border color other than black, you'll have to set that property in code.

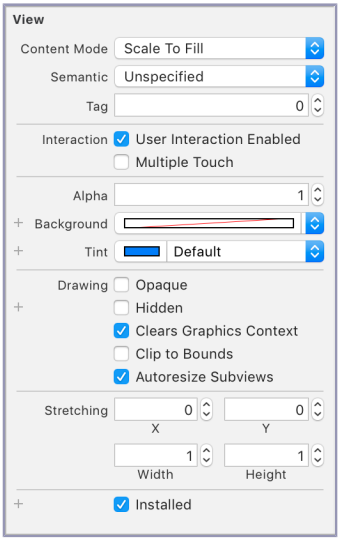
Still, this can be a terrifically useful technique, particularly with your own custom views, view controllers, and even the arbitrary objects mentioned earlier. Any non-private property can be set this way, so if it makes more sense to customize an object in the storyboard, expose a property as one of these editable types, and then configure it directly from the storyboard.

Attributes Inspector

If you've only been using Xcode and storyboards for a while, it's possible the only inspector you've ever used is the Attributes Inspector (⌥⌘4). This is where you go to change the text of a button or label, or customize a font. But there's a lot more in here than it might originally appear.

The Attributes Inspector is split into sections, with subclass-specific behavior at the top, then through its superclass' editors, down to the common-to-all-UIViews behavior in the bottom section. So, top to bottom, for buttons it goes button, control, view, and for text views, it goes text view, scroll view, view.

Common View Attributes

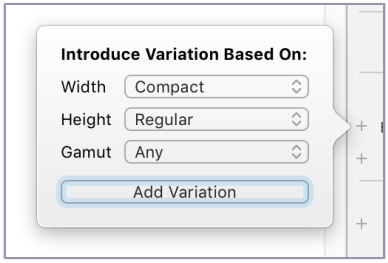


Head to the bottom and look at the attributes common to all views. Some of the settings here are going to depend on the type of the view. For example, most views have "Scale to Fill" for their "Content Mode", but labels use "Left". You rarely need to edit this, nor do you often need to mess with the "Stretching" geometry.

The third field, "Tag" can be useful for quick-and-dirty apps. Usually, to find a specific subview in code, you need to connect an IBOutlet, which in turn means custom subclasses, control-drags to make the connections, and those tried-and-true techniques. But another option is to just use this field to set the tag of the subview to some agreed-upon integer value, and then find it in code with the UIView viewWithTag(_:). Obviously, there are lots of ways for this to break if the tag value in the storyboard doesn't match what the code is expecting. It tends to be hacky, but it's there if you need it.

The "Interaction" section's check boxes will tend to be dictated by the type of view you're working with. "User Interaction Enabled" determines if touches are accepted, so buttons and sliders will enable it, and labels won't. But that doesn't mean you can't! You can turn any view into a de facto button by adding a tap gesture listener to it, but for the recognizer to get the taps, you'd have to enable user interaction here.

The "Alpha" setting lets you make your view (and all of its subviews) partially or completely translucent, which can be nice for certain effects. You can also animate the view's `alpha` in code for a fade-out effect.



The background color is self-explanatory, except to note that the red diagonal line represents a default color (which could be white, clear, or some other color entirely). Like many of the attributes, background color has a small plus (+) button to its left. This lets you create a second value for the attribute which only applies for certain combinations of size class and color gamut. This is used for creating device-specific tweaks to your views. A given background color might be fine on iPhone but overkill on the iPad. In that case, make the iPad color the default, then add a variation for the compact width size class—the variant color will only be used for portrait layout on iPhones (and a few edge cases that we'll discuss later).

The "Drawing" section has a few settings that aren't always intuitive by name, but are hugely important. "Opaque" means that a given view draws all pixels within its bounds rectangle and doesn't need to be composited with any views below it—supposedly, this can improve drawing performance if you're really sure you don't need any compositing. "Hidden" makes a view invisible and prevents it from receiving touch events, while still participating in auto layout, which means it keeps its shape and size. "Clips to Bounds" means that any subview that goes outside the bounds of this view will have its drawing cut off at this view's bounds.

Other View Subtype Attributes

There are a few other attributes specific to subclasses of `UIView` that are worth understanding:

- *Labels*—The "Lines" field defaults to a single line. If you have an arbitrarily long string as its text, set the value to `0` to create a variable-height label. This will make the label exactly as tall as it needs to be, provided you also change the "Line Break" setting to "Word Wrap" or "Character

Wrap". Your auto layout code (see next section) will also need to tolerate the variable height of the label.

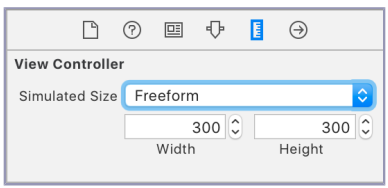
- *Buttons*—Buttons can have different titles or images for each of the four states: default, highlighted, selected, and disabled. For a play/pause style button, rather than changing its title in code, you can set the default title to "play", the highlighted title to "pause", and then just toggle the button's `isHighlighted` property to change the title.
- *Sliders*—If you leave "Continuous Events" selected, then any method you connect to the value-changed event will receive many callbacks as the user drags the playhead. If this check box is deselected, a single callback is delivered when the user lifts their finger to end the interaction.
- *Table Views*—Most of the time, you'll create table views that are connected to objects implementing the `UITableViewDataSource` and `UITableViewDelegate` protocols to provide table contents at runtime and handle user interactions with the table. But if a table will always have the exact same contents—such as for a settings UI—you can change the "Content" setting to "Static Cells", and then add and layout all the table cells directly in the storyboard.

Size Inspector

The important thing to remember about the Size Inspector is *never trust the Size Inspector.*

OK, kidding, but just barely. In an app that uses auto layout (see next section), the sizes and locations of views are a result of auto layout's calculations. As a result, while it looks like you can edit your views' origin points or sizes, it's all an illusion. These are *simulated* sizes and locations. Put in a different `x` for your button and it'll move in the storyboard, but you'll also see the various orange indicators telling you the view is now misplaced.

For the most part, the Size Inspector is useful for looking at what auto layout has done for you, and tweaking the layout constraints directly, which you'll do soon.



There is one place the Size Inspector is truly useful, albeit still in a "simulated" size way. If you use a pop over with iOS, the view controller you're popping up will still appear in Interface Builder as the full-size screen of the currently selected device. This is true even if you set the view controller's "Use Preferred Explicit Size" in the Attributes Inspector. It won't matter on iPhones, where pop overs fill the screen. But on

iPads, you don't want to be designing for a 300x300 pop over with a 768x1024 preview.

In this case, select the view controller (not the view!), and go to its Size Inspector. Change the "Simulated Size" from "Fixed" to "Simulated", and then enter the size and width in which you want to work. Keep in mind, this is just a preview (and it will be ignored on iPhones, where the pop over will fill the screen), but it will be a more accurate representation of how your layout will look on iPads.

Connections Inspector

The last inspector, at least on iOS, is the Connections Inspector (⌥⌘6). This is where you can inspect and set connections—IBActions and IBOutlets—from storyboard objects into your code.

Fixing Broken Connections

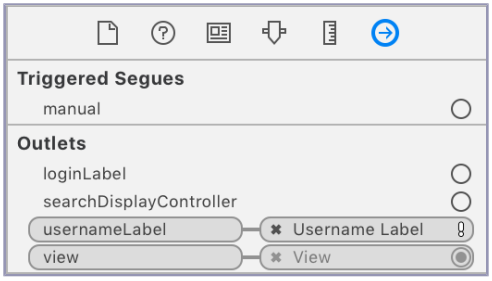
A good way to understand how and why this inspector works is to consider a common problem it solves. Imagine you have a label that you connected to your code as a property named usernameLabel. But then you decide that's an imprecise name, so in the code you rename it to loginLabel.

You run the app again… and it *crashes* immediately upon reaching this scene.

What happened? Start by looking at the console output:

```
*** Terminating app due to uncaught exception 'NSUnknownKeyException',
 reason:'[<BrokenConnectionSample.ViewController 0x7fa0f2402420>
 setValue:forUndefinedKey:]: this class is not key value coding-compliant
 for the key usernameLabel.'
```

Oops, somewhere in the code, the name usernameLabel is still hanging around. And that's because you made this change in the code, but not in the storyboard. Select the view controller in the code, bring up its Connections Inspector, and you'll see the offending party, as revealed by the following figure:



The exclamation point next to "Username Label" indicates this is a connection to an outlet property that doesn't exist in the code (because you deleted it).

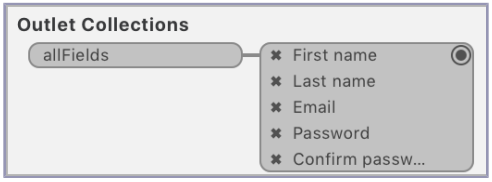
You can also see that the “Login Label” remains unconnected. The fix is to click the (x) button next to “Username Label” to delete the broken connection. And then you can drag from the circle “Login Label” over to the View Controller icon, where the drop target will offer the correctly named loginLabel outlet.

Outlet Collections

Another useful but overlooked feature of the Connections Inspector is the idea of the *outlet collection*. The idea is that you can create a connection to multiple objects in a storyboard, which a normal outlet can’t do. An outlet collection is simple enough in code—it’s just an array of some UIKit class, marked with the @IBOutlet attribute, so Interface Builder knows to pick it up from the code:

storyboards-appearance/OutletCollectionSample/OutletCollectionSample/ViewController.swift
```
@IBOutlet var allFields: [UITextField]!
```

Once this is in the view controller, the property will appear in the “Outlet Collections” section of the Collections Inspector. From here, you can drag connections from the circle to the right of the property name, over to different objects in your storyboard, one after the other. In the following figure, the five text fields in the scene are wired-up:



With the outlet collection set up, you can now access the array of text fields and perform actions on all of them. A simple example is to clear the contents of every text field, and kick the user out of editing any they might be currently editing (by calling resignFirstResponder()):

storyboards-appearance/OutletCollectionSample/OutletCollectionSample/ViewController.swift
```
@IBAction func clearAllFields(_ sender: Any) {
    for field in allFields {
        field.text = nil
        field.resignFirstResponder()
    }
}
```

This is much nicer than having to put all of the fields in an array manually, which you’d have to do in viewDidLoad(), to ensure they had already been loaded. Also be aware that the order in which you make your connections to the outlet collection is preserved as the order of the array. This matters if you

want to use the collection for something like validation, where you want to fail on the first invalid field, and not just check them in a random order.

Making Sense of Auto Layout

Few features in Xcode are as controversial as *Auto Layout*. For every developer who swears by it, there seem to be at least two who swear *at* it.

Auto Layout is meant to solve a hard problem: where do you put your UI elements in a variable-size display? Of course, that means all the sizes of iPhones, including the iPhone X with its intrusive sensor housing (commonly known as "the notch"). But remember that Auto Layout is also used on the Mac, where the user can usually resize your window at any time.

If you want to master Auto Layout, start a macOS project, build a user interface in a window, and then grab the corner and start resizing it. Watch what happens, and if you see your UI components getting crushed or stretched, start thinking about how to deal with that gracefully. A morning spent building Mac UIs is like a week doing the same for iOS.

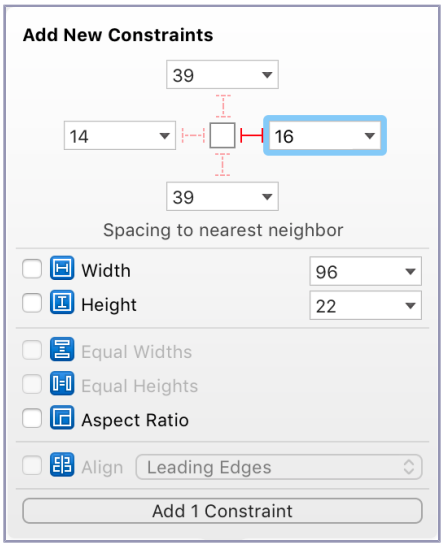
Auto Layout Essentials

Let's take a step back and look at layout from first principles. Any view or subview on the various Apple platforms has an origin point and a size. While you *can* build all the UI components in code and then assign their `frame` rectangles (which represents both the origin and size), you *shouldn't* do this. The reason being that iOS devices and Mac windows are many different sizes, and then doing the math to reposition and resize the views quickly gets horrible. Trust me, you don't want to spend a cold Tuesday figuring out logic like "subtract half the label's width from half the screen width, then add back a 6-point margin, then…"

This is the problem Auto Layout solves. You create *constraints* that describe what you want the layout engine to do: "make this button 60 points wide", "keep these two labels' left edges aligned", "keep this subview vertically centered in its parent". None of these explicitly sets the frame of any subview. Instead, Auto Layout takes all of the constraints and finds a solution that fits all of the requirements. It's a declarative style of programming, rather than an imperative one. It may help to think of Auto Layout as a rule engine: you feed in requirements, and it produces a layout that satisfies those requirements.

Another thing to keep in mind is just what a constraint is. The `NSLayoutConstraint` class has seven properties that usually matter: `firstItem`, `firstAttribute`, `secondItem`, `secondAttribute`, `relation`, `multiplier`, and `constant`.

Consider the constraint being created with the pin menu in the figure. It's meant to maintain a 16-point space between the selected view and the view to its right in the storyboard.



The constraint created by the pin menu in the figure is equivalent to the following code:

```
let constraint = NSLayoutConstraint(item: firstItem,
                                    attribute: .trailing,
                                    relatedBy: .equal,
                                    toItem: secondItem,
                                    attribute: .leading,
                                    multiplier: 1,
                                    constant: 16)
```

Now, that's just one constraint. The trick of Auto Layout is to get the right number of constraints—not so few that something important is unaccounted for, nor so many that there are irreconcilable inconsistencies. Imagine the rule engine again, and assume you have created a single view pinned to the 320 x 568 point size of an iPhone SE, but with no positioning constraints. What happens when you run it on a 414x736 iPhone 8 Plus? There's more space than the view needs. Should it be centered, anchored to the top left, or stretched? There aren't enough rules for Auto Layout to know what to do.

Now, instead, pretend that you fix the size, but also pin the position to a zero-point distance from all the borders. Again, it breaks on the 8 Plus—it can't simultaneously have 320-point width, while also hugging the sides of the screen, which would make it 414 points wide.

Strategizing for Auto Layout

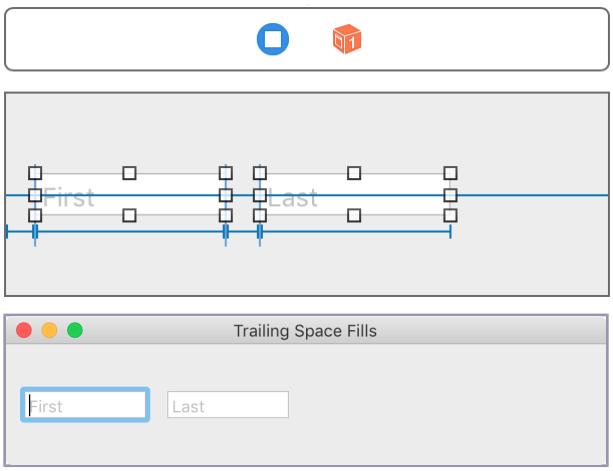
As a general strategy, you want something in your UI to be able to expand or contract. That can be whitespace, or inherently stretchable components, like text areas and tables. And then you use the constraints to imply where the stretching happens.

Some views have an *inherent size*. For example, UISwitch and UIActivityIndicator objects have a consistent width and height. Others are consistent in one dimension; for example, every UISlider is the same height. A button has an inherent size based on its title (and the font used to display the title), but it's legal to add size constraints to make it larger. These elements with known sizes, or sizes that you set manually, give you a start on laying out the view as a whole.

Here's a simple example you can play with: how can you lay out two text fields, side-by-side, and keep them looking good? You'll do this in a Mac window, so you can resize the window to see how the layout performs.

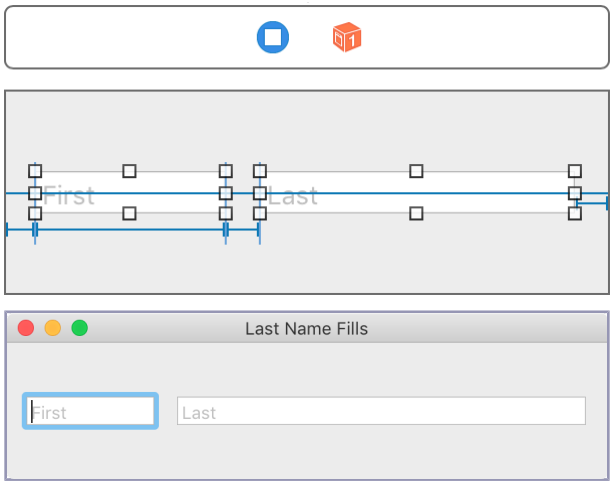
Text fields have an intrinsic height, so for this example, you can vertically center each of them in their superview.

So where does the extra space go if you resize the window? Let's try a simple option: give both text fields a fixed size (96 points in the sample code project), space the first field 16 points from the left side of the superview, and the second field's left edge 16 points from the first field's right edge. This is enough to provide the position and size of both fields, so it's legal. You have no constraints that are relative to the right side of the superview, and as a result, any extra space goes to the right of the second field.

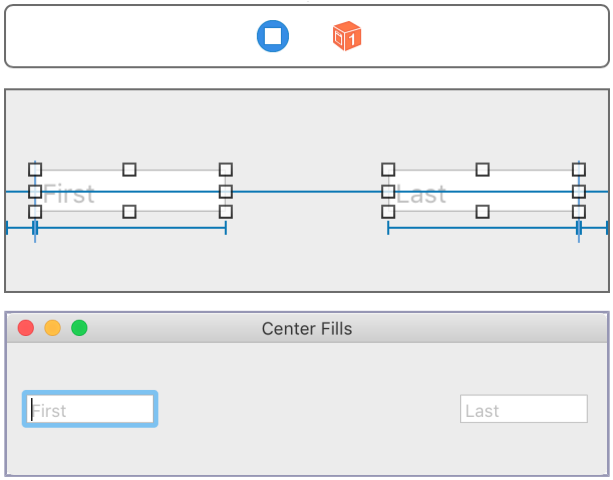


That's… not bad. It's pretty typical on the Mac. But on iOS, it will be less than ideal to have a bunch of unused space on the right for users with bigger screens. And the wider the window gets, the worse it is on a Mac.

OK, plan B: let's allow the second text field to take up the extra space. To do this, remove the 96-point fixed width on the second text field, and then add a 16-point constraint from its right edge to the right edge of the superview.

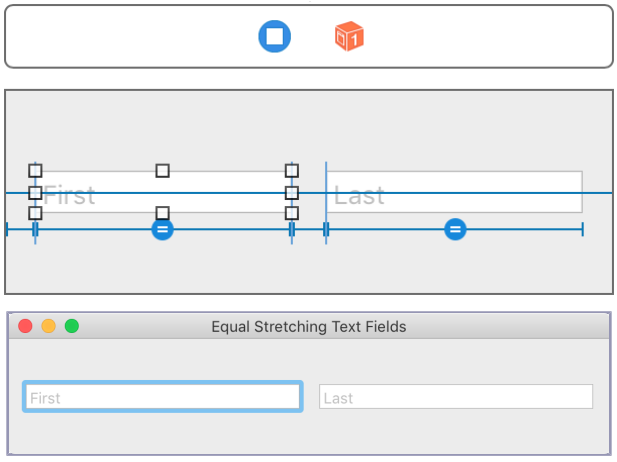


This looks a little silly on the Mac, depending how far you stretch the window, but it'll be a little more useful on iOS, which is about device size and not window resizing. Still, how about a Plan C, where you let the center take up the extra space? So then you have a width constraint for each text field, leading edge spacing for the first field, and trailing edge spacing for the second.



Ha ha ha ha…*no.* Let us never speak of this again.

What if you would like to make the text fields the same size, and have them equally share the space? It isn't obvious how you do this, but the trick is to use the pin menu again. Usually, you only use the pin menu with a single selection, to set explicit sizes or distances to other views. But if you select *both* text fields and show the pin menu, the "Equal Widths" and "Equal Heights" check boxes become enabled. And that's what's needed: a layout that pins leading distance to the first text fields, 16 points between the text fields, trailing distance after the second text field, and (most importantly) sets their widths equal.

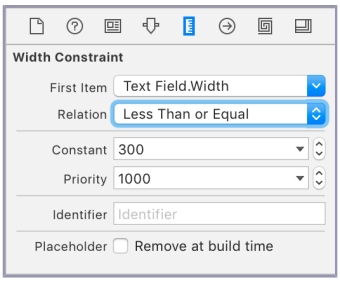


This is probably going to be your best bet on iOS, since it evens out the extra screen real estate between the multiple text fields, rather than having all the extra space go in one place.

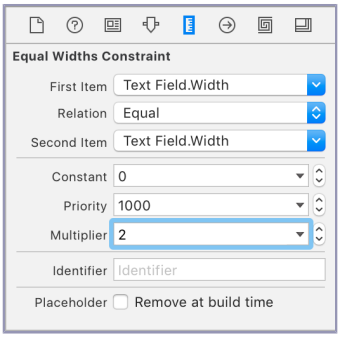
Tweaking Auto Layout Constraints

While the equal widths layout might be fine on iOS, it can still look ridiculous if you stretch the window really wide on the Mac. Does Auto Layout have an answer for that? Actually, yes.

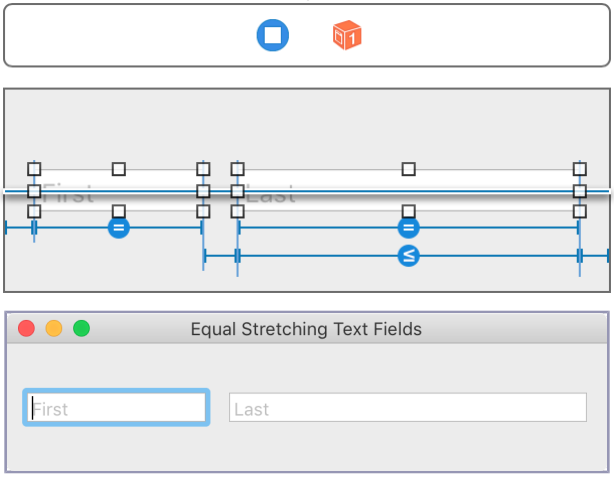
Let's say you never want the text fields to be wider than 300 points each. Fine. You can add a second width constraint to either of the text fields, set to 300 points. Now here's the interesting part: you can select that constraint —either as the horizontal blue bar under the text field, or as one of the tree nodes under "Constraints" in the Document Overview—and bring up the Size Inspector.



Yes, this is the same Size Inspector that was dismissed as useless earlier in this chapter because it only shows simulated sizes. Thing is, with constraints, it's much more useful. This lets you directly edit the various properties of the `NSLayoutConstraint` that were mentioned earlier. One of these is the "Relation". By changing that to "Less than or equal", you state that this text field can only get that wide, but no wider. Since you have the equal-sizes constraint, the other text field can also be 300 points wide but no wider. Add in the three 16-point size spacing constraints, and Auto Layout will figure out that the window's content area can never be wider than 648 points. And when you stretch the window—surprise!—this is actually enforced. You can't stretch the window wider than the constraints will allow.



There's another fairly useful change you can make in this inspector: the constraint multiplier doesn't *have* to always be `1`. What if you wanted to give proportionally more space to the last name field than the first name? You can do that by tweaking the equal-sizes constraint. You change the multiplier to either `2` or `0.5`, depending on which field was selected first when you created the constraint (if this is awkward, the inspector also has a "reverse first and second item" command). The available space is allocated proportionately, and now our layout is better looking and even more robust.



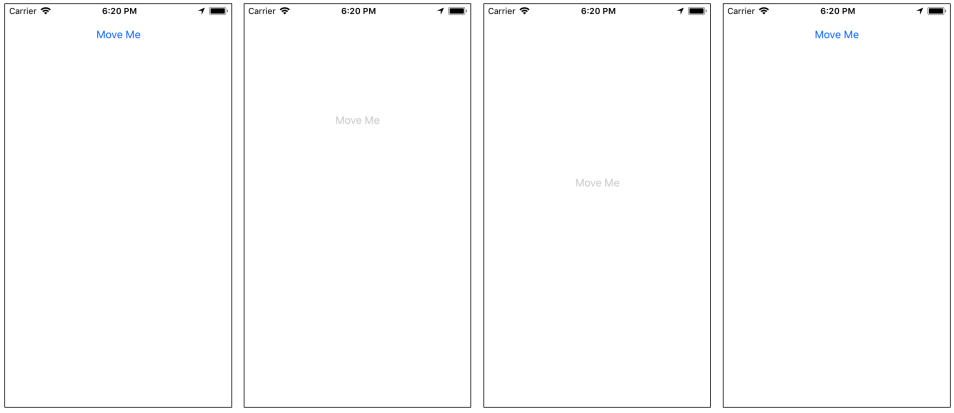
Animating Constraints

One thing about Auto Layout that might not be immediately obvious is how you're supposed to animate parts of your user interface, when Auto Layout is determined to keep the subviews in the rectangular spaces it has computed for them. Core Animation seems like it was built for updating the `frame` of a view in real time, but Auto Layout owns the frame. So what do you do?

As it turns out, one viable technique is to animate the layout constraints themselves. If you look at the `NSLayoutConstraint`, you'll see that all of the variables are immutably set at `init()`-time, except for one: `constant`. This is the property used for pinned widths and heights, as well as spacing between views. And it's the one property of a layout constraint that can be modified after the constraint has been added to a view.

To use this in code, you need one more trick: accessing a layout constraint at runtime. Obviously, that would work if you created constraints in code. But what if you used a storyboard? Fortunately, constraints in a storyboard can be `IBOutlets`, just like views, view controllers, and other objects. You can use the usual wiring-up techniques—like control-dragging from storyboard to source in the Assistant Editor—to create outlets to the constraint.

In the download code, there's a trivial demo of this: a button which, when tapped, moves partway down the screen and resets. The following figure shows a time-lapse of the running app:



The first thing you need to do is create an outlet to the top constraint, which sets the distance from the top of the button to the top of its superview. Typically, this line is automatically written by control-dragging in the Assistant Editor:

storyboards-appearance/AnimatedConstraintDemo/AnimatedConstraintDemo/ViewController.swift

```
@IBOutlet var buttonTopConstraint: NSLayoutConstraint!
```

You also need to wire the button itself to an action method. The action method can just use UIView.animate(withDuration:animations:completion:), like usual for a UIKit animation. The difference is, this time the value you're animating is the constant of the NSLayoutConstraint:

storyboards-appearance/AnimatedConstraintDemo/AnimatedConstraintDemo/ViewController.swift

```
@IBAction func moveButtonTapped(_ sender: Any) {
    moveButton.isEnabled = false
    UIView.animate(withDuration: 3.0,
                   animations: {
                    self.buttonTopConstraint.constant = 300
                    self.view.layoutIfNeeded()
    },
                   completion: { _ in
                    self.buttonTopConstraint.constant = 8
                    self.moveButton.isEnabled = true
    })
}
```

By setting the value of constant inside the animations closure, its value is periodically updated over the course of the animation. Then, in the completions closure, set it back to its original value.

Other Auto Layout Tips

Someone could probably write a book on Auto Layout (hey, maybe that's what we'll do next), so there's always more to discuss and we can't cover everything here. But there are a couple of everyday tips that should make your Auto Layout experience more pleasant:

- With complex layouts, particularly on iPads, it often makes sense to collect related items into views, and make your layout relative to those views. For example, an app's main screen might have a login area, an ad banner, another area for news and notifications, and so on. Rather than having every label, field, and button use constraints relative to the top-level view, it's more sensible to create subviews for each topic area, and give them fixed sizes and offsets. Then, you can put subviews into these views, and have each subview's constraints be relative to its respective container.
- With labels that stretch across the screen, use leading and trailing constraints to set equal distances from the sides of the top-level view. If you just use a horizontal centering constraint, there is nothing to keep a long string from stretching the label right off the sides of the screen. Also, constraining the label horizontally like this is the only way to get the numberOfLines==0 trick to work, since running out of horizontal room is the only way the label will know to word-wrap onto a new line.

- As you saw with the animation, it's OK to play with constraints at runtime. You can also add subviews at runtime. You can create a subview with its appropriate `init()` method, add it as a child of another view, create all needed `NSLayoutConstraints` in code, and then call `addConstraints()` on the superview. Note that if you do this, you want to be sure to set `translatesAutoresizingMaskIntoConstraints` on your new subview to `false`, or else it'll automatically create constraints that may conflict with yours and confuse Auto Layout.

Wrap-Up

In this chapter, you saw some of the main tasks in working with storyboards: creating scenes and populating them with user interface elements, using the inspectors to customize their appearance and behavior, and setting up Auto Layout constraints to size and position everything nicely. Hopefully, you're sold on the idea that you get a lot of functionality for free by building your user interface with Interface Builder, rather than try to do all of this in code. For many developers, storyboards are faster, more elegant, and produce better results.

But storyboards are too big of a topic for just one chapter. Next up, we're going to broaden the scope, by navigating between scenes, splitting storyboards into multiple files, and even creating custom types of objects in the storyboard, complete with visual editing.

CHAPTER 3

Storyboards: Behavior

The previous chapter focused on single scenes within storyboards: how to customize their appearance, connect them to code, and so on. But that doesn't get you far. Most iOS apps employ multiple scenes, transitioning from one to another as users log in to apps, perform searches, drill down to details… you know, *do stuff*.

In this chapter, you'll look at how to work with multiple scenes: how to transition between them, embed them into one another, and more. More broadly, you'll look at the techniques that affect storyboard *behavior*: not what scenes look like, but what they *do*. With these techniques, you'll be able to navigate between scenes in interesting ways, embed smaller scenes in containers, and even bring direct editing of custom views into Interface Builder. The payoff is getting more done in the storyboard itself, so the visuals can live there, while the code can remain focused on the app's functionality.

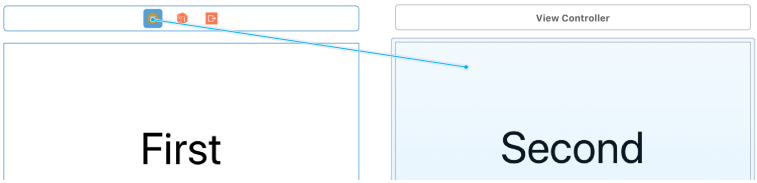
Segues

An empty single-view project starts with one scene. It has an arrow pointing into it, indicating that it's the initial scene. You can set this option with the "Is Initial View Controller" check box in the Attributes Inspector (⌥⌘4) for the scene's view controller.

You can add another scene to the storyboard by dropping a view controller on the storyboard, but it sits there, unconnected to the first scene. In fact, building a project in such a state will produce a warning that the scene is unreachable. To correct this problem, you need to either give it an identifier (so it can be created programmatically with UIStoryboard's instiantiateViewController(withIdentifier:)), or you need to connect it to the first scene.

The obvious way to connect it to the first scene is with a *segue*. Chances are you've used one of the simple techniques for creating segues. For example, if the first scene has a button, you can control-drag from the button to the new scene to create a segue that shows the new scene when the button is tapped. Similarly, if you have a table view or collection view, control-dragging from one of its cells to the new scene creates a segue that shows the new scene when the cell is tapped.

You can even drag from the first view controller's icon to the second to create a programmatic segue, as shown in the following figure. Assuming you then give the segue an identifier string in the Attributes Inspector, you can perform the segue in code by calling the performSegue(withIdentifier:sender:) method of UIViewController.



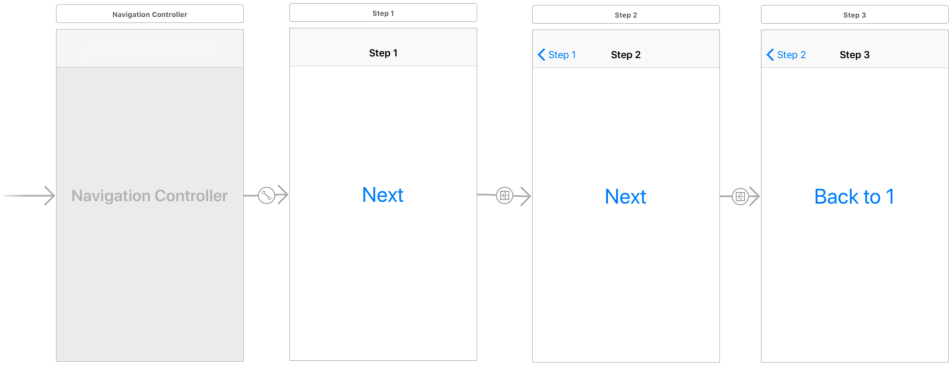
Immediately prior to performing a segue, the source view controller receives two delegate callbacks. First, shouldPerformSegue(withIdentifier:sender:) gives the source an opportunity to consider a segue that's about to be performed and veto it if needed. If approved (or if that method isn't implemented), the source gets a callback to prepare(for:sender:). Since this method provides the UIStoryboardSegue object itself, the source view controller can get the destination view controller and pass data to it prior to the segue actually happening.

Actually, segues go way deeper—and frankly, get way weirder—than this.

Unwind Segues

Let's say you have a navigation controller with segues between several scenes, as shown in the figure on page 45. This kind of thing is common in "drill-down" scenarios. For example, your app's "on-boarding" experience might walk the user through screens for picking their username and password, agreeing to legal terms, creating an avatar, and all the other things your app needs to do.

Thanks to the navigation controller in the beginning of this flow, every scene automatically has a navigation bar with a back button, so the user can go back to the previous step. But what if you get to the end of this flow and you want to go all the way back to the beginning? For example, what if the user cancels out of on-boarding, or they complete it and we want to get back to the login screen? You certainly don't want to programmatically go back through each scene one at a time, right?



Fortunately, this is where you can use one of those buttons up on the scene's dock: the exit button. This allows you to create a new kind of segue, the *unwind segue*.

An unwind segue allows you to return to any earlier scene in the current navigation flow. However, you don't indicate the destination by naming the scene you go to. Instead, you have to implement a special *unwind* method in the view controller of the scene in which you want to return.

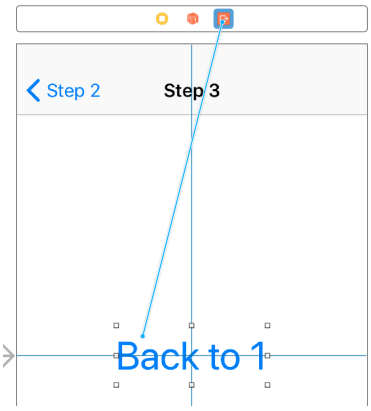
An unwind method needs to take a single `UIStoryboardSegue` parameter, and it must be marked with the `@IBAction` annotation. In Swift, it works better to also suppress the label of the parameter by using `_` for its outer name:

storyboards-behavior/UnwindSegueDemo/UnwindSegueDemo/Step1ViewController.swift

```
@IBAction func unwindToStep1ViewController(_ segue: UIStoryboardSegue) {
}
```

As you can see, the method doesn't actually have to *do* anything, it only needs to exist. That said, if you want to pull values out of the view controller that was exited, you can inspect the `segue`'s `source` property.

Now, you can control-drag from a button, cell, or even a view controller to the dock's exit icon to create an unwind segue. When you release the click on the exit icon, a list of connectable methods appear—these are all of the methods that match the unwind pattern described before. If you choose `unwindToStep1ViewController`, it'll create a segue that unwinds to the scene with that method, namely the first scene.



Unwind segues are a little weird at first, but they're useful in apps with deep navigation.

Modal Segues

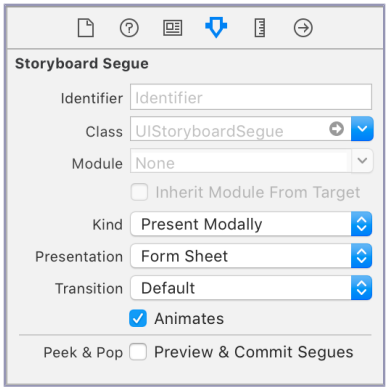
Aside from the scene-to-scene transitions of segues in a navigation flow, the other most common use of segues is in presenting *modal* scenes. These are scenarios where a new view controller is presented and must be explicitly dealt with before the app can continue. In a windowed environment like macOS, modals appear as sheets or dialogs on top of the window they're blocking.

On iOS, the story is a little different. By default, if you create a modal segue, the destination view controller will slide in from the bottom of the screen, as opposed to navigations that slide in from the right.

The modal scene doesn't come equipped with a back button, like it would in a navigation flow, so you need to explicitly provide some way out for the user. Often, this will be an exit segue, either wired to a button, or called programatically. You can also programmatically find the view controller that presented the modal scene, and tell it to dismiss the modal, like this:

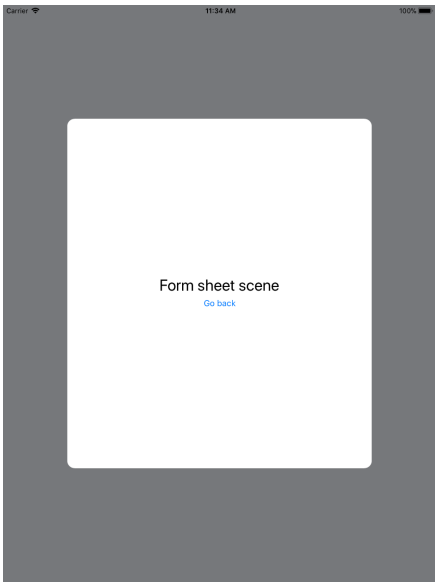
storyboards-behavior/ModalAndPopoverDemo/ModalAndPopoverDemo/ModalSceneViewController.swift

```
@IBAction func handleManualDismissTapped(_ sender: Any) {
    presentingViewController?.dismiss(animated: true, completion: nil)
}
```

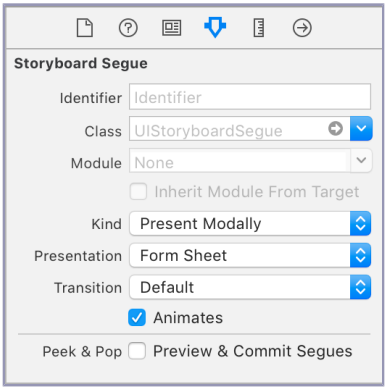


By default, modal segues work the same on the iPhone and iPad. That can be weird on the iPad, since modal scenes are often short interactions like logins or preferences that don't need the full screen space of an iPad. One affordance for this is the *form sheet*. Click a modal segue and bring up its Attributes Inspector to see how this works. If you change the "Presentation" from "Full Screen" to "Form Sheet", you'll get the special behavior on the iPad. On the iPhone, this will continue to be a bottom-up, full-screen modal as usual. But on the iPad, the modal view will now take up only a portion of the middle of the screen, as shown in the figure on page 47.

This is still a modal, so you still need to provide a way to dismiss the view controller via an exit segue or equivalent, just like on an iPhone. Clicking on the gray area outside the form sheet doesn't do anything.



By default, the system will choose a size for a form sheet—currently 540 x 620, regardless of the iPad model—but it's always possible that could change to accommodate much larger or much smaller iPads in the future, so it's smart to use Auto Layout for the contents of the modal scene and be willing to adjust. Or, if you're certain a specific size will work for the contents, you can also enable "Use Preferred Explicit Size" in the modal scene's Attributes Inspector to set a fixed size.



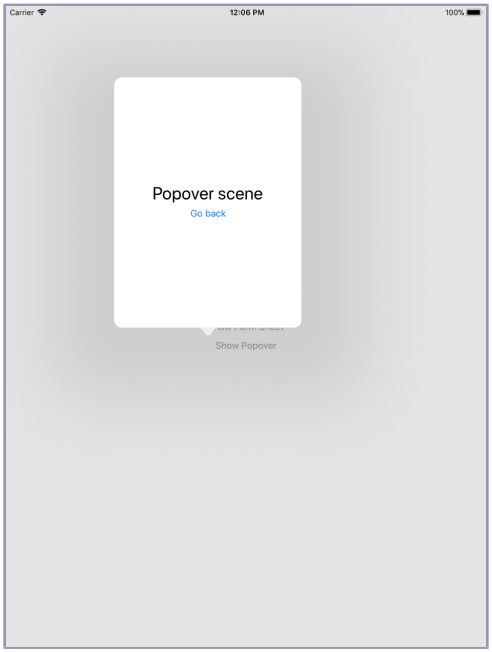
Size Classes in Form Sheets

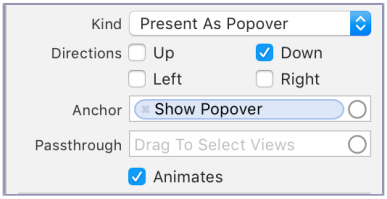
If you use size classes in your Auto Layout or app logic, be aware that the values inside a form sheet may not be what you expect. The horizontal size class is `compact`, and the vertical is `regular`, as if the view controller were being shown on a portrait-orientation iPhone. Oddly, this is the case even if the iPad is in a landscape orientation. If this isn't the behavior you want, you may need some other technique for iPad-specific behavior, like looking at the `userInterfaceIdiom` in `UIDevice`.

Pop-Over Segues

Another option for showing small interactions above the current scene is to use the *pop-over* idiom. In some ways, a pop-over is a lot like a form sheet modal, with the added bonus that the user can tap outside the pop-over to dismiss it. Like the form sheet, you can (and usually should) set an explicit size for it in the Attributes Inspector.

To create a pop-over segue in a storyboard, choose the "Present As Popover" option when control-dragging from the button to the destination scene. You can also turn a segue into a pop-over by selecting the segue, bringing up the Attributes Inspector, and changing the "Kind" to "Present As Popover". On an iPad, the result looks like the following figure:





Unlike a form sheet, which is centered horizontally and vertically on screen, pop-overs are placed relative to an anchor object. If you create the pop-over segue with a control-drag from a button, the button will be the anchor, but you can change this in the segue's Attributes Inspector. You can also use the Attributes Inspector to control from which directions the pop-over will point to the anchor, which in turn will dictate where and how the pop-over can appear.

On the iPhone, the pop-over will be presented as if it were a full-screen modal. This means that if the app runs on both iPads and iPhones, you'll need to present a button or some other UI to use an exit segue or programmatically dismiss the pop-over.

On the other hand, if the pop-over is small, you can make it work as a pop-over on the phone too. To do so, you need to do three things in code:

1. Explicitly declare that the UIModalPresentationStyle is popover.
2. Set a delegate on the view controller's popoverPresentationController.
3. Implement the UIAdaptivePresentationControllerDelegate method adaptivePresentationStyle(for:) to return none.

You can set up the presentation style and assign the delegate inside the view controller's init(coder:):

storyboards-behavior/ModalAndPopoverDemo/ModalAndPopoverDemo/PopoverSceneViewController.swift

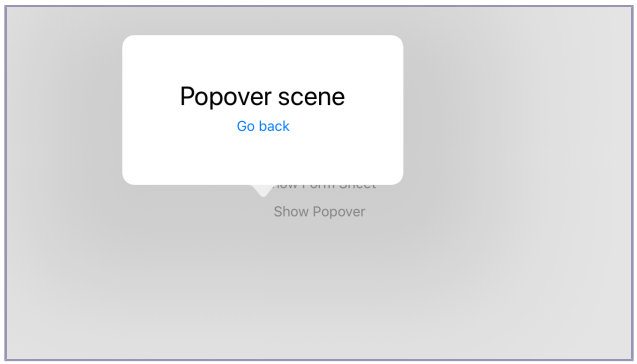
```
private let showAsPopoverOnPhone = true

required init?(coder decoder: NSCoder) {
    super.init(coder: decoder)
    if showAsPopoverOnPhone {
        self.modalPresentationStyle = .popover
        self.popoverPresentationController?.delegate = self
    }
}
```

Assigning the delegate will produce an error until you extend the class to implement the UIPopoverPresentationControllerDelegate protocol. This looks a little odd because you don't need to implement any methods from that protocol. However, the adaptivePresentationStyle(for:) method you need is in its superprotocol, so this is where you override it to return the none style:

storyboards-behavior/ModalAndPopoverDemo/ModalAndPopoverDemo/PopoverSceneViewController.swift

```
extension PopoverSceneViewController:
  UIPopoverPresentationControllerDelegate {

    func adaptivePresentationStyle(for controller: UIPresentationController)
        -> UIModalPresentationStyle {
        return .none
    }
}
```

Run this on an iPhone, and now the pop-over works like it does on an iPad, as seen in the figure on page 50. And with a small enough pop-over, it'll look just fine in portrait or landscape. As a bonus, now that you inherit the pop-over behavior of dismissing by tapping outside the pop-over, you can even remove the explicit dismiss button.



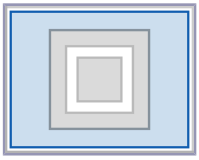
Embedded Scenes

One thing storyboards tend to reinforce is that a single view controller is responsible for one screen-full of content. After all, there is a one-to-one correspondence of scenes to view controllers, and the scenes themselves are shaped like iPhone or iPad screens, depending on what device you've set the "View As" control at the bottom of Interface Builder to show.

This leads to the problem of the "Massive View Controller". If there's a lot going on in one scene, the natural place for all the code to deal with that is in the view controller. And when that view controller is responsible for handling UI events, populating table or collection views, dealing with rotation or remote-control media events, etc., it quickly leads to the view controller's size and complexity getting out of hand. Nobody sets out to write a 2,000-line UIViewController, but some mornings you wake up and *there it is*.

This section is inspired by Dave DeLong's talk "A Better MVC" at Swift by Northwest 2017,[1] and also his follow-up blog post.

One strategy to avoid the massive view controller is to break the 1 view controller == 1 screen mindset. Storyboards give us a technique to break that habit: *container views*. With this approach, we use multiple view controllers in a scene, each one smaller and more focused than would be possible otherwise.

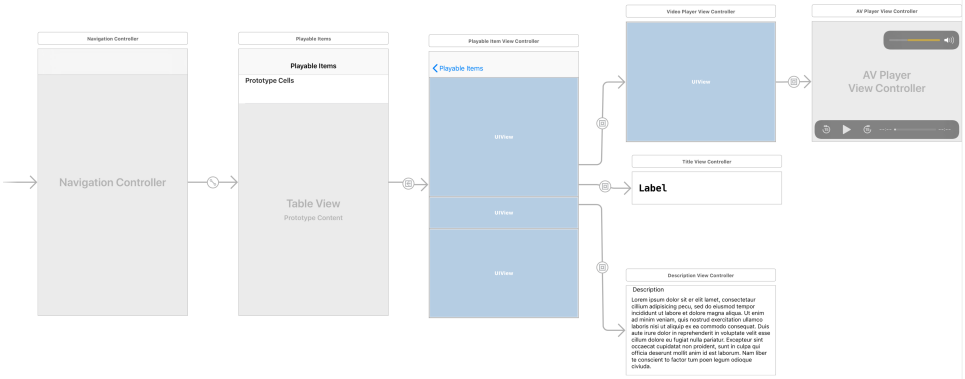


If you pick up the *container view* icon from the library and drop it in a scene, two interesting things will happen. It will place a view in the scene, which works like any other plain UIView, in that it can be laid out in the scene with Auto Layout constraints. But it also adds another whole scene to the storyboard, which is connected to the container view with a segue.

1. https://davedelong.com/blog/2017/11/06/a-better-mvc-part-1-the-problems/

The idea here is that the second scene is where all the content comes from; this is where you can add subviews like labels, buttons, sliders and what have you. More importantly, since this is a completely separate scene, it has its own view controller. And this is the key to breaking up a massive view controller: if parts of that parent screen can take care of themselves, you can split out their functionality into completely separate view controllers.

There's an example of this in the download code. Take a look at the following storyboard and we'll walk through how it works. The app shows a table of PlayableItem instances, which is just a struct with a streaming video URL, a title string, and a description string. When the user selects one of the playable items, the app segues a detail screen, the PlayableItemViewController.



The PlayableItemViewController scene in the middle is where things get interesting. This has just three subviews, but all of them are container views. One goes to a VideoPlayerViewController scene, one to a TitleViewController, and one to a DescriptionViewController. These scenes are trivial; for example, the title screen manages a single label, and the description screen just has a text view, each of which is updated by setting a playableItem property. The video scene is a little weirder, so we'll get back to that one.

Because these scenes are so simple, they've almost no code. They're easy to expose to unit testing (which is covered in Chapter 8, *Automated Testing*, on page 159), and if there is a problem with, say, the description, it's easier to go straight to that source file, rather than searching through a massive view controller's source.

There is, however, one trick to using container views. Notice that the connections to the host scene are actually segues. These are *embed segues*, and they have an important use. Since the parent scene doesn't have a direct connection to the child scenes, it can't access them or their properties. This is a problem, because you need a way for the parent to pass the selected PlayableItem to the child scenes, which will use it to update themselves.

The solution is to use the segue. When the child scene loads and is embedded into the parent, the parent gets a one-time call to prepare(for:sender:), with the embed segue as its parameter. The segue's destination is the view controller being embedded. So the trick here is to save a reference to the view controller that's being embedded.

In the PlayableItemViewController, you set up properties for the three child view controllers:

storyboards-behavior/EmbeddedVCDemo/EmbeddedVCDemo/PlayerSceneVCs/PlayableItemViewCon¬troller.swift

```
private var videoPlayerVC: VideoPlayerViewController?
private var titleVC: TitleViewController?
private var descriptionVC: DescriptionViewController?
```

Then, you implement prepare(for:sender:) to catch each of the embed segues and save off its destination to the correct property. In the example, there are identifier strings on each of the segues to make this step work nicely with a switch statement (and it's a good habit to always name your segues anyway):

storyboards-behavior/EmbeddedVCDemo/EmbeddedVCDemo/PlayerSceneVCs/PlayableItemViewCon¬troller.swift

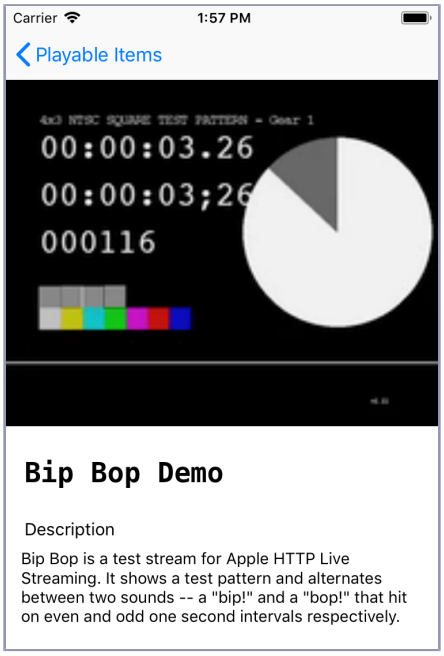
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case "embedTitle":
        if let titleVC =
            segue.destination as? TitleViewController {
            self.titleVC = titleVC
            titleVC.playableItem = playableItem
        }
    case "embedDescription":
        if let descriptionVC =
            segue.destination as? DescriptionViewController {
            self.descriptionVC = descriptionVC
            descriptionVC.playableItem = playableItem
        }
    case "embedVideoPlayer":
        if let videoPlayerVC =
            segue.destination as? VideoPlayerViewController {
            self.videoPlayerVC = videoPlayerVC
            videoPlayerVC.playableItem = playableItem
        }
    default:
        break
    }
}
```

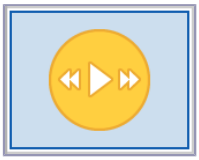
Now that you've got references to the child view controllers, any time the parent's playableItem is set, it can send that struct to those child view controllers:

storyboards-behavior/EmbeddedVCDemo/EmbeddedVCDemo/PlayerSceneVCs/PlayableItemViewCon¬troller.swift

```
var playableItem: PlayableItem? {
    didSet {
        videoPlayerVC?.playableItem = playableItem
        titleVC?.playableItem = playableItem
        descriptionVC?.playableItem = playableItem
    }
}
```

And that's the key! With that, the child view controllers can update themselves from whichever fields of the playableItem are relevant to them. The running app is shown in the following figure:





The idea of container views and embedded view controllers also helps to explain how the "AVKit Player View Controller" icon works. As a view controller, it can't be dropped directly into a scene, and dropping it on a storyboard makes it its own scene. For beginners, this is confusing: can the player only be its own full-screen scene? Nope, the way you want to use it is as a child view controller. This is shown at the top right of the storyboard figure

shown earlier. The VideoPlayerViewController child scene has a child view of its own. To do this, the example project deletes the default scene that came with the container view icon, and replaces it with an embed segue to the AVKit Player scene.

Storyboard References

Once you start adding more and more scenes and segues, your storyboard will naturally get more and more complex. A simple app can easily have more than a dozen scenes, and a complex app can go into the hundreds.

This leads to all kinds of problems. Finding anything in the storyboard becomes a chore when you're visually overwhelmed with scenes and segues. Making changes becomes hazardous, because if you happen to select the wrong thing and delete it, you can break a colleague's work and not even realize it. Most notoriously, storyboards can be challenging to share in a source control system, causing git merges that are nearly impossible to make sense of (though we'll try our best in Chapter 10, *Source Control Management*, on page 189).

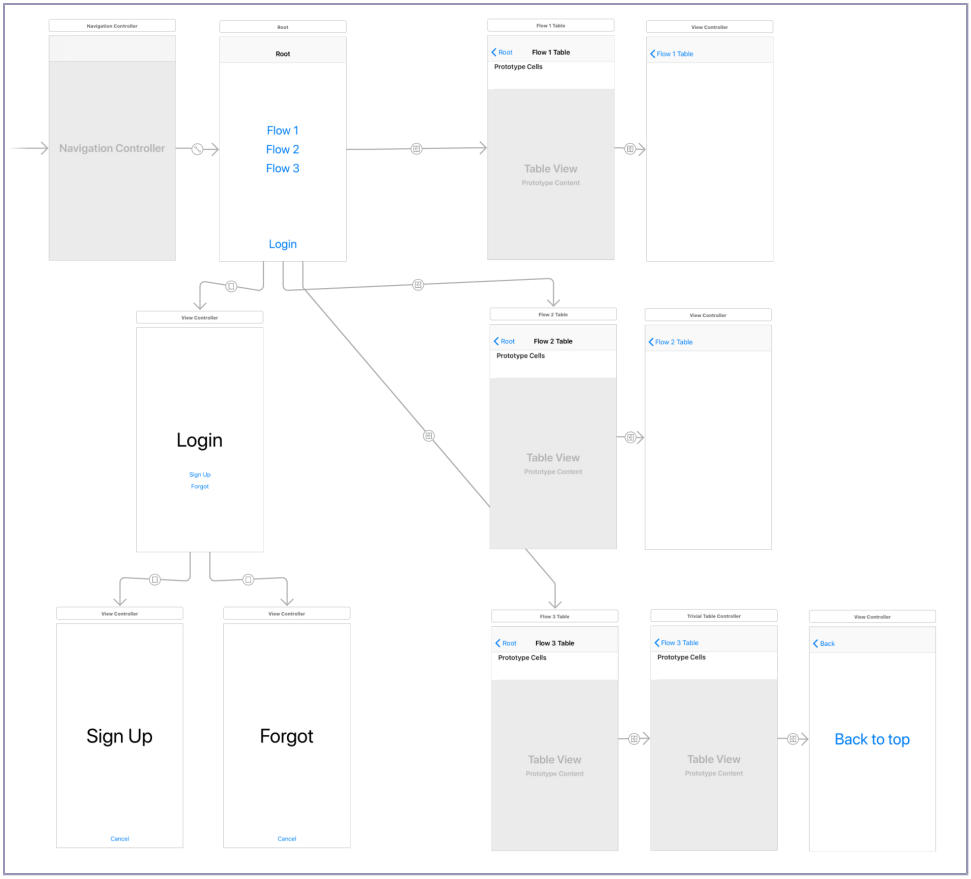
Fortunately, Xcode gives you a fighting chance to rein in this madness by splitting up one storyboard into many files, and connecting them through *storyboard references*.

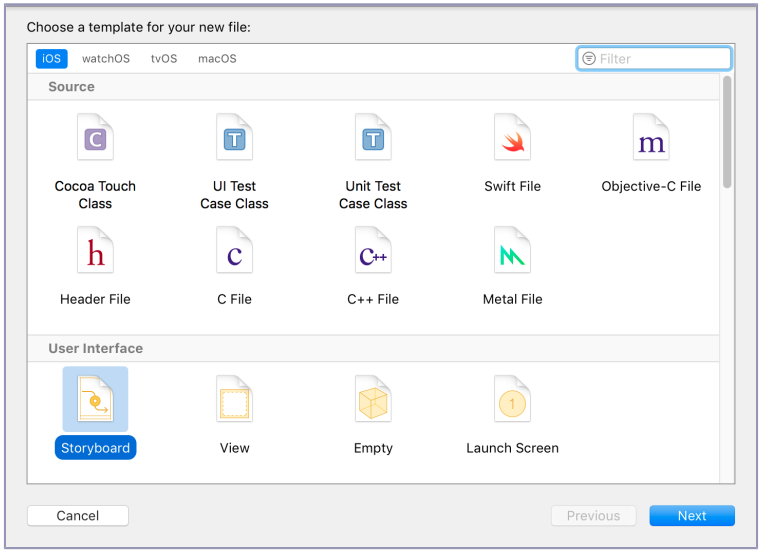
Consider the storyboard in the top figure on page 55. Arguably, it's not *that* bad: a root scene that branches off into three navigation flows, plus a login/sign-up/forgot-password flow. But even at this size, it's hard to read (and hard to reproduce in a book).

To split up storyboards, one good approach is to find scenes that are related to each other and have common functionality. Each of the three left-to-right flows are good candidates, as are the three scenes for login, sign-up, and forgot password.

Start with the login stuff. Select a group folder in the File Navigator—or perhaps create a new one just for storyboards—and select the File > New > File menu item (⌘N). When the new file sheet comes up, scroll down to "User Interface" and select the Storyboard icon, as seen in the bottom figure on page 55. Then, give the new file a meaningful name, like Login.storyboard.

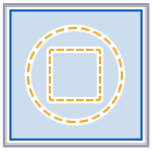
This creates an empty storyboard, just like the Main.storyboard a new project starts with, but with no scenes.


Navigation Controller
Root
Flow 1
Flow 2
Flow 3
Login
Flow 1 Table
Prototype Cells
Table View
Prototype Content
View Controller
Flow 1 Table
Login
Sign Up
Forgot
Flow 2 Table
Prototype Cells
Table View
Prototype Content
View Controller
Flow 2 Table
Sign Up
Cancel
Forgot
Cancel
Flow 3 Table
Prototype Cells
Table View
Prototype Content
Trivial Table Controller
Flow 3 Table
Prototype Cells
Table View
Prototype Content
View Controller
Back
Back to top


Choose a template for your new file:
iOS
watchOS
tvOS
macOS
Filter
Source
Cocoa Touch Class
UI Test Case Class
Unit Test Case Class
Swift File
Objective-C File
Header File
C File
C++ File
Metal File
User Interface
Storyboard
View
Empty
Launch Screen
Cancel
Previous
Next

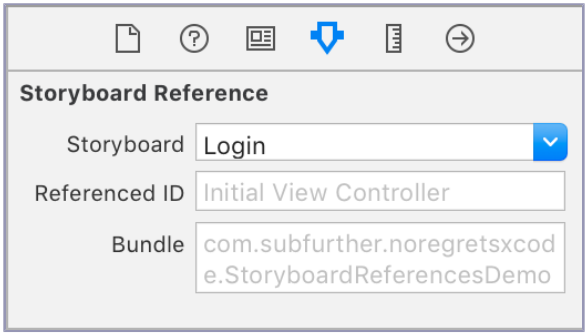
Now, go back to the Main.storyboard and select the scenes you want to move. They can be selected either by command-clicking view controller icons in the Document Outline, or by selecting a box region in the storyboard itself. So, select the three login scenes and cut them from the main storyboard with Edit > Cut (⌘X). Then, switch to Login.storyboard and do Edit > Paste (⌘V). Voilà! You now have a clean, three-scene storyboard.

However, these scenes are living in a different storyboard, completely cut off from the root view controller back in Main.storyboard. If you click the "Login" button now, nothing will happen. You have to create a connection from Main.storyboard to Login.storyboard.



The way you do that is with the Storyboard Reference icon in the Library, which looks like a ghostly outline of the regular View Controller icon. Drag one of those into the storyboard, near the scene that will segue to it. This will appear as just a small rectangle that reads "Storyboard Reference".

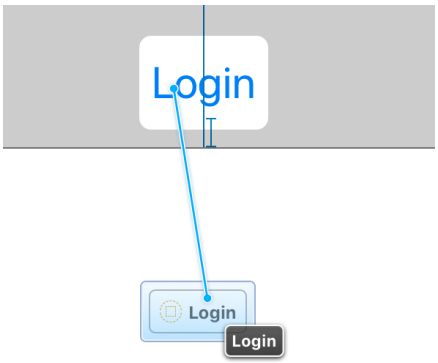
What you need to do now is to select the storyboard reference and bring up its Attributes Inspector. The first field, "Storyboard", is the name of the file you want to refer to (minus the .storyboard extension). You can type in "Login", as seen in the following figure, or use the arrow on the right to pop up a menu of storyboard files in the project.



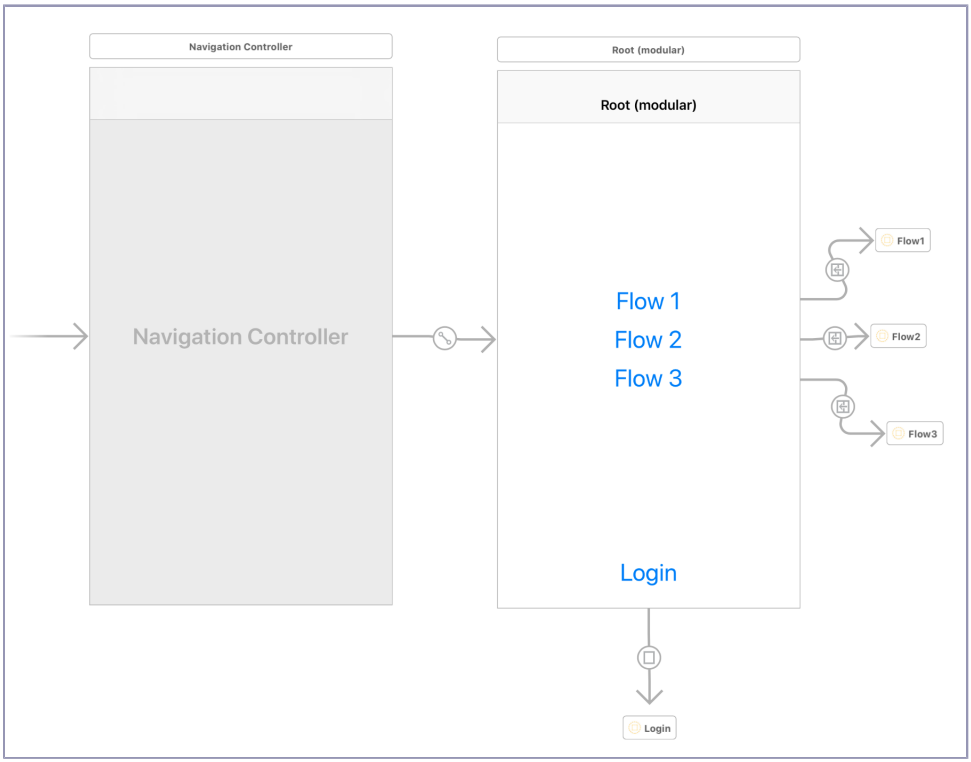
The second field, "Referenced ID" is somewhat optional. To refer to a specific scene in the Login storyboard, you need to give that scene an identifier (look for "Storyboard ID" in the Identity Inspector) in the Login storyboard. Or, you can leave this field blank, and just set one scene in the Login storyboard as the initial scene, by going to the scene's Attributes Inspector and selecting the "Is Initial View Controller" check box. The latter approach was used in the download code.

So now you have a reference to the Login storyboard, but it's not doing anything for you. The trick now is to treat this storyboard reference like any other scene

in the storyboard... since, after all, it's just a reference to one scene in that other storyboard. So, you can just control-drag from the root view controller's "Login" button to the storyboard reference, and choose a segue style.



That's all it takes. If you run the app now, it works exactly like it did when you had the big ugly storyboard. Perform this same technique with the other navigation flows and you can drastically simplify the main storyboard, as seen in the following figure:



The great thing is that this still works like the big storyboard. Within each modular storyboard you can create new scenes and add segues between them, and if any of these storyboards get too big, you can just split those up further. Somewhat surprisingly, unwind segues will work across storyboards—in the last scene of File3.storyboard, there's a "Back to top" button, and if you control-drag to the exit button, it finds the unwindToRootViewController action just fine.

Custom Views

Now that you've had all sorts of fun with segues and storyboards, turn your attention back to individual scenes.

Not everything you want to put in an app is best presented with a built-in component. Sometimes, you want to create views that use your own drawing code. And of course, you're always building custom view controllers, but you tend to do most of that work in code, not in the storyboard.

Actually, Xcode provides several tools for customizing the custom classes in the storyboard itself. This means the storyboard can be reserved for visual design work, and you don't need to do as much in code.

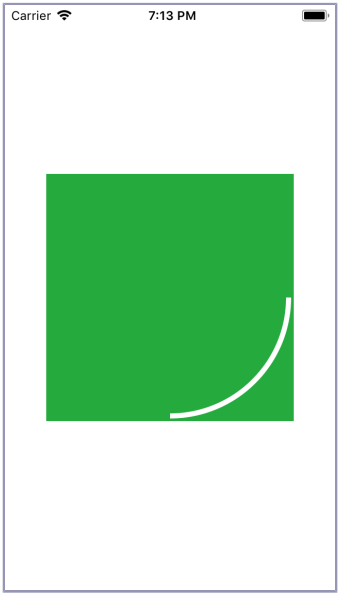
IBInspectable Properties

Start by imagining you have a UIView subclass that only knows how to draw an arc with given start and end angles (in radians), in a certain color and with a certain line width. Here's its simple implementation:

storyboards-behavior/IBInspectableDemo/IBInspectableDemo/ArcView.swift

```
class ArcView: UIView {

  var startAngle: CGFloat = 0
  var endAngle: CGFloat = CGFloat (Double.pi / 2)
  var width: CGFloat = 5.0
  var fillColor: UIColor = UIColor.white

  override func draw(_ rect: CGRect) {
    guard let context = UIGraphicsGetCurrentContext() else { return }
    context.setLineWidth(width)
    context.setStrokeColor(fillColor.cgColor)
    let center = CGPoint(x: rect.size.width / 2,
                         y: rect.size.height / 2)
    context.addArc(center: center,
                   radius: (rect.size.width / 2) - width,
                   startAngle: startAngle,
                   endAngle: endAngle,
                   clockwise: false)
    context.strokePath()
  }
}
```

To put an instance of this view in the storyboard, drag a plain UIView from the library, and use the Identity Inspector to change its class to ArcView. From here, position it with constraints, and set its background color with the Attributes Inspector.



If you run it as-is, you get an arc with the default color, width, and start and end angles, as seen in the accompanying figure. It's... OK, I guess. Problem is, at this point, to change what it looks like, you need to set those properties in code, right? Meh.

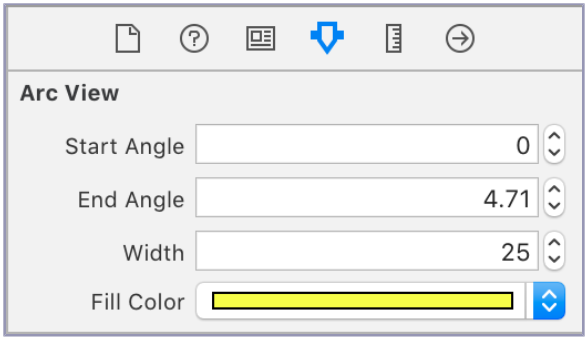
Actually, you might remember from the previous chapter that any property of a storyboard object can be set by name in the Identity Inspector's "User Defined Runtime Attributes", as long as the property is one of a small number of supported types (numbers, strings, points, sizes, rectangles, or colors). All your properties meet these criteria, so that would actually work here.

As it turns out, you can do better. Properties that can be set in the runtime attributes table can get a proper editor by appending the @IBInspectable attribute to them. So, imagine you rewrite the properties as follows:

storyboards-behavior/IBInspectableDemo/IBInspectableDemo/ArcView.swift

```
@IBInspectable var startAngle: CGFloat = 0
@IBInspectable var endAngle: CGFloat = CGFloat (Double.pi / 2)
@IBInspectable var width: CGFloat = 5.0
@IBInspectable var fillColor: UIColor = UIColor.white
```

By doing this, those four properties will get their own section in the Attributes Inspector:



Notice that the editor labels have automatically changed the camel-case variable names into title case; startAngle becomes “Start Angle”. It’s nice, but you can do a lot better.

IBDesignable Views

It’s a nicer editor, but this isn’t doing much to make the custom view particularly suited to being designed visually. If you hand the storyboard off to a designer, they can fill in the values, but they’ll still have to run the app to see if it looks right. What would be ideal is to see changes to those properties reflected within Interface Builder itself, so you don’t have to leave the storyboard at all.

Surprise! Xcode supports this very feature.

To allow a view to be previewed in Interface Builder, start by adding the @IBDesignable attribute to its class declaration, like this:

storyboards-behavior/IBDesignableDemo/IBDesignableDemo/ArcView.swift

```
@IBDesignable class ArcView: UIView {
```

This tells Interface Builder that it should attempt to draw the view inside the scene, just like if it were running in the app. Since the only thing our class does is draw itself, this is a pretty trivial change.

You do need to do one other thing, however. Any time you set one of the properties in the Attributes Inspector, you need to tell the view to redraw itself. Of course, that’s probably something you want anyway, so that if the properties change at runtime, the view will update itself. So you can just provide a private method to force a redraw:

storyboards-behavior/IBDesignableDemo/IBDesignableDemo/ArcView.swift

```
private func updateView() {
  self.setNeedsDisplay()
}
```

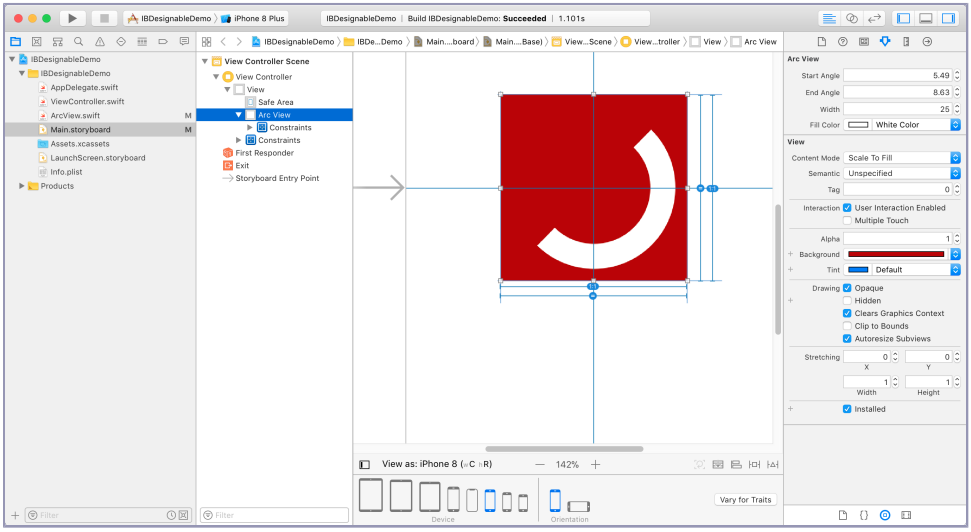
And then call that from any property where you want to force a redraw:

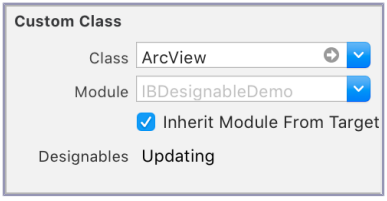
storyboards-behavior/IBDesignableDemo/IBDesignableDemo/ArcView.swift

```
@IBInspectable var startAngle: CGFloat = 0 {
  didSet {
    updateView()
  }
}

@IBInspectable var endAngle: CGFloat = CGFloat (Double.pi / 2) {
  didSet {
    updateView()
  }
}
```

```
@IBInspectable var width: CGFloat = 5.0 {
  didSet {
    updateView()
  }
}

@IBInspectable var fillColor: UIColor = UIColor.white {
  didSet {
    updateView()
  }
}
```

Once this is written, the arc view starts drawing itself *inside* the storyboard, as seen in the following figure. Cool!





There are a couple things to watch for. When you first visit a storyboard with IBDesignable views, it can take some time to build and run the subviews. When this happens, the Identity Inspector shows the “Designables” as “Updating”. When the designables are built and running, this changes to “Up to date”. Also, if your view is complex—as happens if your designable view is actually a superview to a complex hierarchy of subviews—Xcode may suffer an internal crash when building and running the designables. If this happens, the Designables label will actually say “Crashed”. At least they’re honest, right? (If this happens, performing a clean build or restarting Xcode may help. One of our tech reviewers puts all custom

components in their own framework, so when they crash, Xcode only has to rebuild that framework.)

When it works, IBDesignable is a cool feature. But do keep in mind that setting values in the storyboard like this makes the storyboard the source of truth. Sometimes, that's elegant. But there may be cases where you go searching for where a value is getting set in code, only to find it's a storyboard property like this. So, you want to be mindful of what kinds of things make sense to be designable.

IBDesignable Code Tips

Since running as a designable is mostly, but not exactly, like running in the app, there are a few places where you can tweak the behavior of your code for design-time use. You can use the conditional-compilation flag #if TARGET_INTERFACE_BUILDER to box out sections of code that should only be run if the view is being run inside of Interface Builder; this code won't be compiled for debug or production builds of the app.

NSObject also defines the method prepareForInterfaceBuilder(), which will only be called for design-time manipulation of objects. Again, if you have setup code you only want to run in design mode, this is where you can do it. This can help if there are view controllers as part of a designable view hierarchy, and the view controllers would normally get their contents from some source that's not available at design-time, like from a network call. Since prepareForInterface-Builder() is available to all NSObjects, not just views, the view controllers could override it to provide design-time-only contents.

Wrap-Up

In this chapter, you dug into how storyboards work. The important thing about storyboards is how scenes relate to one another, something that's established with segues. Simple segues do things like navigation-flow pushes and modal interactions, but segues also let you make iPad-friendly form sheets and pop-overs. And by using container views and their embed segues, you can split up complex view controllers into smaller and more manageable pieces.

Modularity also helps you as storyboards get big and complex. By using storyboard references, you can break things down into smaller storyboards with just the scenes needed for part of the app. This keeps storyboards easier to

read and less likely to get mangled by source control when they're shared with peers.

Finally, you got a chance to make your own storyboard components, by adding IBInspectable and IBDesignable attributes to custom view classes, which allow them to be drawn and updated at design-time, inside the storyboard itself.

Now that you've had lots of fun playing with the user interface, you should start thinking about writing some code, to provide the apps' functionality. In the next chapter, you'll look at all the ways Xcode makes that a more pleasant and productive experience.

CHAPTER 4

Editing Source Code

As developers, we don't actually spend most of our time configuring projects or laying out storyboards. Most of the time, we're coding. And, in turn, that means we spend most of our time writing and editing source code. So, it's critical that Xcode have a pleasant and productive code editing UI. And as Mac users, we have high standards, thanks to decades of great text editors on the platform, like BBEdit and TextMate. If Xcode can't at least be as good as those, we'll be sorely tempted to take our source code elsewhere.

Fortunately, Xcode's source editor got a *lot* better in Xcode 9, based in part on the work that was done for the Playgrounds app for iOS. In this chapter, we'll look at ways to make our coding life better.

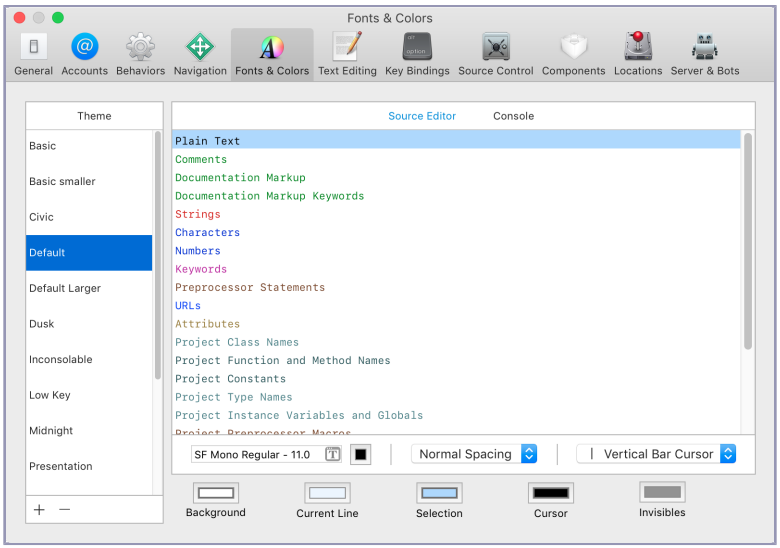
Theming the Code Editor

How's your eyesight? How big is your screen? What's the lighting like in your room? All of these factors will play into what you want source code to look like when you're editing it.

Xcode lets you customize fonts, sizes, and colors, packaging up related settings as *themes*. Visit the themes interface by bringing up Xcode's Preferences and selecting the "Fonts & Colors" tab. The theme editor is shown in the figure on page 66.

The left column lists the available themes, with the current theme highlighted. By default, you'll get "Default", which uses a white background, primary colors, and the font `SF Mono Regular`.

The right side of the editor shows the styling for every kind of syntax that Xcode knows how to style. You can choose one line and change the font, size, or color for just that kind of syntax, or multi-select with shift-clicking, command-clicking, or select-all (⌘A) and change all lines at once.



The right side of the pane also lets you set colors for non-text elements, like the background, cursor, selection, current line, and invisibles. It also lets you change the cursor appearance and spacing, and tabs at the top let you edit the appearance of the debugging console.

Of the built-in themes, the primary choice is between a dark-on-white theme (Basic, Default, Low Key, Sunset) or a bright-on-black theme (Civic, Midnight). Beyond this, there are a few special-purpose themes. The most important is “Presentation”, which is basically Default at a much larger size, appropriate for showing on a projection system if you’re teaching or presenting to a room and want people in the back to be able to read your code. Presentation is also useful for screencasting or livestreaming, where the noise from video compression can make smaller type unreadable.

It’s easy to make your own themes. Duplicate one of the existing themes by clicking the plus (+) button at the bottom of the list on the left, give your new theme a name, and then edit its settings on the right. The “Inconsolable” theme in the earlier screenshot is one I made by duplicating the Default theme and simply changing the font to Inconsolata for every syntax type.

Of course, not all of us developers have a good design eye for this kind of thing, or the patience to make lots of fiddly little edits. Fortunately, the Xcode community has our back. Themes can be easily packaged and redistributed. Hugo Doria has a big collection of third-party Xcode themes, with screenshots, in a GitHub project.[1]

1. https://github.com/hdoria/xcode-themes

Joe asks:
Can You Use a Variable-Width Font?

Since the theme editor uses the normal macOS font picker, you could choose a variable-width font for your code, rather than the traditional monospaced. *Yes, even Comic Sans.* Should you?

I ran an informal Twitter poll[a] and responses were overwhelmingly against the idea, but a handful of people say they already use a variable-width font for code.

The advantage is legibility: display fonts are designed to be easily read, after all. The disadvantage is all the places where fixed-width is assumed, such as formatting. In particular, multi-line statements won't line up the way you expect: they don't indent far enough in C or Swift, and are all over the place with Obj-C.

Also, if you share code with colleagues, having it look so different on your screen than everyone else's could cause problems; a font with tight spacing like Gill Sans might tempt you to cram more on a line than you should.

Still, it's not an inherently unreasonable practice. Maybe try it and see if you like it. After all, a tasteful font like Verdana, shown here, looks pretty good in Xcode:

```
13    override func viewDidLoad() {
14        super.viewDidLoad()
15        // Do any additional setup after loading the view, typically from a nib.
16    }
```

a. https://twitter.com/invalidname/status/891788185323393029

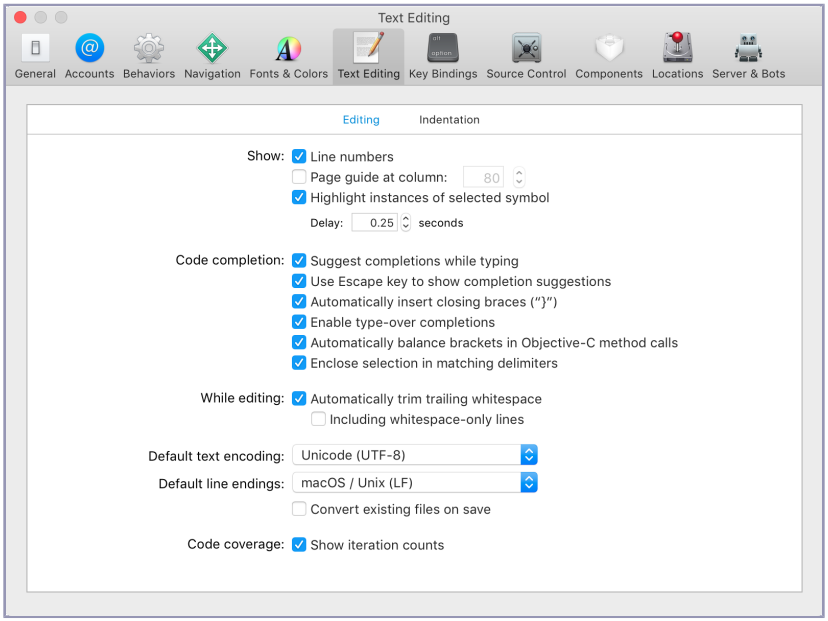
There are scripts on this page to install all the themes he has collected, and it's also easy to just do it yourself. If it doesn't already exist, create the directory ~/Library/Developer/Xcode/UserData/FontAndColorThemes/. Then download .dvtcolortheme files and copy them to this folder. .xccolortheme files work too, and if you edit an existing third-party theme, the edited version will be created as a new .xccolortheme file. For example, here's my themes folder:

```
Ashe:~ cadamson$ ls -ltr ~/Library/Developer/Xcode/UserData/FontAndColorThemes/
total 112
-rw-r--r--@ 1 cadamson  admin  5617 Mar 10  2013 Raspberry Sorbet.dvtcolortheme
-rw-r--r--@ 1 cadamson  admin  5675 Apr 22  2013 Rearden Steel.dvtcolortheme
-rw-r--r--  1 cadamson  admin  5447 Oct 30  2013 Inconsolable.dvtcolortheme
-rw-r--r--  1 cadamson  admin  5387 Mar  5  2014 Basic smaller.dvtcolortheme
-rw-r--r--  1 cadamson  admin  6748 Jul 21  2015 Default Larger.dvtcolortheme
-rw-r--r--  1 cadamson  admin  7395 Jul 25 21:00 Rearden Steel.xccolortheme
-rw-r--r--  1 cadamson  admin  7209 Jul 30 14:56 Raspberry Sorbet.xccolortheme
```

You can see my home-made Inconsolable theme there from 2013, along with my edits to the third-party themes Raspberry Sorbet and Rearden Steel.

Setting Text Editor Preferences

While we're still looking at Xcode preferences, switch over to the next tab, "Text Editing". This is the panel with lots of fiddly little details for the editor.



As seen in the figure, there are two sub-tabs here: Editing and Indentation. The Editing preferences are mostly on-off choices, grouped as follows:

- *Show:* The first setting in this section is whether to show line numbers in the gutter on the left side of the code editor. This is off by default, but I like to always have it on, because once methods or functions are over a certain number of lines, that's a code smell that demands attention. The same goes for letting the file reach 1,000 lines (or, heaven help us, 10,000). Line numbers also widen the gutter, which makes breakpoint buttons (see *Finding Bugs with Breakpoints*, on page 109) wider, and thus easier to see and click. You can also go to an arbitrary line number with the menu item Navigate -> Jump to Line in *Current File Name* (⌘L).

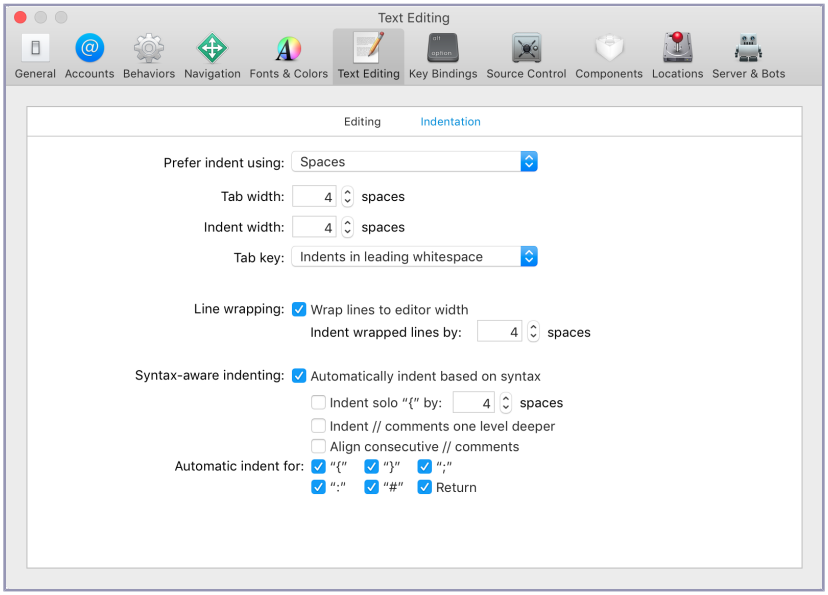
  Another potential best practice is to turn on the next setting: Page Guide. This puts a soft highlight on the first 80 characters of a line—the value is settable—so once you exceed that width, you should start looking for a way to break up the line.

  The last general setting in this section is to automatically highlight all instances of any symbol you select, or even arrow into. This highlighting

can give you a subtle clue for situations like where a variable is declared too far away from where it's actually used.

- *Code completion:* The code completion section has about a half-dozen behaviors indicated by check boxes, all of which are enabled by default. We'll talk about actually using code completion a little later, and chances are you're never going to want to tweak these settings, unless code completion is not your thing and you choose to disable it entirely.
- *While editing*, *Default text encoding*, and *Default line endings*: These are pretty self-explanatory, and unless you're sharing a codebase with developers on a non-Mac platform, it's rare to need anything but the defaults.
- *Code coverage:* Leaving the "Show iteration counts" check box enabled adds a small gutter on the right side of the code editor. Once the project has unit tests—which we'll be covering in Chapter 8, *Automated Testing*, on page 159—each line will display a number showing how many times that line of code was executed during the unit tests. If you want to expose most or all of your codebase to unit testing, this will help you find code that needs tests.

The Indentation sub-tab, seen in the following figure, is somewhat more self-explanatory. First, you set whether you want to default to using tabs or spaces for indentation, and how many space to use for each. This is a default for all your projects; however, it can be overridden on per-project or per-file basis with the File Inspector (⌥⌘1).



The rest of the Indentation settings provide behaviors for syntax-aware indentation, which provides behaviors like indenting after you start a loop or conditional. The handy effects of these preferences are described in *Code Formatting*, on page 78.

Faster File Navigation

So let's turn our attention to the editor pane. Selecting any source file from the Project Navigator will fill the content pane with that file's contents.

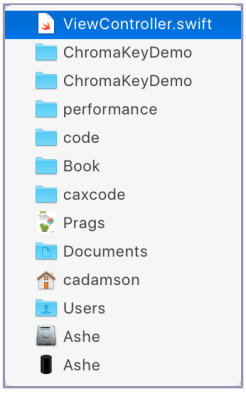
Navigating with the Jump Bar

You may think that using the Project Navigator to switch between files is the fastest way to move around, but that's not always the case. There are other ways, particularly between related files, which can be faster. Notice at the top of the editor, there's a strip of buttons and pop-up menus. This is the *Jump Bar*.

For now, let's skip the first menu item, "Recent Items" (the four boxes), and move on to the more recognizable options, like the obvious "Go Back" and "Go Forward" buttons (the left and right arrow chevrons). Like pages in a browser, these let you navigate through your file history.

To the right of Go Back and Go Forward, there's a breadcrumb-like display of the hierarchy of what you're currently editing: project, group, file, and symbol (i.e., a method, a property, etc.).

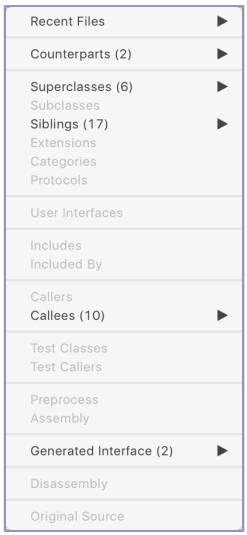
Clicking on these jump bar items shows siblings within the parent structure. For example, if you click on the group folder, the pop-up shows the other groups. If you click on a source file, it shows that file's siblings within its group.



Hidden Jump Bar Awesomeness

There's a hidden reveal-in-Finder functionality in the jump bar too: just control-click or command-click a project, group, or file in the jump bar to get a pop-up with the filesystem hierarchy of that file. Selecting any item from this menu reveals that file or folder in the Finder. If you don't have a Finder window open for your project, revealing its folder via one of its files like this is probably the fastest way to do it.

The final item in the jump bar is individual symbols within a source file, like properties or methods. The item shown here automatically updates as you move around the file while editing, and you can click the item to show all symbols in the file and jump to one quickly. We'll make more use of this menu later, in *Organizing the Symbols Menu*, on page 72.



Now, turn your attention back to the leftmost icon on the jump bar, the four-boxes icon identified by the tooltip as the *Related Items* button. Tap it and you'll get a pop-up menu with variety of hierarchies to navigate to files: Recent Files, Counterparts, Superclasses, Subclasses, Siblings, Callers, Callees, etc. Not all of them will apply at any given time, based on what file you're editing and what symbol is selected (keeping in mind that just having the cursor anywhere within a given method is considered selecting it for our current purposes).

Xcode populates this menu by analyzing your code *while* you write it. For example, it figures out what your super- and subclasses are and creates menu items for them. It also figures out what calls or is called by the code and creates menu items for those too ("Callees" and "Callers", respectively). These can help you notice weird behaviors and entanglements in your code that you don't want.

Navigating to Counterparts

Back on the far left of the jump bar, one of the Related Items menu items is called "Counterparts" and this deserves special mention, because it can be a very common task. In fact, it's so common that it has its own keyboard shortcut, ^⌘↑, to show the next counterpart.

But that begs the question of "what the heck is a counterpart?" A *counterpart* is a known relationship between two files. The most obvious and common example is in C-based languages—C, C++, and Objective-C—where there are typically separate implementation and header files. These two are counterparts: if you're editing `foo.c`, jumping to the next counterpart will open `foo.h`, assuming it exists. The same goes for other implementation file types, like `.m` for Obj-C, `.cpp` for C++, and `.mm` for Objective-C++.

Only certain file types have counterparts. Storyboards, Asset Libraries, and `.plists` don't. For `.swift` files, there are two programmatically generated counterparts. The first is a `.h` file offering an Objective-C header file for your Swift

members, reachable only via the Recent Items menu. The second counterpart, reachable by either Recent Items or the keyboard accelerator, is a pseudo-file of the form YourFile.swift (Interface). This is a Swift-syntax interface showing your file's various members, but without implementations, kind of like a C-language header file. For example, for an empty view controller, the interface looks like this:

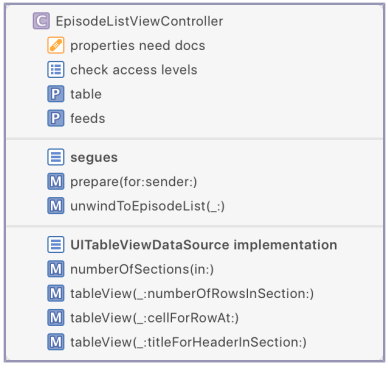
```
import UIKit
internal class ViewController : UIViewController {
    var foo: NSObject
    override internal func viewDidLoad()
}
```

Neither of these Swift interfaces shows any private or fileprivate members of your file, so this can be a good way to reveal what your file exposes to potential callers, reminding you to close off access to properties or methods you don't want outsiders to be able to call.

Comment Magic

There are several useful editing tricks that won't be found in any of the menus or keyboard shortcuts. These tricks are invoked by putting specially formatted comments in your code. Arguably, you should be commenting your code anyway, but with the tricks in this section, you'll pick up navigational, organizational, and documentation wins as a bonus.

Organizing the Symbols Menu



The last menu of the jump bar is the current file's symbols: properties, methods, functions, etc. It's easy enough to jump around in the file by scrolling through this menu, but a lot of times, there are too many members for it to be practical to find anything.

One solution to this is that if you start typing while the symbol list is open, a text bar will appear at the top, and immediately filter the list to only show symbols matching what you've typed. For example, in a view controller source file, you might type viewWill, and the list will filter down to viewWillAppear(), viewWillDisappear(), and so on.

You can also use specially formatted comments to add your own organizational members to this menu. In Swift or the C-based languages, a comment starting with // MARK: (the capitalization matters) will add the remaining text of the comment as a menu item. You can also use hyphen (-) characters to add a divider line; a hyphen before the comment text adds a line above it, and a hyphen after the comment text creates a line below it. For example, to create a section header for all the UITableViewDataSource methods, you would just add a comment like:

```
//MARK: - UITableViewDataSource implementation
```

There are two other magic comment strings that work with this menu:

- //FIXME: adds a bandage icon and the text of what you want to come back and fix.
- //TODO: shows a "to-do list" icon and the text of what you still need to do.

The earlier figure shows an example of a FIXME saying "properties need docs", a TODO to "check access levels", and section headers for segues and the UITableViewDataSource, both created with the //MARK: - syntax.

Documentation Comments

There's much more commenting magic that you can apply at the level of individual symbols, like properties and methods. Like many other modern development environments, Xcode has a documentation comment feature that you can use to document how your code works.

Let's start with a given method, and add documentation comments to it. Here's a bare method signature:

```
func transmute(sources: [Any], recipe: Recipe) -> Any {
}
```

Put your cursor anywhere on the signature line, and use the menu command Editor > Structure > Add Documentation (⌥⌘/). This adds a large comment block above the method:

```
/// Description
///
/// - Parameters:
///   - sources: sources description
///   - recipe: recipe description
/// - Returns: return value description
func transmute(sources: [Any], recipe: Recipe) -> Any {
}
```

The generated documentation comment has placeholder boxes over several of the terms (Description, return value description, etc.), which makes them easier to type over. But before that, let's see what this is actually buying us. Option-click on the transmute method name, and you'll make a documentation pop-over appear, as seen in the following figure:

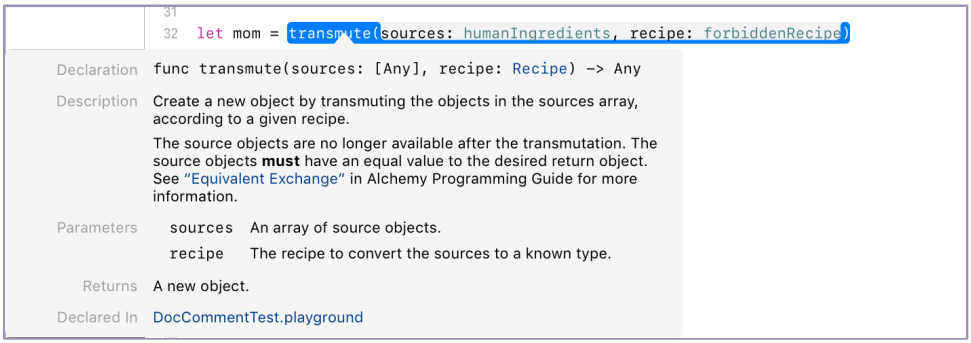
These documentation comments can either be written with three leading slashes (///) on each line, which is the C# standard, or inside /** … */ characters, the familiar javadoc syntax from Java. The syntax inside the comment allows for basic Markdown use, including bold, italics, links, and lists. For parameters and return types, you can either use the syntax shown in the previous listing, where each parameter name is a child of a Parameters list item, or put each on its own line, preceded by the word parameter like this:

```
- parameter sources: An array of source objects.
- parameter recipe: The recipe to convert the sources to a known type.
```

Personally, I prefer the javadoc (/** … */) style, but to each their own. Here's an example of a complete doc comment for the transmute() method, using some Markdown formatting where appropriate:

```
/**
 Create a new object by transmuting the objects in the sources array,
 according to a given recipe.

 The source objects are no longer available after the transmutation.
 The source objects **must** have an equal value to the desired return
 object. See ["Equivalent Exchange"](http://example.com/) in Alchemy
  Programming Guide for more information.

 - parameter sources: An array of source objects.
 - parameter recipe: The recipe to convert the sources to a known type.
 - returns: A new object.
 */
func transmute(sources: [Any], recipe: Recipe) -> Any {
```

That said, once this documentation comment is in place, you hardly need to option-click the method signature itself to see it, since the docs are *right there* above it in a comment. Where this really pays off is when you want to call this method from somewhere else in your app, and maybe you don't remember the parameter conventions or other caveats. When you get in that situation, just option-click the method name at the call point, and you'll see the documentation, just like in the following figure:



There are also third-party documentation tools like Jazzy[2] that can scrape an entire Xcode project or target for documentation comments, and produce formatted documentation in HTML, PDF, and other formats. This is useful if you've developed a framework and need to document your API for others to call.

Finding Documentation and Definitions

It's great to provide documentation for any code you write, but what about the code you use from the iOS frameworks? Of course, this works the same way. Option-click on any symbol—method, function, initializer, property, or what have you—and you'll get the same kind of pop-up documentation.

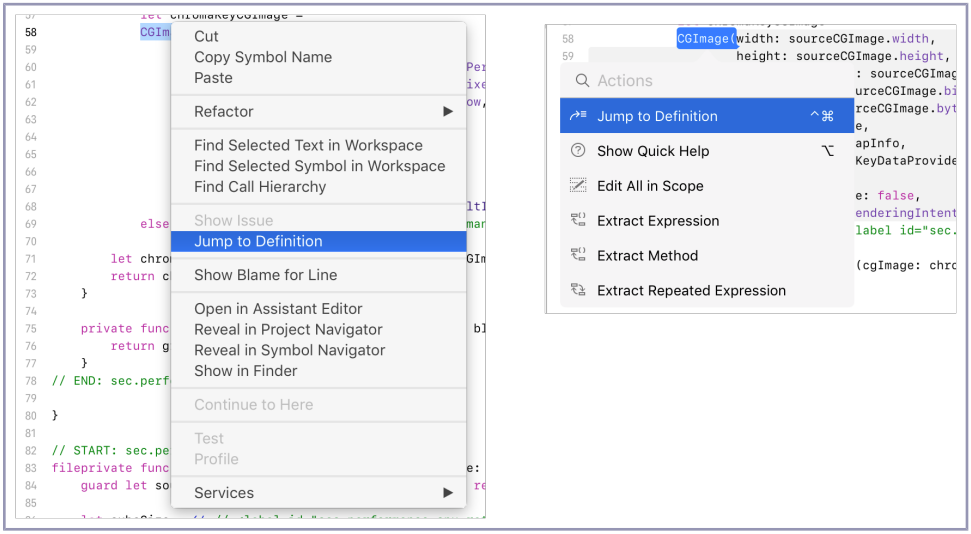
Sometimes, though, that's not enough. Sadly, some things in Apple's frameworks lack adequate documentation, particularly in C-based frameworks like Core Graphics, Core Audio, Core Media… pretty much anything starting with "Core" tends to have lousy docs. Option-click on an undocumented symbol and Xcode will only show you the method signature and a link to its documentation in the Developer Documentation viewer.

Jumping to Definitions

But you don't have to settle for a signature and an empty doc page. There are a number of ways to quickly jump to the file where the symbol is defined.

2. https://github.com/realm/jazzy

Select a symbol and either right-click it, control-click it, or command-click it. The first two options will show a traditional macOS pop-up menu, while the last will show an iOS-like pop-over, as shown in the following figure. Either way, “Jump to Definition” will be one of the items.



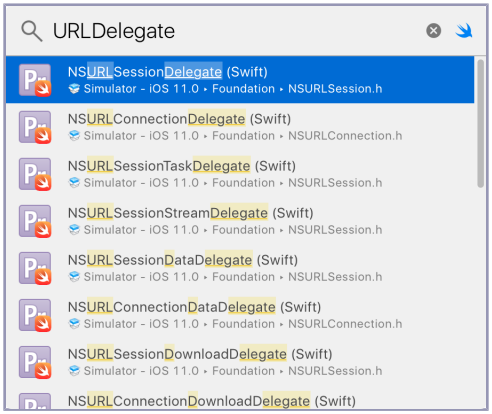
You can also use the menu item Navigation > Jump to Definition, or the keyboard shortcut ^⌘J. And here’s why it’s awesome: there is often a trove of information in the public source files that doesn’t make it to the official documentation. Jump to an arbitrary definition and you’ll usually find comments from the original developers about calling conventions, implementation details, and other great stuff that probably should be in the docs but isn’t.

The file you go to will usually be an Objective-C .h header file. If you’re jumping from a Swift file, the file you navigate to will actually be a programmatically generated Swift pseudo-file, just like the Swift interface files you saw earlier as Counterparts to your own sources. So you’ll get the comments from the Objective-C original, but the syntax will all be Swift.

Finding Definitions with Open Quickly

A related way to navigate to these files is the *Open Quickly* command (⇧⌘O). This brings up a floating text entry box, into which you can type the name of any symbol, either in your source or any framework you’re using, including the iOS APIs. As you type, possible completions appear in a list below the text field. One of the nice things is that the matching is non-contiguous, meaning it will skip over characters or words you’ve forgotten. The following

figure shows a search for URLDelegate, and matches the various delegate protocols in the URL Loading System:



Notice the little Swift icon at the right side of the text field. If this is selected, the file you navigate to will be a programmatically generated Swift file. If not, you'll get the original C or Objective-C header file.

Once you've gotten what you need from the header file, remember that you can navigate back to your own source file with the back button on the Jump Bar (^⌘←).

Typing Source Code

OK, time to get in front of the keyboard and start typing some source. With a good visual theme and (hopefully) a Retina monitor, your environment should be easy on the eyes. Now let's make it easy on our fingers.

Emacs Keybindings

When you're really in the groove, it can disrupt your flow to have to reach for the mouse or trackpad to do something, so it's nice be able to keep your fingers on the keyboard. The various keyboard accelerators for menu commands help with that, but there's a less-obvious helper: like most Mac apps, Xcode text editing supports a small subset of keybindings from the venerable Emacs text editor. I guess because it's always 1985, *somewhere*.

Xcode's Emacs keybinding support is largely limited to common cursor movement keys—no crazy META-X commands or anything like that. Still, this can be a big win if you have one of those keyboards where the arrow keys have been compromised in the name of fussy Apple aesthetics (looking at you, MacBook Pro). The supported keys are shown in the table on page 78.

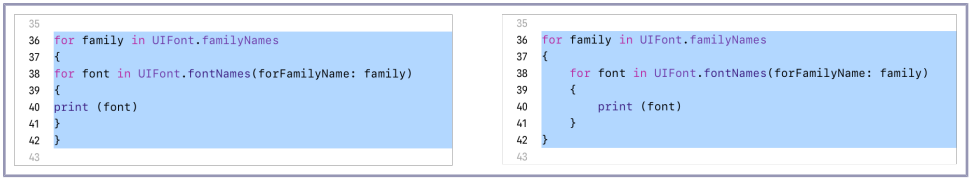
Key	Meaning
^F	Forward one character (→)
^B	Back one character (←)
^P	Previous line (↑)
^N	Next line (↓)
^A	Go to beginning of line
^E	Go to end of line
^D	Delete character forward
^K	Kill to end of line
^Y	Yank from kill ring

The last two entries are weird little Emacs-isms that need explaining. The Emacs *kill ring* is a memory buffer similar to, but separate from, the macOS clipboard. Killing with ^K deletes all characters to the end of the current line, and copies them to the kill ring. Further kills will delete subsequent lines, ending only when you move the cursor. The contents of the kill ring can then be inserted with a yank (^Y). And like pasting from the clipboard, you can repeatedly yank to insert the same kill ring contents over and over. Even if you're not an old Emacs graybeard like me, having a second clipboard available at all times can be enormously helpful for copying from two distinct sources and pasting to a third.

Code Formatting

As you type your code, Xcode will automatically add appropriate indentation once you begin nesting loops or conditionals, or split long lines onto multiple lines. It'll use tabs or spaces based on your project settings (defaulting to your text editing preferences), and is generally pretty smart, but you can manually add spaces or tabs as you see fit. If you want to change indentation for multiple lines, select them and do Editor > Structure > Shift Right (⌘]) or Shift Left (⌘[).

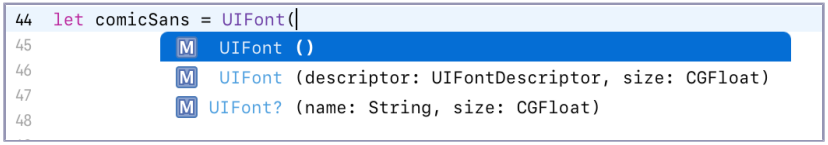
If you want Xcode to fix your formatting problems, just select a range of code and do Editor > Structure > Re-Indent (^I). This will shift lines left or right to achieve a proper balancing of indentation and curly braces. It's fairly smart, too. For example, it understands and respects the difference between "GNU-style" indentation (where an opening curly brace appears on a newline, left-aligned with its loop or conditional statement), and "K&R-style" (where the curly brace appears at the end of the line). The figure on page 79 shows a before-and-after of using Re-Indent, with a GNU-style double-for loop.



One other useful trick in the Editor > Structure menu is Toggle Comments (⌘/). If you want to comment out some code—meaning you want to temporarily turn the code into a comment so it doesn't get compiled or executed—just select a range of lines and do ⌘/. This will also un-comment any code that starts with the double-slash (//) syntax, but only if those lines *start* with the comment syntax; if there's any indentation, it's assumed to be a comment you want to keep, so it actually becomes *more* commented.

Code Completion

Like most IDEs, Xcode offers *code completion*, a small UI that appears to help complete syntactically correct expressions as you work. Unless you disabled it in the preferences, the code completions will appears as a pop-up menu as you type, once you pause in the middle of a line. And if you did disable completions, you can show them manually with Editor > Show Completions (^Space). In the following screenshot, we've paused after momentarily forgetting the initializers for `UIFont`:

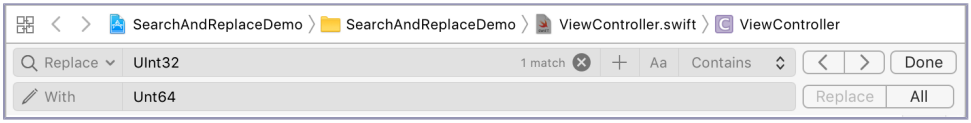


Once the pop-up appears, typing further will dismiss it until you pause again. If you want to use one of its selections, you can select it with the mouse or trackpad, but the fastest way is to just arrow down to the completion you want and press Tab or Enter to accept it. When you do this, names for parameters to methods or functions will be presented as placeholder text; you can tab to each one and type over it with a value for that parameter. Keep the fingers on the keyboard—keep going fast!

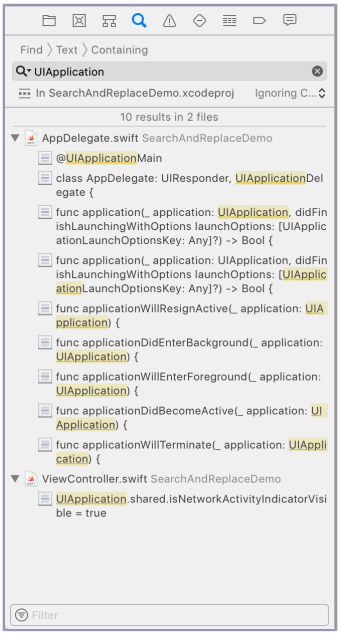
Searching for and Replacing Text

In any kind of text editing—code or otherwise—you often need to find certain strings, or to search for and replace some or all occurrences of a string. And of course, Xcode supports this.

Within a given source file, use Find > Find (⌘F) to bring up a search bar at the top of the content pane, as seen in the following figure. You can use the pop-up menu on the left side to switch between finding and replacing, or look at your recent search terms. There are also buttons for setting case sensitivity and type of match to perform (contains the term, matches a word, etc.). Pretty much the same as any other text editor.



Using the Find Navigator



There's a much more powerful search feature over in the left pane. Use View > Navigators > Show Find Navigator (⌘4) to bring up the *Find Navigator*. Being part of the navigation UI for the whole project, this is where Xcode puts the ability to search multiple files.

At first glance, this just offers the same features of the in-editor search: you can set the UI to find or replace, and set case sensitivity. Results are shown in a list just below the find/replace fields, in a tree view with files as parents to search hits. Click one of the hits and you navigate to that file in the content pane, with a yellow highlight box briefly popping up in the editor to show you where the hit is.

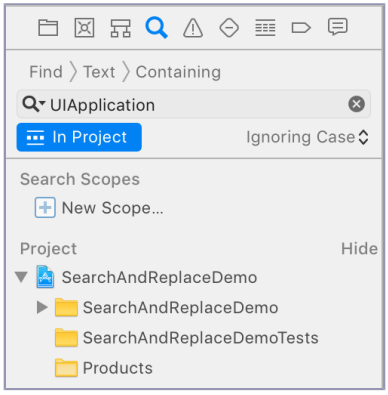
If you're doing a replace, you can arrow up and down through the matches and click "Replace" to replace that instance, or click "Replace All" to bulk-replace all matches in all files.

One of the more powerful features is that you can control what gets searched. At the bottom of the left pane, there's a filter that includes files based on whatever you type into it. This lets you restrict the search to specific file extensions like .swift or .h, or files matching some useful substring, like ViewController.

Doing a Blind Replace-All

With dozens or hundreds of matches, Replace All becomes a potentially dangerous option; you could mess up a file if you haven't inspected whether the search result there is really what you meant. On the other hand, stepping through each match in the list manually can get tedious after a while.

Strangely, Xcode 9 is worse in this respect than its predecessors, at least as of beta 5. In Xcode 8, there was a "Preview" button next to the replace buttons. The preview feature displayed a sheet of search results on one side and the proposed effects of the replacement on the other, with an on/off switch between each pair. This offered a way to selectively edit the replace-all command before invoking it, or cancelling out altogeher. There's no mention in the Xcode 9 docs about the removal of this feature, or whether it will be back.



Another way to control the search is to click the icon under the text fields, which defaults to "In Project". Click this to show a tree structure of your project, from which you can select a target, a group folder, or a combination of sources. This allows you to limit your search to just your app files and not the unit tests, or vice versa. You can also create custom *search scopes*, which match a combination of file metadata, like name, path, extension, whether it's been checked into source control, and more.

Find and Replace with Regular Expressions

There's another powerful feature in the Find Navigator, but you have to know to look for it. The Find/Replace button is a multi-segment pop-up, much like the Scheme Selector. In its simplest form, it says "Find > Text > Containing". You can change the last segment to search for exact words, words that start or end with the search text.

But the real neat stuff is in the middle segment, the one that by default says "Text". Searching for text is not the only feature available. You can also find for when terms are either defined or referenced. The most powerful option is to change the search mode from "Text" to "Regular Expression". This allows you to write a straight-up regular expression as a search term, and to perform a replacement with the matched text.

Here's an example. Let's say you have a C header file, which defines several functions:

```
OSStatus myFirstFunc(int foo, long bar, UInt32 biz);

CFStringRef mySecondFunc(CFStringRef baz);
```

Now, imagine you want to find all C functions in header files, and turn them into Swift equivalents. You can use the filter to limit the search to `.h` files. But from there you need to get clever. A C function starts with a type, then whitespace, a function name, open parentheses, parameters (in the form *name:type*), close parentheses, and a closing semicolon.

Regular expressions are a dark art, and way beyond the scope of this book, but take it on faith that you can match these function signatures with the following pattern:

```
.+? .+?\(.+?\);
```

Each of those `.+?` terms says "non-greedily match one or more characters". So you match the type, skip over a space, match a function name, get an open parenthesis character (notice the `\` escape syntax), match its contents, and end with a closing parenthesis and semicolon. Setting the search type to Regular Expression and using this search string will indeed match the two C function signatures shown here, but none of the Swift functions or methods in the rest of a typical Xcode project.

But how do you do a useful search and replace? With regular expressions, you can surround matchers with parentheses to recover them as match strings. These strings are referred to in the form `$1`, `$2`, and so on, in the order they were matched. So, with the search term

```
(.+?) (.+?)\((.+?)\);
```

you can then use a replacement string like this:

```
func $2($3) -> $1
```

This replacement gives you Swift's `func` keyword, the function name, the parameters, and then the return type. Notice that the numbering of the match

terms allow us to change their order in the replacement: the return type is the first thing in a C function signature, but it comes at the end in Swift.

Click “Replace All” and the function signatures are converted to:

```
func myFirstFunc(int foo, long bar, UInt32 biz) -> OSStatus

func mySecondFunc(CFStringRef baz) -> CFStringRef
```

Now, this isn’t quite right, since the replacement only handled adding the func keyword and moving the return type to the end of the declaration (with its leading ->). The replacement did nothing with the parameters, which would still need to be flipped for Swift (UInt32 biz should be biz: UInt32, for example). Also, some of the types are slightly different in C versus Swift (int versus Int, CFStringRef versus CFString), but those would be easy enough to clean up. And it’s probably still faster than walking through all your header files and manually making this change to every function signature.

Edit All In Scope

Find-and-replace across many files is great, but sometimes you want the exact opposite: replacing a string not even in just a file, but just in *part* of a file. In that case, even find-and-replace within just the file might make changes you don’t want.

Imagine if you use the term “id” a lot, like for playerId and serverId and so on. Now imagine there’s one method where we’ve just used a variable called id, and you want to change it. If we search and replace the string id, we could cause all sorts of unintended havoc by replacing those two characters in all sorts of unrelated symbols.

The solution here is Editor > Edit All In Scope (^⌘E). The idea is to find all occurrences of a given symbol that are in the same lexical scope, and then edit them *en masse*. Consider this code from a hypothetical mech game:

```
func login() {
    var id = PlayerId()
    id.uniqueId = UUID.init().uuidString
    id.teamId = "Default team"
    id.mechId = "Type-00 Takemikazuchi"
}

func logout(id: PlayerId) {
    print ("logout \(id)")
}
```

This works with a PlayerID struct, which has several members that use the term id, so a search-and-replace on id would be a train-wreck. If you want to

change the name of the local variable in login(), even a case-sensitive search will fail us, since it will also match id in the logout() method.

Instead, select any use of the variable id in login(), and do Edit All In Scope. This will put a ring highlight around the instance you've selected and make it editable, and also highlight every other instance within login(), but nowhere else. With each keystroke or other edit in the selection, all the other instances update in real time, as seen in the figure.

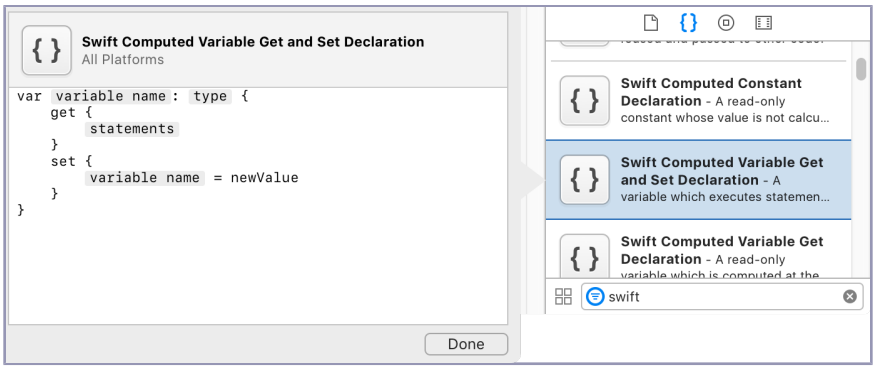
This is exactly what you need: all the instances of the id variable get changed, but not the struct members that happen to have id in their name, nor the use of variable id in the other method. Edit All In Scope is a really cool feature, as long as you remember it's there and waiting for you to use it.

Coding with Snippets

One other handy tool for coding lives over on the right side of the window, in the bottom portion of of the Utilities Pane. In the small toolbar, click the curly-braces icon (or just press ^⌥⌘2) to show the Code Snippet Library. *Code snippets* are small, reusable code templates, meant to reduce boilerplate coding. They're also a big help when you can't quite remember the syntax for something, but have the general gist of how it works.

The provided snippets are arranged alphabetically, and their names start with their language, so you'll see C and Objective-C before you get to Swift snippets. You can also type into the Filter text field at the bottom to narrow things down to just Swift, or a keyword you're looking for. Click any of the snippets to pop up an info window showing a description of the snippet and its code. For example, the figure on page 85 shows the code snippet for a Swift computed variable, with its get and set blocks.

Using a snippet couldn't be easier: just drag it from the Code Snippet Library into your code and drop.



Creating Code Snippets

Of course, the Xcode team can't know which snippets will be useful to any given developer. What's easy for you to remember might be hard for me, or vice versa. Fortunately, you can easily create your own snippets. The process is simple: select and drag some code into the Code Snippet Library, then add some metadata to find it later.

One piece of code I often write is the "Weak-Strong Dance", in which a `weak self` optional is unwrapped inside a closure, so there's a strong instance I can use (or I just exit early from the closure if `self` no longer exists). The reasons for this technique are explained much later, in *How the Strong-Delegate Memory Leak Turns Up in Closures*, on page 146. For now, have a look at the code:

```
foo.methodThatTakesAClosure() { [weak self] parameter1, parameter2, ... in
    guard let strongSelf = self else { return }

}
```

In time, you can learn to type this from memory, but there's a good argument to be made for not having to—especially when you could just make it a snippet. To do so, you just write this in the editor, select it, and drag it to the Code Snippets Library.

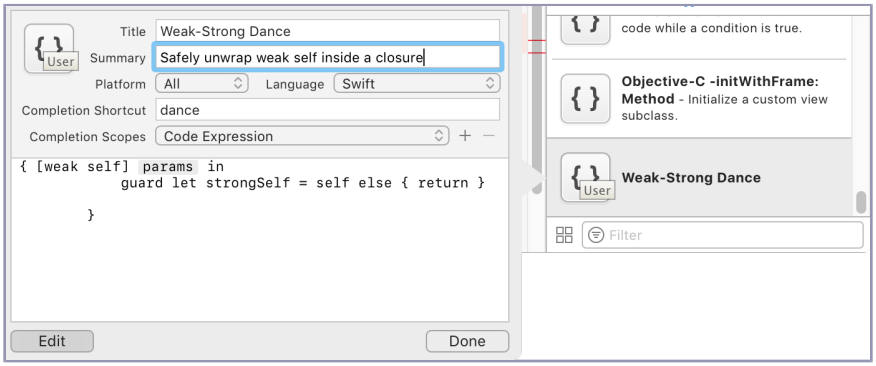
This creates a new snippet called `My Code Snippet`, and immediately pops up an editor that allows you to rename the snippet and enter some metadata. You can set the language appropriate for the snippet, the platforms it can be used on, and more.

Most importantly, you can enter a completion shortcut that will offer the snippet as a code completion. While you can drag and drop a snippet anywhere, code completion will honor any metadata you enter for the snippet's language, supported platforms, and completion scopes. That way, if you declare that a snippet

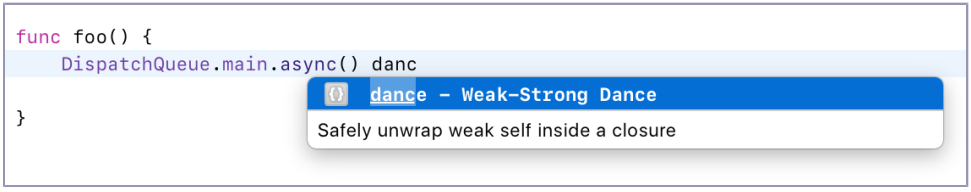
only applies at the top level of a file, it won't be offered as a completion while you're coding inside a method.

One other technique you can use when creating a snippet is that any text in the form <#placeholder#> will become placeholder text that you can tab onto and replace, just like when accepting auto-completion for a method or function call.

The following figure shows the snippet editor with the Weak-Strong Dance snippet filled out (note that while there's no visual indication in the editor, params is written with the previously described placeholder syntax):



Now, while you're coding, you no longer have to remember the fancy closure syntax. As shown in the following figure, all you need to do is start typing the word dance and you'll get the automatic code completion:



Any time you find yourself looking up boilerplate syntax (something you're going to use briefly and then not again for another couple months), you should probably consider whether it's worth a few minutes of your time to make it a code snippet. It could save you time and hassle later.

Wrap-Up

In this chapter, we've tried out all manner of techniques for making your source editing a more pleasant, more productive experience. From the pure aesthetics of fonts, colors, and themes, to various code completion techniques to get your code written correctly and faster, we've hopefully sped things up. In particular, Xcode offers lots of ways to avoid losing focus—whether that's

being able to use a snippet rather than going to the documentation window (or worse yet, opening a browser on Stack Overflow), to using the Jump Bar and other navigation techniques within the editor rather than needing to turn to the File Navigator.

You're going to be spending most of your time in the editor, it might as well be a pleasant experience, right?

Next up, we're going take all this code (and your storyboards and other assets from earlier), and run them through the build system. Just like with the editor, there are lots of things you can control to improve both your developer experience and the apps that Xcode produces.

CHAPTER 5

Building Projects

You've set up your project, created storyboards, and written the code… so the next step is obviously to build and run. Thing is, a lot of developers focus on the "and run" part of that, and remain blissfully unaware of the "build" part. And for sure, it's liberating to go straight from your UI and code directly to how it works on the Simulator or a device. But the reason you can do that is that Xcode makes good default decisions about how to build your code.

Thing is, you may eventually have needs that take you beyond the defaults. Sometimes it's necessary to take the build process into your own hands. And to do that, you have to understand how the build system works, and what's even being built in the first place. That's what this chapter is all about.

Understanding App Bundles

To understand how apps work, it helps to take a step back and think of how you even go from all the files in your project to a running app. For apps on Apple's platforms, executable code and resources are handled very differently, and are brought together in a very clever way.

Compiling Code

In fact, step all the way back to 1978: the era of disco, Cold War tensions, and most importantly, the first edition of *The C Programming Language [KR98]*. Many developers started their careers with this book, which begins with the iconic "Hello, World" program, which looks almost exactly like this:

building/HelloWorldC/HelloWorldC.c

```
#import <stdio.h>

int main ()
{
        printf("hello, world\n");
}
```

This is identical to K&R's listing, except for the addition of int, since main() should have a return type in modern dialects of C.

At any rate, you can type that program into a text file, but it won't do anything until you convert it to machine code. That's the job of the C compiler. On macOS, you can build this program as a command-line executable in Terminal by going to the source directory and typing:

```
⇛ clang HelloWorldC.c
```

clang is LLVM's tool for compiling code in several C-based languages (C, Objective-C, and C++) into an executable file; it replaces gcc in the Apple developer's toolchain. When you execute this command, it creates a file named a.out—yeah, I know, I guess it made sense to someone 50 years ago. You can then execute this file in Terminal by simply typing its file path:

```
⇛ ./a.out
❮ hello, world
```

So, there you go, source to executable from first principles. This example covers not just C, but also C++ and Objective-C, since those languages are supersets of C, and anyway you don't need to introduce objects just to print a string. Swift works the same way, but it's actually easier: you don't need to formally define a main() function. You can just freely write code, like in a Playground. That makes "Hello, World" in Swift literally a single line of code:

building/HelloWorldSwift/HelloWorldSwift.swift

```
print ("hello, world")
```

For Swift, the command-line compiler is swiftc. It has the sense to use a better default name for the executable than a.out; for a single file, it will just remove the .swift file extension from the source file. That means the Swift build-and-run cycle looks like this:

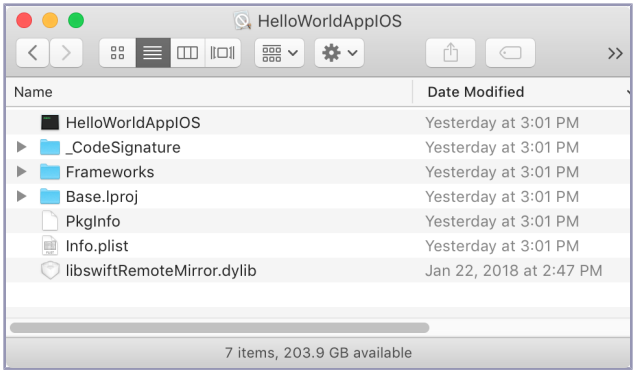
```
⇛ swiftc HelloWorldSwift.swift
⇛ ./HelloWorldSwift
❮ hello, world
```

Now, if you can imagine these two compilers running against many source files—clang handling the .c, .cpp, .m and other C/C++/Obj-C files, and swiftc taking care of the .swift files—you can intuit how you get from the collection of source files in your Xcode project to an executable that runs on the iPhone, iPad, Mac, Apple TV, or Apple Watch.

Inside App Bundles

However, on the Apple platforms, the executable code is only part of the story. Apps are distributed as *bundles*, directories with a defined structure of contents. While the Finder treats them as flat files, bundles are actually directories, and can be explored either with the contextual menu item "Show Package Contents" in the Finder, or by just doing a `cd` in Terminal to change the working directory to be the top-level directory of the bundle.

For example, if you create a trivial "Hello, World" iOS app that just consists of a one-scene storyboard with a label that says "Hello, World", the contents of the bundle will look like the following figure:



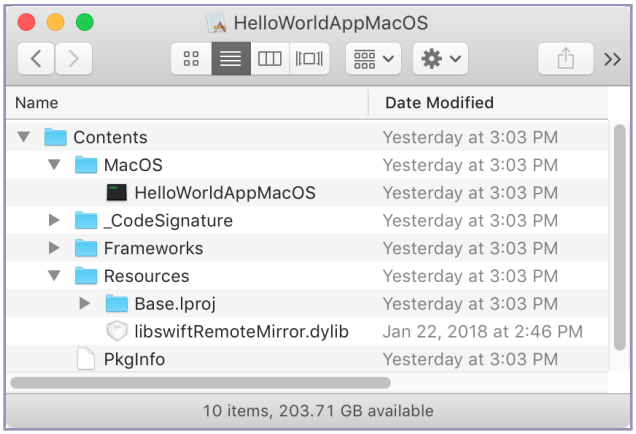
The first file in the figure, `HelloWorldAppiOS`, is an actual executable, as indicated by its Terminal-like icon. This file is the result of running `swiftc` on all the source code (and `clang` if you have C/C++/Obj-C source files). Everything else in the bundle is something the executable code makes use of:

- `_CodeSignature` contains a `_CodeResources` file used to validate the authenticity of the bundle's contents.
- `Frameworks` contains library code for third-party frameworks used by the code, and (for Swift apps) system frameworks.
- `Base.lproj` contains the base version of storyboards and other localizable resources. If you have localized versions of any of these resources, they'll be in separate folders named for the locale, like `fr.lproj` for French or `jp.lproj` for Japanese.
- `PkgInfo` just has the four-character code for file type (always `APPL` for applications) and signature (unique to your app, defaulting to `????`) for use only on macOS.

- Info.plist is metadata about the app. While it has the same name as the Info.plist in your Xcode project, this is a completely different file, and contains values of many build settings that will be discussed later.
- libswiftRemoteMirror.dylib is a support file that is only present in debug builds of your app.

Other files will be present based on what you've included with your app. For example, once you have any contents in your project's assets library, the app bundle will have an Assets.car containing the assets.

The bundles created for apps on the other Apple platforms use the same basic approach, with slight differences in their contents. The one that's kind of interesting is macOS, since you can access the bundle contents in ways not possible on iOS, tvOS, or watchOS. For starters, build a Mac app—you can use HelloWorldAppMacOS from the sample code download if you like—and look inside its bundle, as seen in the following figure:



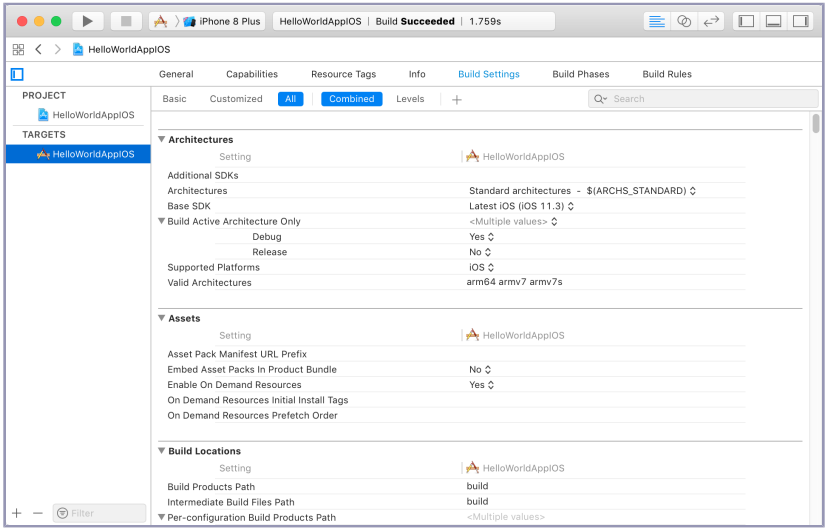
There are slight differences here, such as the presence of a Resources folder, and the executable being in a MacOS folder. Now here's the crazy part: you can use the command line to run the HelloWorldAppMacOS executable. In Xcode, find the app in the "Products" group, and right-click to do "Show In Finder", and in the Finder, do "Show Package Contents". Then type the full path to the executable directly into the command line; to save time, you can drag the executable's file icon to the Terminal window to fill out the path. When you do this, *the app launches and runs.* You'll see it bounce up as a new icon on the Dock, and you can use it like any other Mac app. Terminal will actually block on the app until it quits, since it's running as a child process of your command-line shell.

You might recognize the usefulness of this technique if you've ever used Mac apps that provide their own command-line launchers, such as the text editors BBEdit and TextMate. In fact, your app can include multiple command-line tools inside the `MacOS` bundle folder, other than just the app's main executable. How you actually copy files like that into your bundle is something you'll see later, in *Copy File Phases*, on page 96.

Build Settings

Now that you know how an app bundle works, the next thing to think about is how files get there. How is it that doing a build knows to put everything in a bundle like this, and what can you do to customize that process? The answer is in the target settings, under two tabs: *Build Settings* and *Build Phases*.

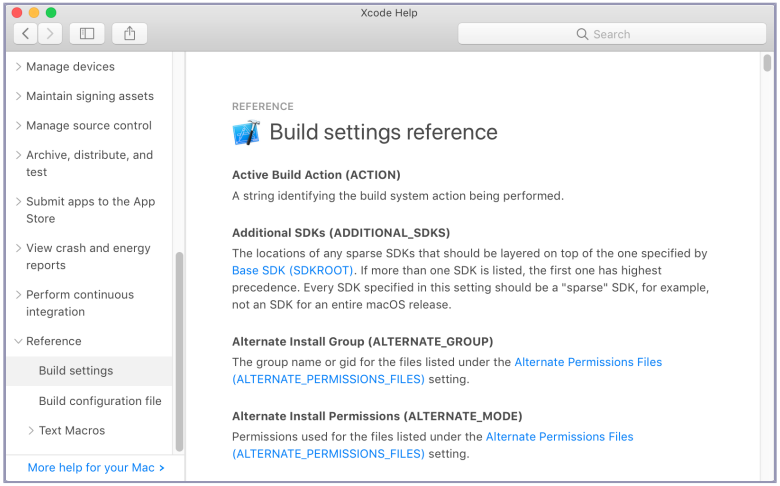
You saw the Build Settings tab back in *Working with Project and Target Settings*, on page 4, where you used it to set things like the version of your app, the OS version to set as a deployment target, and so on. That chapter also showed how to create a user-defined setting on this tab for `BACKEND_URL`, so that the developer, beta tester, and deployment versions of the app could all work with separate server URLs.



Now take a look at the rest of the build settings. As seen in the figure, it's a huge list, split into sections. The first few cover overall traits like the architecture you're building for and where the build products go, then later you find detailed sections for the Apple LLVM Compiler (which compiles your C, C++, and Objective-C code) and the Swift compiler. Keep in mind that each setting can be customized on a per-configuration basis, as indicated by the disclosure

indicator triangle. Expand any setting to show its value for the two default configurations: Debug and Release. Also keep in mind that any setting that has been changed from its default is shown in bold.

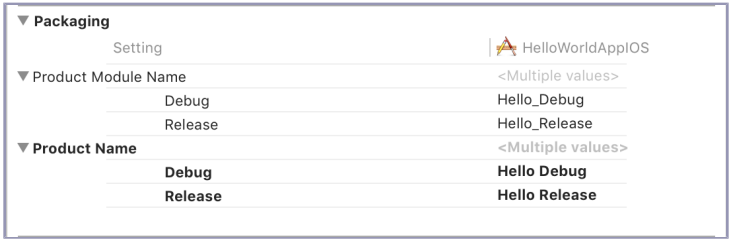
Problem is, the setting names don't always give you the full story of what they do. Fortunately, there's more documentation under Help -> Xcode Help. This shows Xcode's own documentation (as opposed to the SDK documentation shown by the Developer Documentation menu item) and the last section, "Reference" contains detailed information about every setting available in Xcode.



For example, one setting some developers like to change is to treat compiler warnings as errors, meaning you can't build your code without fixing all your warnings. You'll have to set this for each compiler: under "Apple LLVM 9.0 - Warning Policies" for your C-based code, and "Swift Compiler - Warnings Policies" for Swift. For the C languages, you can also enable or disable specific warnings, or turn on "Pedantic Warnings" if you really want to suffer. Keep in mind that not every warning is useful. For example, there's a warning for using four-character literals in C code, off by default because it can be a useful technique with some older and low-level APIs, like Core Audio, where you search for components like Audio Units with four-character codes.

Also notice that the Build Settings Reference shows every setting with a name in parentheses, all caps and with underscores between words. These names can be referenced during the build itself. For example, your Info.plist file uses these settings names to fill in some of its values. "Bundle Name", which provides the app icon's name on iOS, defaults to the value $(PRODUCT_NAME), meaning it uses the syntax $() to expand the value of PRODUCT_NAME. The product name is one of your build settings (in the Packaging section) and actually defaults to

the value $(TARGET_NAME). If you reset it in the build settings, that'll change the value in the Info.plist. As you can see in the following figure, one useful thing to do with this is to have different app names for your debug and release builds:

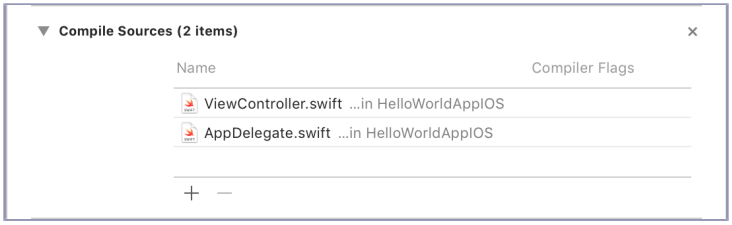


Build Phases

Build Settings are great and all, but they don't actually *do* anything in and of themselves. The next tab over, Build Phases, is where the action takes place. This tab describes each step of the build process in order.

Depending on what type of project you created, several phases will already be present. For an iOS or Mac app, there are default build phases to do the following:

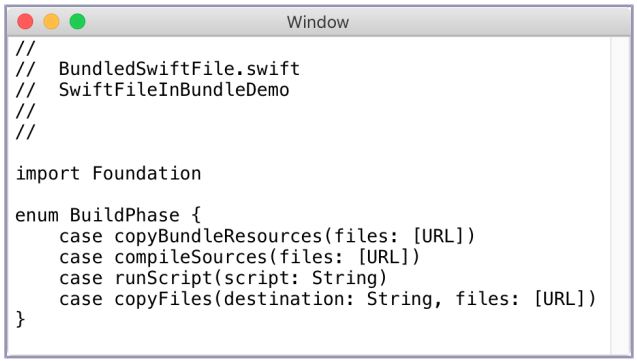
- Build any dependencies first, which are other targets in the project that need to be built before this one.
- Compile your source files into executable code.
- Link your executables with system libraries.
- Copy resource files, like storyboards, asset catalogs, other kinds of media, and so on.



You can expand a phase with the disclosure triangle on the left. Each phase contains a list of files that the phase applies to, along with + and - buttons to add and remove files from the phase. So, for the Compile Sources phase, all .swift, .c, .m (Objective-C) and other source files are automatically added to the compile phase as you add them to the project. Same goes for resource files in the Copy Bundle Resources phase.

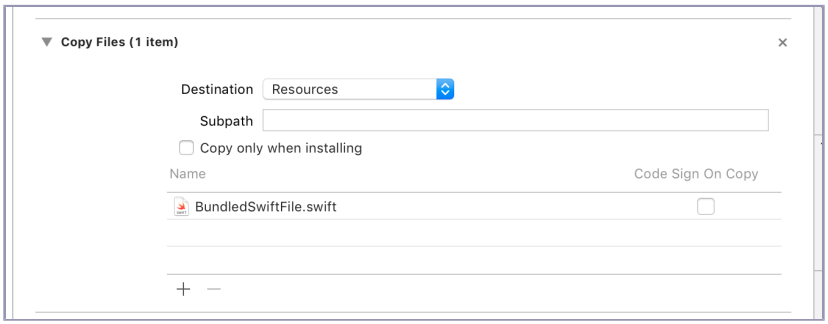
Copy File Phases

In rare occasions, you might want to tweak this. Imagine, for example, if you had a programming-tutorial app that needed to bundle .swift files for the user to view and edit. You wouldn't want to build these files, but instead copy them to the app-bundle as-is. In the following figure, BundledSwiftFile.swift is meant to be shown in a window, and not to actually be built as part of the app:



To do this, you need to do two things. First, go to the Compile Sources build phase and use the minus button to remove BundledSwiftFile.swift from the set of files to compile. Next, since the file isn't of a resource type (storyboards, asset catalogs, images, etc.), you can't just add it to the files in the Copy Bundle Resources rule. Instead, click the + at the top of the Build Phases tab and choose "New Copy Files Phase" (or use the menu item Editor -> New Build Phase -> Add Copy Files Build Phase).

This copy-files phase is different from the default Copy Bundle Resources phase, because it lets you specify a known destination within the app bundle, along with an optional subpath if you want to get fancy with your file structure. To read the file at runtime, the best place is the Resources folder, since that's in the search path used by the Bundle class' url(forResource:withExtension:) and similar resource-loading methods.



With your file copied to this expected location, reading it at runtime is easy:

building/SwiftFileInBundleDemo/SwiftFileInBundleDemo/ViewController.swift

```
guard let sourceURL = Bundle.main.url(
    forResource: "BundledSwiftFile", withExtension: "swift"),
    let sourceText = try? String(contentsOf: sourceURL) else { return }
textView.string = sourceText
```

Build Rules

To the right of the Build Phases tab, there's one more tab that you'll probably never need to use, but which answers an important question: *how does Xcode know what to do with each file type?* The *Build Rules* tab is a list of file types and actions to take on them. You can search through the list to find the rule that manages the existing types. You can also use the + button to define a new rule, which matches files either by known types or by a substring you choose (like a file extension), and can then run one of several dozen built-in actions, or run an arbitrary script.

This lets you add pretty much any kind of processing to an Xcode build. For example, if you had a compiler for some arbitrary language that produced code in an appropriate binary format (x86_64 for Mac, ARM for iOS devices), you could add a rule to run a script to call that compiler, and then write app sources in that language. And that's great news for anyone with legacy FORTRAN code from 1965 that they want to embed in an iPhone app, right?

Run Script Phases

Along with tweaking the files processed by the built-in rules, and having the ability to copy files to the target, there's one more option to really open up what you can do with builds: the Run Script phase. This allows you to write a shell script that can do, well, anything a shell script on your Mac can do. For example, the SwiftLint[1] code checker uses custom scripts to scan your source code and enforce good Swift coding style.

As an example, I once hid an Easter egg in one of my apps that would show what I was playing in iTunes at the time the build was performed. You can get the current song title with a three-line AppleScript:

```
tell application "iTunes"
    get the name of the current track
end tell
```

Next, with the command-line utility `osascript`, you can run AppleScripts by either separating each line with the `-e` flag, or providing a source file as an

1. https://github.com/realm/SwiftLint

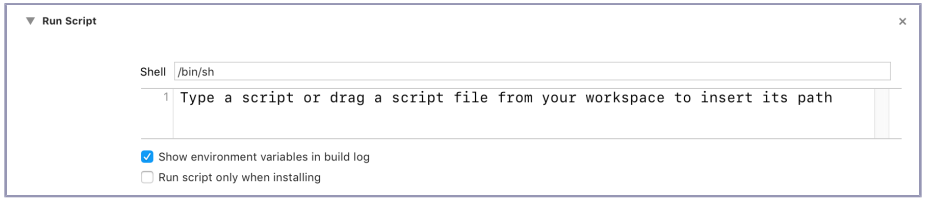
argument. So you can get the iTunes current song title written to standard out like this (note that this listing uses the \ line-wrap operator to split the command over several lines to fit the book's formatting; you can write it all as one line):

```
⇛ osascript -e "tell application \"iTunes\"" \
⇛ -e "get the name of the current track" \
⇛ -e "end tell"
❮ Gonna Needa Pasteboard
```

So, you can probably imagine that with a series of osascript calls, you can extract whatever you need from iTunes. Now you'd need a way to get them into a file that could then be copied into the build. You could write out a simple text file, but for build scripts, you can make use of a wonderful command-line utility called PListBuddy. This executable, which lives in /usr/libexec, can read and write individual entries from plist files, which in turn can be easily read into memory as NSArray and NSDictionary objects.

PListBuddy's commands can by shown with its -h command. For this demo, all you need to know is that you can say PListBuddy -c "Add *key-name value-type value file-name*" to provide the value of a key in a given .plist file, which will be created automatically if it doesn't already exist.

With these two tools, you have everything you need to write a script to set up the needed file inside the bundle. Start by clicking the + button and choose "New Run Script Phase".



There's one other thing you need to know for this script to work: where to write the .plist file. Fortunately, all the build settings described earlier are available in scripts, so $TARGET_BUILD_DIR contains the path to the directory where the app is being built. And that means you can write a file inside the bundle by using this path and appending the app name. Then you need to know where to put a file inside the app bundle so the Bundle class can find it at run-time. On iOS, just put your file in the top-level directory, and on macOS, put it in Contents/Resources.

So here's a script to create the Easter egg file (like before, this has to use bash's line-wrap syntax to fit the formatting of the book):

```
tmpfile=$(mktemp /tmp/tunes.txt)
rm $TARGET_BUILD_DIR/BuildScriptEasterEggDemo.app/BuildTunes.plist > \
/dev/null 2>&1
osascript -e "tell application \"iTunes\"" \
-e "get the name of the current track" \
-e "end tell" > $tmpfile
/usr/libexec/PListBuddy -c "Add :SongTitle string $(cat $tmpfile)" \
$TARGET_BUILD_DIR/BuildScriptEasterEggDemo.app/BuildTunes.plist
osascript -e "tell application \"iTunes\"" \
-e "get the artist of the current track" -e \
"end tell" > $tmpfile
/usr/libexec/PListBuddy -c "Add :SongArtist string $(cat $tmpfile)" \
$TARGET_BUILD_DIR/BuildScriptEasterEggDemo.app/BuildTunes.plist
rm "$tmpfile"
```

This script starts by creating a temporary file descriptor and deletes any BuildTunes.plist file left over from an earlier run (writing any output or errors to /dev/null so Xcode doesn't see them as errors and stop the build). Then it does a call to osascript that writes the song title to the temp file, and a call to PListBuddy to write the temp value to the BuildTunes.plist file. Next, it repeats these steps with the song artist. Finally, it deletes the temp file.

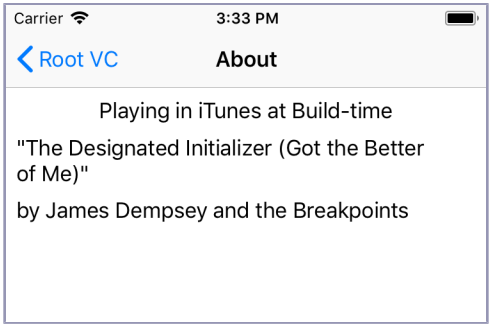
Try a build, look in the package contents of the app file, and you'll see the BuildTunes.plist file. Now all you need to do is read it at runtime:

building/BuildScriptEasterEggDemo/BuildScriptEasterEggDemo/AboutViewController.swift

```
guard let songInfoURL = Bundle.main.url(forResource: "BuildTunes",
                                        withExtension: "plist"),
    let songNSArray = NSDictionary(contentsOf: songInfoURL),
    let title = songNSArray["SongTitle"] as? String,
    let artist = songNSArray["SongArtist"] as? String else {
        titleLabel.text = "Didn't find"
        artistLabel.text = "Didn't find"
        return
}
titleLabel.text = "\"\(title)\""
artistLabel.text = "by \(artist)"
```

And that's all it takes. Just like the custom-copy-phase Swift file in the previous section, the newly created .plist file is waiting for you to find and use at runtime, as shown in the figure on page 100.

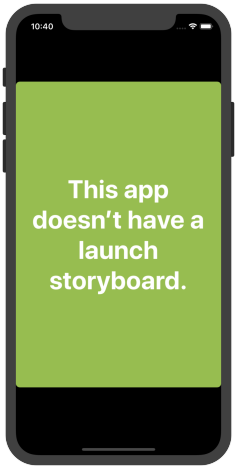
Granted, this is a silly exercise, but it should drive home the point that anything you can do on the command line—which pretty much means *anything*—can be part of your build process, thanks to the run script build phase. All you need are some mad bash skills, and all those Xcode build variables mentioned earlier.



Special Files in Builds

There are a handful of special files and file types whose very presence affects the build or behavior of the resulting application. One example is entitlements files, which grant permission for potentially insecure actions, and which will be covered further in Chapter 9, *Security*, on page 171.

The Launch Screen Storyboard



By default, iOS app projects have a LaunchScreen.storyboard file. This is used when animating the launch from the app's icon into the first screen of the app. It also has a stealth purpose: if it's absent, iOS assumes you haven't accounted for different device sizes, so your app is shown with black bars at the top and bottom of the screen on iPhones bigger than the original 3.5-inch iPhone, as seen in the figure.

What iOS actually looks for is an entry in the Info.plist called "Launch screen interface file base name", which defaults to LaunchScreen; if you wanted to rename the storyboard file, you'd need to change this entry too.

At any rate, you want to keep the launch storyboard as simple as possible: you can use Auto Layout constraints to adapt its contents to different sizes or orientations (size classes really help here), but you shouldn't back it with a custom view controller and try to do work at launch-time, because you'll just slow the rest of your app down. Often, a single UIImageView, possibly with size-class-specific images, is the way to go. Apple has further advice for launch screens in their Human Interface Guidelines.[2]

2. https://developer.apple.com/ios/human-interface-guidelines/icons-and-images/launch-screen/

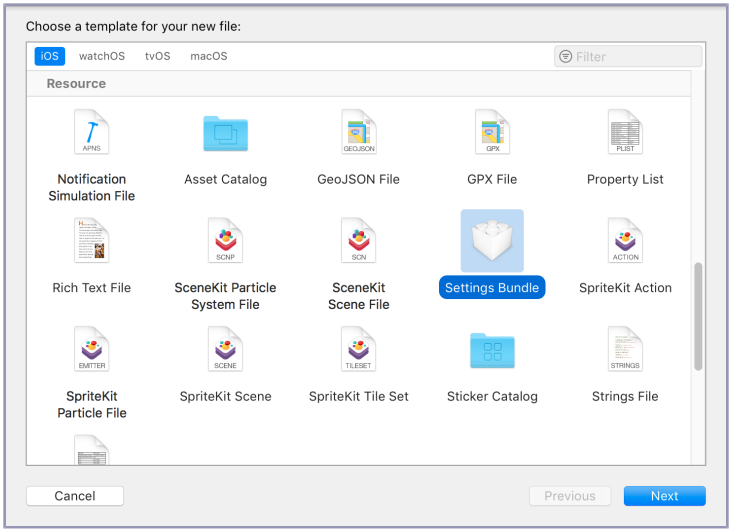
Launch Screen Madness

The launch storyboard replaces earlier schemes where you'd have to include PNG files of every screen size and orientation you wanted to have a launch screen for. With the proliferation of device sizes over the years, that became impractical. Another change is that Apple originally suggested you replicate the first screen of your app with a static launch image, to conceal how slow app-launching was on the original iPhone. Now that iPhones are faster than many desktop computers, that's no longer a concern.

Settings Bundles

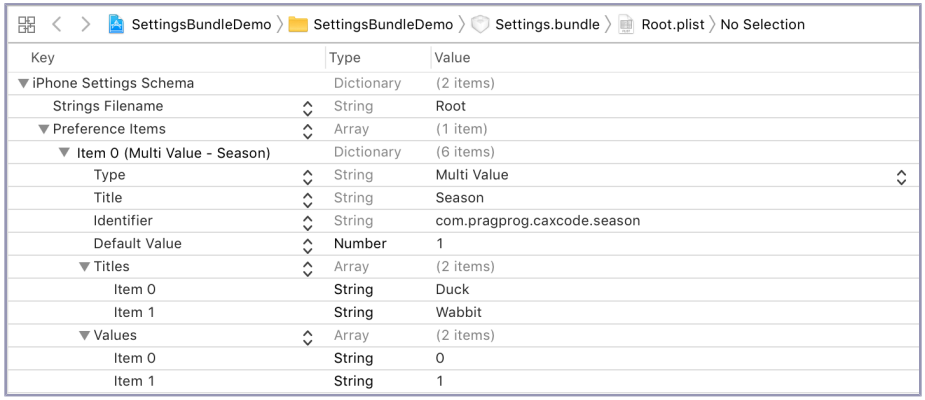
On iOS, Settings.bundle is another file that causes interesting side effects just by being present. This file adds an entry for your app in the main Settings app. Apple used to recommend that all apps put their settings here, but that guidance has been relaxed and now apps tend to put frequently changed settings in the app itself, and infrequently changed stuff in the Settings app.

To add an entry to the Settings app, use File->New->File… to add a file to the top level of your app target, and choose the "Settings bundle" bundle template, as seen in the following figure (the filename will default to Settings.bundle, and you should not change it):

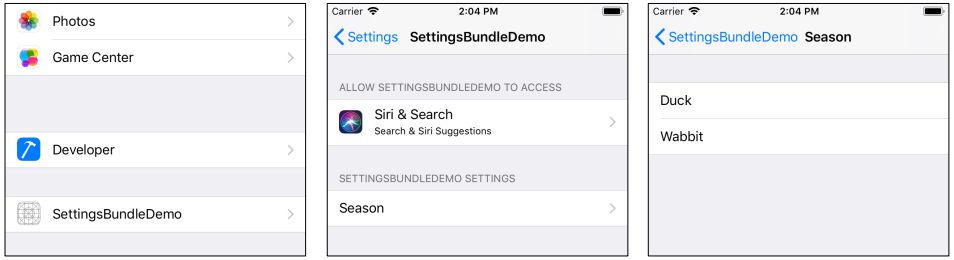


The settings bundle is a folder containing localization folders and a Root.plist property list file with a default group of settings specifiers for the first page in the Settings app. The default settings show a few simple examples for things like buttons and text entry. There are more component types available,

as well as the ability to create several pages of settings by using multiple .plist files — see Apple's "Preferences and Settings Programming Guide" for details and syntax of every option. For a simple example, consider the following figure, which offers a choice between two mutually exclusive options for cartoon hunting season—Duck and Wabbit:



By just including this in the app bundle, the SettingsBundleDemo app gets an entry in the Settings app, as shown in the three successive drill-down screenshots shown here:



The choices the user makes in the Settings app are communicated via the UserDefaults class (NSUserDefaults in Objective-C). In this example, the value uses the key com.pragprog.caxcode.season.

However, the default value won't be picked up from the settings bundle alone. For that, you need to register a set of defaults when the app starts up:

building/SettingsBundleDemo/SettingsBundleDemo/AppDelegate.swift

```
UserDefaults.standard.register(defaults:
  ["com.pragprog.caxcode.season" : Int(1)]
)
```

Now, whether the preference has been set with these initial defaults or the Settings app, you can read the value from UserDefaults:

building/SettingsBundleDemo/SettingsBundleDemo/ViewController.swift
```
let defaultSeasonValue = UserDefaults.standard.integer(
  forKey: "com.pragprog.caxcode.season")
```

If your app has its own UI for setting this preference, then just set it on UserDefaults to update the stored value:

building/SettingsBundleDemo/SettingsBundleDemo/ViewController.swift
```
UserDefaults.standard.set(Int(season.rawValue),
                          forKey: "com.pragprog.caxcode.season")
```

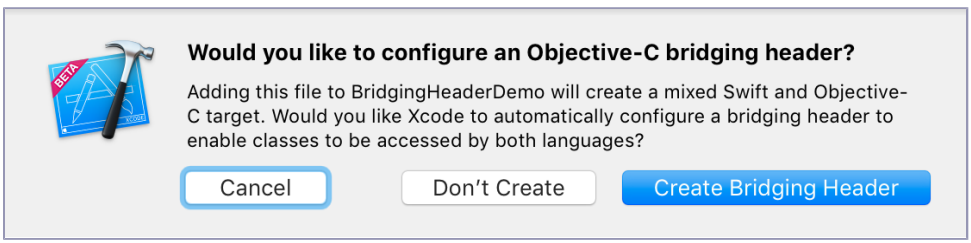
The other thing to watch for is the possibility of the user switching out of your app and changing a default in the Settings app, while your app is still running. Fortunately, in this case, there's a Notification you can observe:

building/SettingsBundleDemo/SettingsBundleDemo/ViewController.swift
```
notificationObserver =
  NotificationCenter.default.addObserver(
    forName: UserDefaults.didChangeNotification,
    object: nil,
    queue: nil) { notification in
      let season = self.fetchDefaultSeason()
      self.updateSeasonLabel(season)
}
```

Depending on other system features enabled by your app, the Settings app will be the place the user goes to enable or disable things like push notifications or microphone and camera access. They'll see your settings while they're there, so using a setting bundle is sometimes a good companion or even alternative to building an in-app preferences UI.

Bridging Headers

One other important type of file to know about comes up when you're building mixed-language projects. The *bridging header* is a file that exposes C functions and Objective-C classes and methods to Swift. Xcode offers to create this file for you the first time you add a file in one of these languages to a Swift project, as seen in the following figure:



In the bridging header file that's created, do a #import for every .h file you want to expose to Swift. You can also be more granular with C functions and just list those functions by themselves.

For example, consider a C function that simply calls the old rand() function, which isn't available in Swift because it's old and bad and has been replaced by arc4random():

building/BridgingHeaderDemo/BridgingHeaderDemo/CStuff/CUtilities.c

```
int oldRand(void) {
    return rand();
}
```

In the bridging header file, you just need to import the C header file that declares the oldRand() function:

building/BridgingHeaderDemo/BridgingHeaderDemo/CStuff/BridgingHeaderDemo-Bridging-Header.h

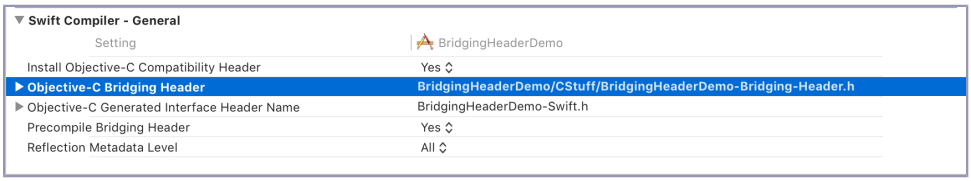
```
#import "CUtilities.h"
```

And with that, you can call your C or Obj-C code directly from Swift:

building/BridgingHeaderDemo/BridgingHeaderDemo/AppDelegate.swift

```
print ("Calling old C rand() function: \(oldRand())")
```

The bridging header isn't a magical file: it's one of your Build Settings, in the "Swift Compiler Settings - General" section, as shown in the figure. Keep this in mind, because if you ever move or rename the bridging header file (or a group that contains it), your build will break until you re-point this setting to the new path.



Calling C++ from Swift

A bridging header won't let you call C++ directly from Swift. However, you *can* call C++ from C or Objective-C, so the solution is to just make a wrapper for your C++ code in one of those languages, and expose the wrapper in the bridging header.

Building on the Command Line

It turns out you don't have to be running Xcode to build an Xcode project. The command-line tools installed with Xcode include xcodebuild, which lets you perform build actions on the command line or as part of a shell script.

This style of build is particularly important when you're doing *continuous integration*, in which your codebase is being automatically built on a regular basis. One common technique for this is to set up Jenkins,[3] which can scan for commits to a source control system like Git (to be introduced later, in Chapter 10, *Source Control Management*, on page 189), check out the project, build it, and then package it for distribution or send emails if the build failed. For this to work, the CI script has to have the ability to perform builds programmatically, and that's what xcodebuild provides.

xcodebuild has a lot of options you can pass as arguments, which are shown on its manual page, by entering man xcodebuild on the command line. To do a build, you call either xcodebuild -project Project-Name.xcodeproj or xcodebuild -workspace Project-Name.xcworkspace, depending on whether you're working with a regular Xcode project or a workspace. You can also use -target to choose specific targets, -scheme to pick a scheme, -configuration for configurations, -sdk to choose between an iOS device or simulator, and several other options.

In fact, sometimes the first step to using xcodebuild is figuring out how to call it. The -list command inspects a project or workspace, and shows you its configurations, targets, custom schemes, and more. For example, the UpcomingConferences project back in *Creating App Extensions and Frameworks*, on page 18 had an app, an app extension, and a supporting framework, so it provides some interesting output. cd to that directory and use the -list command to inspect it:

```
⇛ xcodebuild -list -project UpcomingConferences.xcodeproj/
❮ Information about project "UpcomingConferences":
    Targets:
        UpcomingConferences
        UpcomingConferencesTests
        UpcomingConferencesUITests
        UpcomingConferencesToday
        UpcomingConferencesFramework
        UpcomingConferencesFrameworkTests

    Build Configurations:
        Debug
        Release

    If no build configuration is specified and -scheme is not passed then
         "Release" is used.

    This project contains no schemes.
```

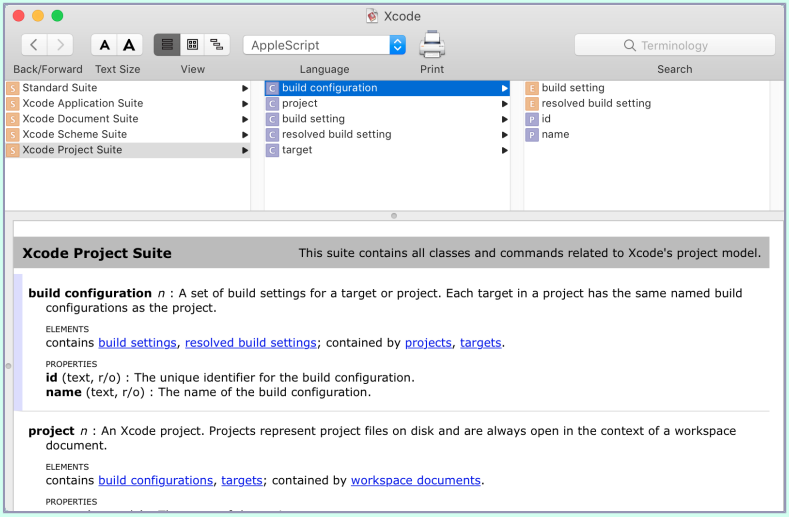
So, as you might suspect, you could build just the framework with xcodebuild -target UpcomingConferencesFramework -project UpcomingConferences.xcodeproj, or the whole

3. https://jenkins.io

app with `xcodebuild -target UpcomingConferences -project UpcomingConferences.xcodeproj` (since the app builds the framework as a dependency). Actually, you don't even need to specify the app as the target, since it's the default. You could do a debug build by appending `-configuration Debug`, or even build for the simulator with `-sdk iphonesimulator11.2` (or the version of whatever Simulator is current at the time you read this), although just building for the Simulator and not actually immediately running it in the Simulator may not be particularly useful, which is why the default is to do a `Release`-configuration build for the device.

Building with AppleScript

If Automator and AppleScript are more your speed, you'll be pleased to know that Xcode supports both of them. Automator comes with a "Build Xcode Project" action that you can make part of a workflow. Meanwhile, the AppleScript dictionary for Xcode offers a deep look into a project or workspace and its build settings, along with the ability to build and run from an AppleScript.



Wrap-Up

In a way, this chapter has gone backwards, or to be more purposeful, started with the end in mind. By understanding what an app bundle looks like, you can have a better sense of what the end product of a build looks like. With that in mind, you've seen how build settings are used by the various build phases like compiling sources and copying resources, and how you can work your own stuff into this process with custom phases to copy files or run

arbitrary scripts. And finally, you saw how the presence of special individual files can affect the build or the resulting app, like launch storyboards and settings bundles.

With that in mind, you can tailor a build process to produce exactly the app you want… until things inevitably go wrong. After all, it's all for naught if your code is riddled with bugs. In the next chapter, you'll see how Xcode provides tools to find and fix bugs.

CHAPTER 6

Debugging Code

It feels great when your code builds without errors, but sometimes that means your work is just getting started. Even if your app runs, chances are it has bugs: logic that does the wrong thing when it's actually run, UI elements that don't look right, or worst of all, a full-blown crash of your app.

When you have bugs, it's time to start debugging.

But… how? If you wrote the code wrong the first time, how do you figure out how to make it right? Is program execution not reaching some important part of the code, or is it making bad decisions once it does? Many developers will hop into the troubled part of the code and add logging statements (`print()` in Swift, `NSLog()` in Objective-C, or `printf()` in C) to get some idea in the Xcode console of what's going on. This inevitably takes multiple build-and-run cycles as you narrow down what you even need to log, figure out what the problem is, fix it, and then clean up all the logging statements.

This process can be a lot more elegant and productive than that. Xcode has deep and powerful debugging features, courtesy of the *Low Level Debugger*. `lldb`, as we'll call it from here on out, gives you powerful abilities to inspect the program flow and state of your app, catch your app in mid-crash, and even let you *change program state at runtime*. Once you know `lldb`, you may never want to use logging statements for debugging again (which means you can leave them to their true purpose: logging problems experienced by users in the field).

This chapter focuses on logic, appearance, and crashing bugs. Once those are dealt with, the next chapter will move on to finding and fixing performance bugs.

Finding Bugs with Breakpoints

The first step to using the debugger is to get into the mindset of using *breakpoints*. A breakpoint is a location in your code where the debugger will pause

the app's execution and let you inspect the app's state. This gives you an opportunity to figure out what's going on.

Consider the following function to calculate prime numbers up to a certain maximum, and return them as an array. It sets aside some trivial cases upfront with a guard and a switch, since negative numbers, 0, and 1 are not considered prime, and 2 is too simple a case to bother iterating over. In the default case, it takes each candidate value and looks for integers that divide evenly into it:

debugging/PrimeCounterDemo/PrimeCounterDemo/ViewController.swift

```
func calculatePrimes(to max: Int) -> [Int] {
  guard max > 0 else { return [] }
  switch max {
  case 1:
    return []
  case 2:
    return [2]
  default:
    var primes: [Int] = [2]
    for candidate in 3...max {
      var isPrime = true
      for divisor in 1...(candidate/2)+1 {
        if candidate.remainderReportingOverflow(dividingBy: divisor) ==
          (0, false) {
          isPrime = false
          break
        }
      }
      if isPrime {
        primes.append(candidate)
      }
    }
    return primes
  }
}
```

When this function runs by calling calculatePrimes(to: 1000), the return value is [2].

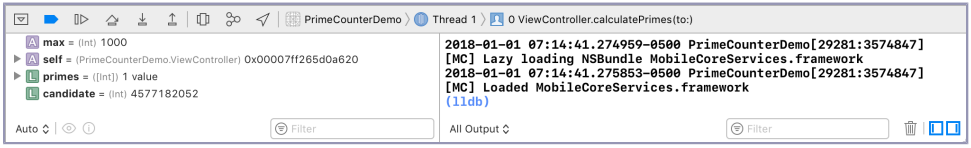
Um, *no*. That's missing dozens of valid primes. Let's try that again.

Setting Breakpoints

Maybe you see the bug, but let's assume you don't. In that case, you can use the debugger to figure out what's going on. Next to the line for candidate in 3...max, click in the gutter, which is the vertical strip along the left side of the text editor (and where the line numbers are, if you have them turned on). This creates a blue arrow pointing to your code, representing the breakpoint as shown in the figure on page 111.

```
44          for candidate in 3...max {
```

Now, when you run the app again, it will start up as usual, but when execution reaches the breakpoint, the app will halt and Xcode will switch to the foreground. By default, the project window will show the *Debug area* at the bottom of the window. If it doesn't (possibly because of settings which will be covered in *Debug-Time Behaviors*, on page 120), you can show it by navigating to View > Show Debug Area (⇧⌘Y), or clicking the Show Debug Area button in the group of three buttons on the far right of the toolbar.



As seen in the figure, the debug area is split into two panes. On the left, the *Variables View* shows the name and value of every variable in scope, as an expandable tree view. This lets you see the members of these variables and drill down, which is how you get at the current class' properties: they're children of the self variable. On the right, the *console* shows any log statements produced by the app. You'll be using the console much more later on. For now, you can resize the panes with the divider between them, and show or hide either pane with the two buttons at the bottom far right.

Stepping Through Breakpoints

But how does this help fix the bug? The answer is to slowly work through the code. Like, *step by step*. At the top of the debug area, there's a small toolbar, shown in the following figure:

The buttons in the debug toolbar let us control how far to advance in the code before we pause again. Going left to right, the buttons are:

- *Show/Hide Debug Area* (⇧⌘Y)—Shows or hides the entire Debug Area. You can also do this with the menu item View > Debug Area > Show Debug Area (or "Hide Debug Area", if it is already showing).
- *Activate/Deactivate Breakpoints* (⌘Y)—If deactivated, code will no longer pause on breakpoints until reactivated.
- *Continue/Pause Program Execution* (^⌘Y)—If paused on a breakpoint, continues running the app. If the app is currently running, pauses as if you'd hit a breakpoint (usually on the main thread).

- *Step Over* (F6)—Execute one statement and immediately pause again. The meaning of "one statement" depends on where you're stopped, and thus what's shown in the editor. If it's in your own code, you step over one line of source; if you get into compiled system code, you step over one CPU instruction. This is the action you'll use the most.
- *Step Into* (F7)—Enter the function being called by this line of code, and immediately pause again on *its* first statement. This lets you dig down to trace problems, without knowing in advance every file you'd need to set a breakpoint in.
- *Step Out* (F8)—This is the opposite of Step Into: it lets the current function or method complete, then stops again once control returns to whatever called the function.
- *Debug View Hierarchy*—Show a 3D view of the app's views and subviews, something you'll do later in *Visual Debugging*, on page 130.
- *Debug Memory Graph*—Show a graph of which objects hold retaining references to others. This is covered later, in *Fixing Memory Leaks*, on page 142.
- *Simulate Location*—Set a location to be returned by calls to the Core Location framework.

The general flow of working with breakpoints is to pause at a point of interest, then step over the code line by line. You're usually looking for two things here:

- *What is the program flow?* In other words, how do the various `if`, `else`, `switch`, `case`, `do`, `while`, `until`, `for`, `sync()`, `async()`, `performSelector:`, and other instructions cause the app to follow one path through the code and not others? Do the expected branches of the code get executed?
- *What is the state of the app?* When stopped at a breakpoint, the Variables View shows the values of every variable in scope. Do those values make sense? Are the things expected to be populated `nil`?

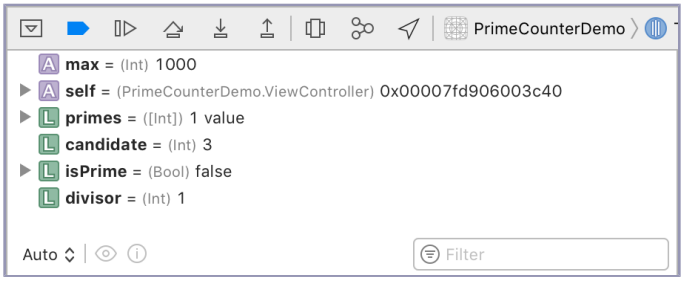
So with the app paused at the start of the outer `for` loop, it's time to apply that logic to this errant prime-finder method. The breakpoint takes effect prior to executing the line of code, so the Variables View shows the `candidate` variable as being in scope, with some huge nonsense value, since the `for` hasn't actually initialized it yet. Click Step Over and it will be assigned the value `3` for the first trip through the loop. Step Over again to set `isPrime` to `true`.

This leaves the application ready to perform the `for divisor in 1...(candidate/2)+1` loop. The idea of this part of the code is to count the integers up to half-plus-one of

candidate, testing to see if candidate divides by divisor with no remainder. If so, the number is not prime, and the code breaks out of the inner loop.

So, step over the if statement and… oops, it went into the success case and is sitting on isPrime = false. That's obviously wrong, because everyone knows the candidate, 3, is prime. So it's clear from following the program flow that the logic here is buggy.

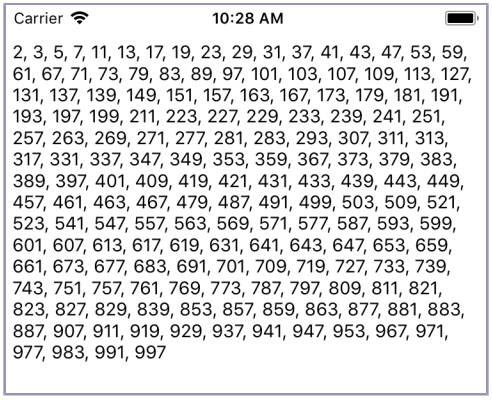
But why did the program flow enter this block? Take a hard look at the Variables View, as shown in the following figure:



Now, plug those values into the if expression that succeeded: candidate.remainderReportingOverflow(dividingBy: divisor) == (0, false). That function returns a tuple with the remainder and an overflow Bool flag; you only care that the first member is 0. The value of candidate is 3 and divisor is 1, so do the math: 3/1 is 3 with a remainder of 0.

Oh wait, darn it, you shouldn't be dividing by 1. *Every integer divides evenly by 1.* That's the bug.

Stop the app, change the 1 to a 2 and run again, disabling the breakpoint by either clicking it in the gutter, or turning off all breakpoints with the Deactivate Breakpoints button in the Debug Area toolbar. Now the bug is fixed and you get all the prime numbers you'd expect:



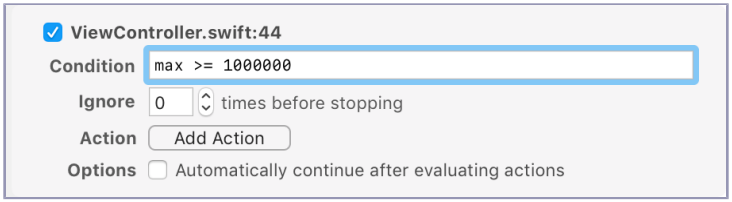
So, that's a simple example of how to use breakpoints to follow program flow, inspect variables, and figure out what's causing a bug. Still, this barely scratches the surface of what you can do with breakpoints.

Digging Deeper into Breakpoints

By default, a breakpoint does one thing: it breaks! That is to say, it halts execution when reached, and lets you inspect the program state.

While it's not immediately obvious, breakpoints can do a lot more than this. If you control-click or right-click a breakpoint in the gutter, you can delete it, disable/reenable it, or edit it. Editing a breakpoint shows a pop-over window with a bunch of interesting options.

The first thing you can do in the editing pop-over is to assign a condition. If a condition is set, the breakpoint will only pause if the condition is met. For example, if the `calculatePrimes(to:)` method were misbehaving for large values of `max`, you could set the condition to only break for `max >= 1000000`, or some similarly high value, as shown in the following figure:



The code you enter into the condition field needs to be valid for whatever language is in scope at the breakpoint. So if you were testing to see if some variable `foo` even has a value, you would use `foo == nil` to evaluate an optional in a Swift source, and `foo == NULL` for pointers in C and C++. For Objective-C, it would depend if `foo` is an object (i.e., of type `id`) or a raw pointer, using `nil` or `NULL` respectively.

As shown in the figure, you can also ignore the breakpoint a set number of times. This can be useful for situations where you have a bug that doesn't occur on the first time through the troublesome part of code, but seems to emerge later.

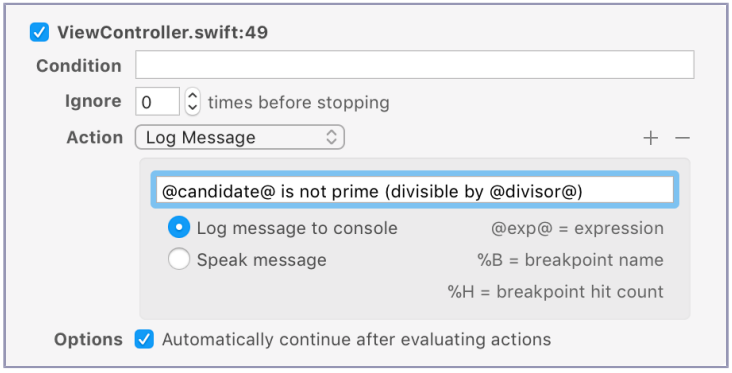
Breakpoint Actions

The bottom two items in the breakpoint editor pop-over are the Action and Options sections. There's only one option: automatically continuing after hitting the breakpoint. That seems pointless, right? Why hit a breakpoint only to then continue?

The reason to continue is that the "Add Action" button lets you add actions, and it's often enough to perform that action and continue, without stopping to investigate anything.

The simplest thing you can add is a logging action. Once you click, "Add Action", the UI expands to offer a pop-up menu of action types, and various buttons and text fields to customize the action. The simplest to understand is the Log Message action. It can log a string to the console or send it to the Mac's speech synthesizer to be spoken aloud. It also allows you to use @expression@ syntax to evaluate expressions in the currently active language and insert them into the log string.

The following figure shows a breakpoint on the isPrime = false line, logging a reason for ruling out candidate as a prime number, with the log string @candidate@ is not prime (divisible by @divisor@):



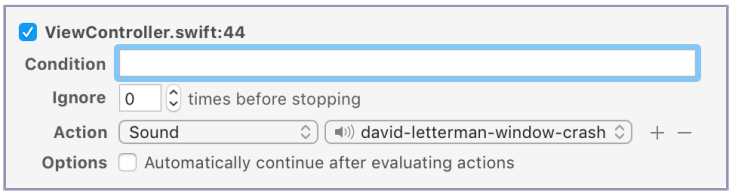
When the app runs, it doesn't stop on the breakpoint, but each hit does log out a message to the Console down in the Debug Area:

```
3 is not prime (divisible by 1)
4 is not prime (divisible by 1)
5 is not prime (divisible by 1)
6 is not prime (divisible by 1)
7 is not prime (divisible by 1)
```

Using this approach shows the cause of the earlier bug quite easily. Also, using breakpoint log actions can save you time compared to manually adding logging statements to your code, since it eliminates two compiles: one to add the log statement and another to remove it. Plus, it eliminates the risk of accidentally shipping logging statements with your code, which will only slow down your app at runtime and bog your users' filesystems with worthless log messages.

Another interesting action is to play a sound when the breakpoint is hit. The list of available sounds are the various alert beeps shown by System Preferences on

your Mac, but you can add sound files to ~/Library/Sounds and they'll appear in the pop-up after you restart Xcode. In the following figure, a new breakpoint is using the window crash sound effect from David Letterman's "Top 10" segment:



OK, this sounds kind of silly, but it can be genuinely useful. If you have a rare condition that you want to catch, set a crazy loud breakpoint like this, leave the app running and you will be able to hear it even if you are in the next room.

There's an even more practical use of sound breakpoints, and that's when you have some code that should be called very frequently, but you're not sure if it is. This happens a lot with the media frameworks and callbacks from continuous user input actions, like mouse drags and sliders being swiped back and forth.

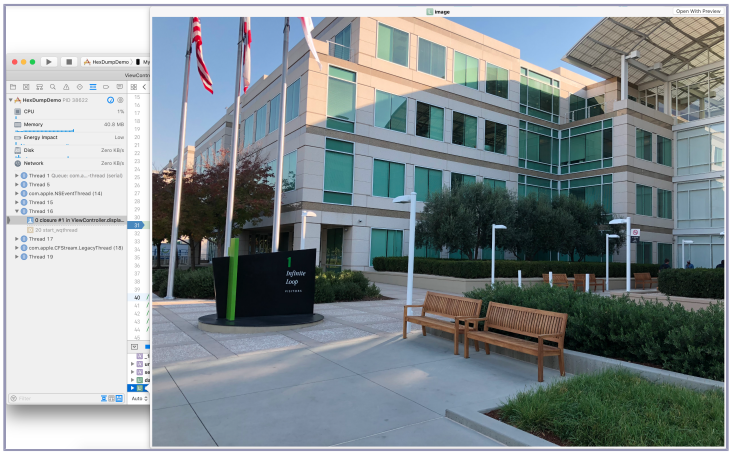
For example, imagine you have an iOS `UIPanGestureRecognizer`, and you want some kind of feedback while testing a drag gesture. In the callback method that receives the gesture recognizer's events, you could set a sound breakpoint that plays a very short sound (like "Pop" or "Tink"), and automatically continues after performing the action. If your gesture recognizer is set up correctly, this will play the sound every time your touch moves, so you can hear the rapid series of tinks or pops as confirmation that your method is being called.

Digging Deeper into Variables

Once you're stopped at a breakpoint, you can do more with the variable view than just look at the list of names and values.

For numeric types, strings, and types that can be usefully expressed as strings (like URLs), the one line of text in the variable view is often enough to be useful, at least in Swift and Objective-C (in C, it may be helpful to use the contextual menu item "View Value As…" to switch between showing the value as a numeric type, pointer, C string, etc.). For other types, however, the Variables View just shows the instance's address in memory. That's usually not something you can work with, but some of these types have a more useful preview available. At the bottom of the Variables View, there is a button that looks like an eye. This is the *Quick Look* button.

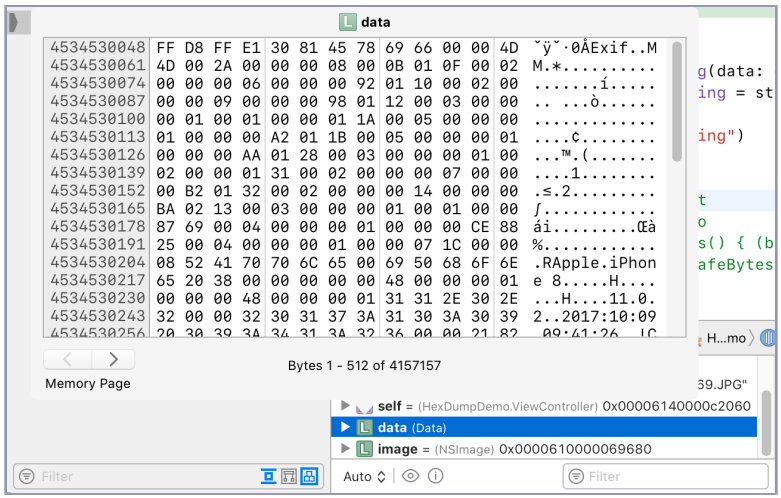
You enable the Quick Look button by selecting one row in the Variables View. For arbitrary types, it shows a pop-over with simple things like the variable's name and type. But for some types, it provides a visual preview. For example, for any kind of image (UIImage, NSImage) or view (UIView, NSView), or even a URL whose path extension is a known image type like .jpg, Quick Look shows the visual contents in a pop-over. There's a button in the pop-over's top frame to open the image in the Preview app, or you can dismiss the pop-over by just clicking anywhere outside of it. The following figure shows a Quick Look preview of a UIImage:



Quick Look also has a nice preview mode for working with Data or NSData instances. It offers a paged view of the actual memory contents pointed to by the data reference. This can be useful if the raw memory layout is in a format you can recognize, like file formats that start with familiar magic strings like ID3 at the beginning of an MP3 file with ID3 metadata, JFIF in a JPEG graphic, PNG in a PNG graphic, etc. In the following figure, it's obvious that the image encoded in this Data begins with Exif metadata, and the photo was taken with an Apple iPhone 8 on October 9, 2017. On the other hand, it's kind of bizarre that the pop-over has a fixed width and wraps lines based on your font size, rather than breaking on power-of-two groupings. If the data in your buffer is meaningful enough for you to want to inspect it, chances are that strides of 4, 8, or 16 bytes would make more sense for most than the 13 seen in the figure on page 118.

The Debugging Navigators

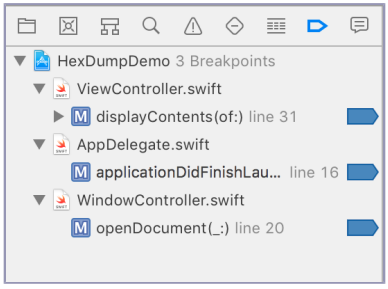
Once you've added a few breakpoints, it might occur to you that in a large project, you could easily lose track of all the places you've set up breakpoints. And it would be a hassle to have to visit each source file and manually remove its breakpoints when we're done with them.



Deep Diving into Memory

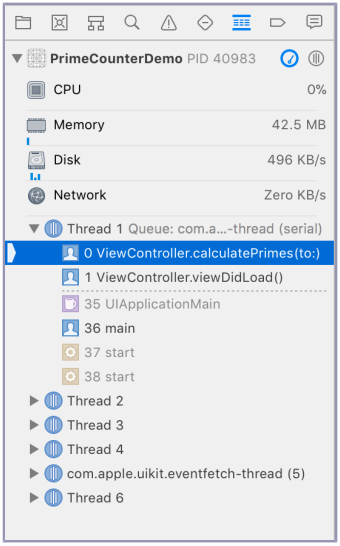
You can also right-click variables and use “View memory of…” to switch the Editor Area to a view of the raw memory at the location pointed to by the variable. This usually isn’t very useful for high-level languages like Swift, because the internal representation of their types is opaque. Even a `Data` looks like a bunch of nothing. Where it’s useful is in C code, when you have a `void*` reference to a memory buffer. Viewing the memory locations of these pointers will show you their contents, and with a lot more room to work with than the Quick Look pop-over provides.

Fortunately, there’s a centralized view for all the breakpoints in the current project. In the Navigator pane on the left side of the Xcode window, notice that one of the small toolbar icons is a little breakpoint icon. This takes you to the Breakpoint Navigator; you can also show it with View > Show Breakpoint Navigator (⌘8), as seen in the figure:



This navigator shows every file that has breakpoints set. From this view, you can enable, disable, edit, or delete any of the breakpoints. Selecting a breakpoint from the list also navigates to that file and line number in the Editor Area. The bottom of the list also has a filter field to help you find breakpoints by filename or method/function name, in case you create more than you can quickly read through.

At the bottom of the navigator, there's also a plus (+) button for adding special kinds of breakpoints, which will be shown later in *Finding and Fixing Crashing Bugs*, on page 124.



The other navigator that comes into play with breakpoints is the Debug Navigator. By default, Xcode automatically switches to this navigator when it stops on a breakpoint; you can also show it with View > Debug Navigator (⌘7), as seen in this figure.

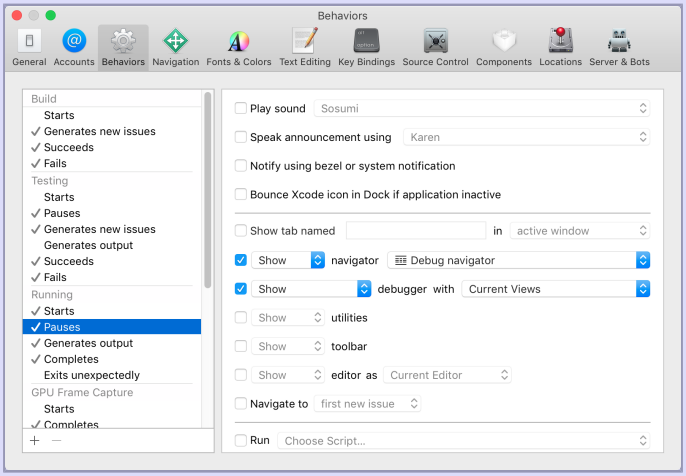
The purpose of the Debug Navigator is to show you an overview of resource usage (CPU, memory, disk, and network), followed by all the currently active threads, and what they are doing at the moment the app is paused. Each thread can be expanded to show its *call stack*: the chain of function calls that began with the creation of the thread (or the app itself) and has reached the currently executing statement. When you hit a breakpoint, the top of the stack will be a function or method in your code.

Each line of the list actually represents a *frame*, the state of the app at that call point. For example, in the figure, the top of the stack is the breakpoint in `calculatePrimes(to:)`. With that line selected, the Editor Area will show that source file, and the Variables View will show all the variables in scope at that point. Now, notice the next line down in the Debug Navigator is `ViewController.viewDidLoad()`. This is the method that *called* `calculatePrimes(to:)`. If you click that line, the Editor Area will show `ViewController.swift`, and the Variables View will show the variables visible inside `viewDidLoad()` at the moment it called `calculatePrimes(to:)`. This can be useful when figuring out why one of your methods called another, or perhaps why it sent incorrect parameters.

Code from your project is shown with dark text in the list. Calls made from code that you don't have source for, such as the system frameworks, is shown in gray. You can still click these, but all you'll see in the Editor Area is the disassembled machine code. Since the call stacks tend to be *very* deep, with dozens of system framework calls that don't really affect you, there are three buttons in the filter at the bottom of the navigator that hide certain calls, such as those that don't contain debug symbols (meaning they're optimized code that the debugger won't be able to do much with… again, this usually means the system frameworks).

Debug-Time Behaviors

The automatic presentation of the Debug Navigator is actually a default behavior that you can customize. Under Xcode's Preferences, select the Behaviors tab, and look at the list of events on the left side. Two of these are relevant to debug time. Running > Pauses determines the behaviors for when you hit a breakpoint or click the pause button manually. By default, it shows both the Debug Navigator and the Debug Area, although you can customize or disable both these behaviors. You can also add other behaviors, like opening a new tab just for debugging.



The Running > Generates Output event determines what happens when the app generates any log messages. Usually, this is used to show the Console if it's not already visible. But you can customize this event as you see fit too.

Debugging with the Console

So how does this all work, anyway? All the features of Xcode's debugger are provided by the *Low Level Debugger* (LLDB), which combines reusable components from the larger LLVM project. LLDB is deeply integrated into Xcode, but also exists in other forms, such as a command-line executable.

The Many Faces of LLDB

LLDB also powers the Swift REPL (Read, Evaluate, Print, Loop), which allows you to work with Swift code interactively on the command line. Try it out sometime by typing `swift` in the Terminal app.

LLDB also offers a Python API to interact with debugging sessions programatically. The LLDB website at http://lldb.llvm.org/ has documentation and sample code.

Interacting with LLDB

As it turns out, the Console side of the Debug Area isn't just for log output: it's a full-power version of the LLDB command-line interpreter. And it can do some really cool stuff.

Notice that when you're stopped on a breakpoint, there's an "info" button on the toolbar at the bottom of the Variables View, next to the Quick Look button. The tooltip says "Print Info", and if you click it, a summary of the selected variable is printed to the Console. For example, for a table view, the output looks something like this:

```
Printing description of tableView:
<UITableView: 0x7f7f17007000; frame = (0 0; 375 667); clipsToBounds = YES;
 autoresize = W+H; gestureRecognizers = <NSArray: 0x60c00045d130>;
 layer = <CALayer: 0x604000222200>; contentOffset: {0, 0};
 contentSize: {375, 0}; adjustedContentInset: {0, 0, 0, 0}>
```

It turns out, the button is just a convenience. You can type directly into the console to get the same result. Just click in the Console after where it says (lldb), and type `po tableView` to get exactly the same output.

What you're doing here is interacting directly with LLDB. `po` means "print object" and works with any variable in scope. `po` uses the object's `debugDescription` string (falling back to `description` if necessary) to show a human-readable summary of the variable. You can also use an unformatted print command, `p`, which shows the object's address in memory and the raw values of all of its members.

Evaluating Expressions with expr

You can type `help` in the console to get a list of all the available LLDB commands. Notice, for example, that everything you've seen with breakpoints can be done from this command line: you create breakpoints with `breakpoint set` (passing in the filename and line number with `-f` and `-l`) flags, step over lines with `step`, and unpause execution with `continue`. The GUI's nicer, sure, but all the functionality is also here on the command line if you need it.

The most powerful command in LLDB is `expr`. As its name suggests, it evaluates arbitrary expressions. You can type in a Swift expression and see the results:

```
⇛ expr ["iMac", "Mac Pro", "iPad"].filter({$0.starts(with:"i")})
❮ ([String]) $R2 = 2 values {
    [0] = "iMac"
    [1] = "iPad"
  }
```

It's not just arbitrary Swift language code that works here either. LLDB is fully aware of your code, since debugging your code is the whole point of being in the debugger after all. So, one thing you can do is to call your own functions or methods, as in expr calculatePrimes(to: 23), which will log the resulting array to the Console.

But it goes even deeper. Consider the following trivial implementation of a UITableViewDataSource:

debugging/LLDBExprDemo/LLDBExprDemo/ViewController.swift

```
class ViewController: UITableViewController {

  var members: [String] = ["Madoka", "Sayaka", "Mami", "Kyoko", "Homura"]

  override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
  }

  override func tableView(_ tableView: UITableView,
                          numberOfRowsInSection section: Int) -> Int {
    return members.count
  }

  override func tableView(_ tableView: UITableView,
                          cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
      let cell =
        tableView.dequeueReusableCell(withIdentifier: "MemberCell",
                                      for: indexPath)
      cell.textLabel?.text = members[indexPath.row]
      return cell
  }
}
```

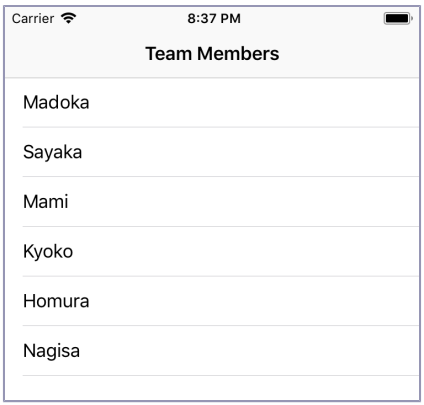
Obviously, this just creates a table with the five strings in members. *Unless someone were to interfere with it,* that is.

Imagine what happens if you put a breakpoint in numberOfSections(). The first time the breakpoint is hit, enter the following command in the Console:

```
⇛ expr members.append("Nagisa")
```

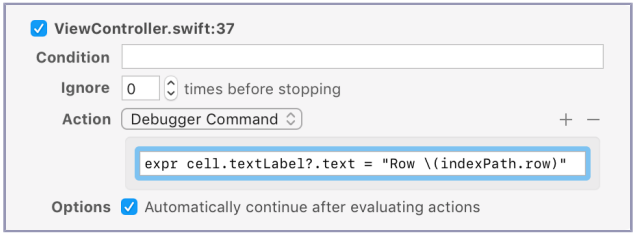
Then disable the breakpoint and unpause the app. When the table comes up, it will show *six* members, including the one you just added via the debugger, even though it's not in the compiled code as shown in the figure on page 123.

Now, think about what this means. You can use expr to change the values of variables *while your app is running* and stopped on a breakpoint. Instead of applying a fix with a compile-and-run-again cycle, you can sometimes try out different values while the app is still running. "Sometimes," because there are some limitations. expr can't rewrite the value of a Swift let constant, because

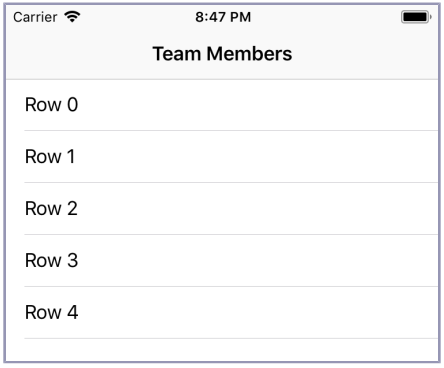


the whole point of a constant is that it can't change. As a result, this technique tends to be a little more useful in Objective-C, where most things are variable.

In fact, one of the actions in the breakpoint editor is a text field that accepts an lldb command. Almost anything that works on the command line is fair game here, including calls to expr. So, you can put a breakpoint on `return cell` in `tableView(cellForRowAt:)` and set an action to execute the command `expr cell.textLabel?.text = "Row \(indexPath.row)"`, as shown in the following figure:



Set the breakpoint to automatically continue, run the app, and enjoy the sight of the table's canned data being completely ignored and instead overwritten by the debugger:



Finding and Fixing Crashing Bugs

Breakpoints are great, but being able to set a breakpoint implies that you have some clue about where a bug is hiding, that you know something in a certain class or method is misbehaving. Problem is, what do you do when that's not the case?

The most common scenario for not knowing where to begin is crashing bugs. When you crash while running your app from Xcode, the debugger halts the program, but often you'll find that it's stopped at the highest level of the app, like in the main() function that launched the app, completely removed from whatever caused the crash. In Objective-C—and this was particularly common before Automatic Reference Counting (ARC) simplified memory management—crashes often stop the app in objc_msgsend(), mistakenly leading newcomers to the platform that this function is buggy, when in fact it's just the C function to perform any Obj-C method call.

Whether you land in main() or objc_msgsend(), the underlying problem is usually that an error or exception has gone unhandled all the way up the call stack. Sometimes the crash is on a different thread than where you stop, or you don't stop until the next trip through the run loop. In all these cases, you need a technique that gets you into the debugger at the instant the bug occurs, not after it's crashed everything.

Exception and Error Breakpoints

For some kinds of crashes, Xcode gives us exactly what you need. In Objective-C, many runtime problems throw exceptions which, if they aren't caught, crash the app. Perhaps the most common of these is making an out-of-bounds access of an NSArray. Consider the following buggy model for a table view:

debugging/ExceptionBreakpointDemo/ExceptionBreakpointDemo/ViewController.m

```
-(instancetype) initWithCoder:(NSCoder *)aDecoder {
  self = [super initWithCoder:aDecoder];
  self.members = @[@"Honoka", @"Kotori", @"Umi", @"Hanayo",
                   @"Rin", @"Maki", @"Nico", @"Nozomi", @"Eli"];
  return self;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
  return 1;
}

- (NSInteger) tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {
  return self.members.count + 1; // bug!
}
```

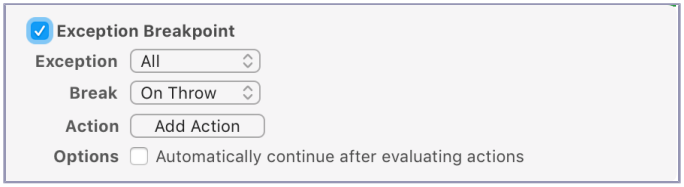
Because [tableView:numberOfRowsInSection:] reports more rows than the members array actually contains, the app will crash when trying to fill a non-existent 10th row of the table.

Problem is, when you run this code, you have no idea this is the problem. The crash stops the app in main.m, which exists only to launch the app in the first place. The console tells us what the exception is, but not where it occurred:

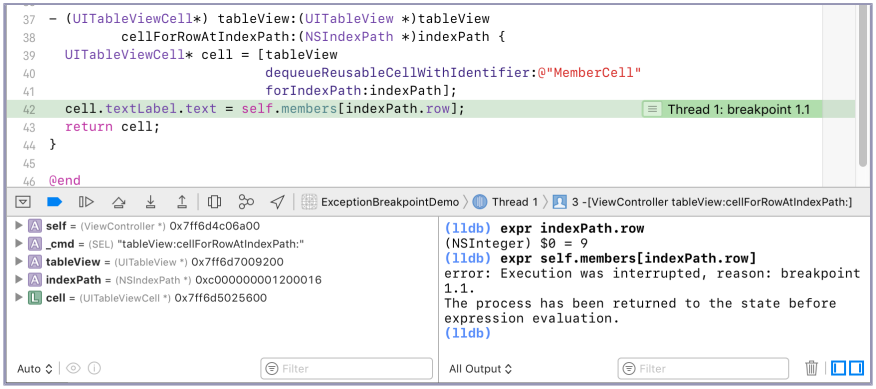
```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[__NSArrayI objectAtIndexedSubscript:]: index 9 beyond
bounds [0 .. 8]'
```

In a big app, you'd have to start trying to imagine every place in your codebase where there are arrays that might have gone out of bounds. Fortunately, there's a better alternative: *exception breakpoints*.

Open the Breakpoint Inspector and notice at the bottom, there's a plus (+) button. This pop-up menu adds various special kinds of breakpoints. If you choose "Exception Breakpoint..., as shown in the figure, you can add a breakpoint that stops for Objective-C and/or C++ exceptions, either when they're thrown or caught:



Run the app again and now Xcode will catch the exception when it's thrown, opening the Editor Area to that point, with the breakpoint indicating the line that is crashing. As you can see in the figure, it's the call to self.members[indexPath.row], and from there you can investigate the crashing state with the Variables View or by using debugger commands in the Console:



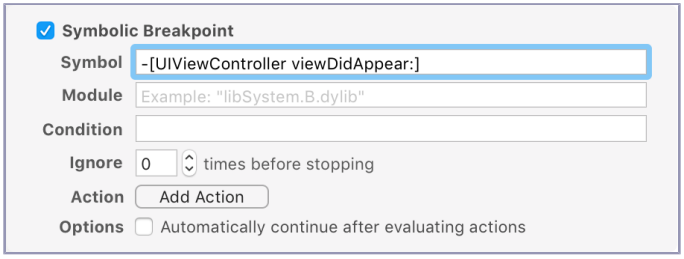
It's also worth noting that if you look at the Debug navigator, your method will not be the top of the call stack—you'll instead see that objc_exception_throw() is the current call.

What's interesting about exception breakpoints is that they let the debugger pause the app for a special condition, rather than just breaking on a combination of source filename and line number like normal breakpoints do. The other breakpoints created with the plus (+) button are similar. For example, there's a *Swift error breakpoint* that's more or less equivalent to the Objective-C/C++ exception breakpoint, except that it works when Swift errors are thrown. However, since Apple's frameworks aren't written in Swift (yet), this will only be useful for catching Swift errors thrown by your own code, or third-party Swift frameworks.

Symbolic Breakpoints

Another of the special breakpoint types, the *symbolic breakpoint*, is also really useful for investigating hard-to-find bugs. It lets you stop on any method or function call, by name. This gives you a way to set breakpoints in code you don't own, such as the system frameworks. For example, you can set a breakpoint on viewDidAppear to catch when any UIViewController finishes showing its view.

The challenge with setting symbolic breakpoints is you have to use the language of the symbol you're calling. For example, UIKit is written in Objective-C, so even if all your classes are in Swift, you still have to use Obj-C syntax to set this breakpoint—such as -[UIViewController viewDidAppear:]—as seen in the following figure:



On the other hand, if you only wanted to stop on a single class' viewDidAppear(), and you knew the class was written in Swift, you'd use the Swift syntax for the breakpoint: SomeOtherViewController.viewDidAppear. For Swift, you only provide the method name, not the inner or outer names of the parameters in its signature, nor the parentheses.

Symbolic breakpoints are really useful for finding out whether certain code is being reached. If you think you're leaking memory in a navigation-based

iOS app, a go-to technique is to set a symbolic breakpoint on -[UIViewController dealloc], drill down in the navigation, and then start going backwards through the navigation stack. Each time you go back, the view controller you backed out of should hit the breakpoint a second or so after the animation finishes, which indicates that the view controller is being freed from memory. If it doesn't, you're probably leaking memory. *Discovering Memory Leaks*, on page 136 introduces more advanced tools for finding and fixing memory leaks.

Main Thread Checker

Another common kind of crash occurs when you access UIKit or AppKit objects from a Grand Central Dispatch queue other than the main queue. For performance reasons, much of the code in these frameworks is thread-unsafe, with the understanding that it will only be called from one thread, and thus, one GCD queue.

Problem is, it's easy to overlook which queue your code is being executed on. The classic example of this is performing networking, and then updating the UI with data fetched from the net. The completion handlers and callbacks used by Foundation's URL Loading System always use background queues—this keeps them from slowing down the main thread and its UI tasks. But the obvious upshot is, well, you're obviously not on the main queue anymore.

Here's a simple example of code that crashes. It just loads the HTML of the http://apple.com/ website and attempts to convert it to a string and set that string as the text of a UITextView:

debugging/MainThreadViolationDemo/MainThreadViolationDemo/ViewController.swift

```
let url = URL(string: "http://apple.com/")!
let dataTask = urlSession.dataTask(with: url) { dataOrNil, _, _ in
  if let data = dataOrNil,
    let string = String(data: data, encoding: .utf8) {
    self.textView.text = string // bug!
  }
}
dataTask.resume()
```

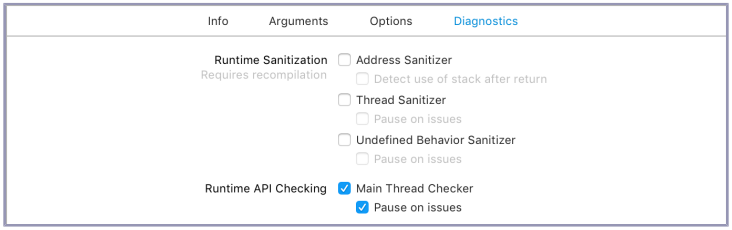
The closure will be executed on a non-main queue. Since it attempts to modify a UIKit class by updating textView, it's doomed to crash. And while it does crash, the Console output is pretty interesting:

```
Main Thread Checker: UI API called on a background thread:
 -[UITextView setText:] PID: 77721, TID: 8567832, Thread name: (none),
 Queue name: NSOperationQueue 0x61000022fa00 (QOS: UNSPECIFIED), QoS: 0
```

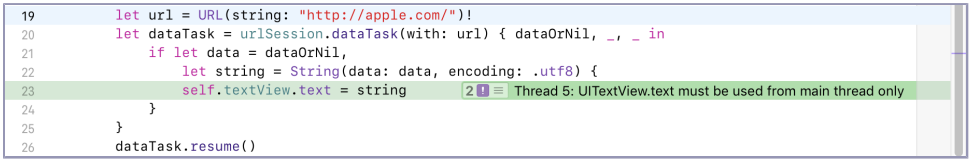
The *Main Thread Checker* is a tool that only runs at debug time, and checks the thread modifying UIKit or AppKit objects to make sure it's the main thread

(which, as developers, we access by putting work on the main GCD queue). When a violation is caught, it logs the above message and a stack trace which implicates the call.

The stack trace can be hard to read because it uses the machine-generated name of the offending closure, and the error still crashes the Content area into the assembly of a system call, __pthread_kill, so it's not the prettiest way to catch a bug. Fortunately, Main Thread Checker can be made nicer. Go to the scheme selector, choose "Edit scheme…", select the "Run" scheme, and then go to the "Diagnostic" tab, shown in the figure:



By default, Main Thread Checker is enabled, but its option "Pause on issues" is not. Enable that and run again. This makes the Main Thread Checker work kind of like an Exception Breakpoint, stopping execution at the point where the crash begins, and showing it to you in the Editor area:



Of course, now that you know where it is, you still need to fix the bug. The simple recipe for problems like this is to just put your UI work on GCD's main queue with DispatchQueue.async:

debugging/MainThreadViolationDemo/MainThreadViolationDemo/ViewController.swift

```
DispatchQueue.main.async {
  self.textView.text = string
}
```

Address Sanitizer

Now, throwing an exception out to the main() function is all well and good, but experienced developers know that to really create terror and havoc, you need to be working with pointers, preferably in C. Not for nothing does Swift give its pointer types scary names like UnsafeRawPointer.

Here's a C function called accessDeallocatedMemory() that is a guaranteed 100% crasher. It's dirt simple, even if you don't read C—it allocates some memory

(just big enough to hold one UInt32) and stores the pointer as buffer, deallocates the memory, and then tries to set the value of pointed to by buffer:

debugging/AddressSanitizerDemo/AddressSanitizerDemo/BuggyCFunctions.c

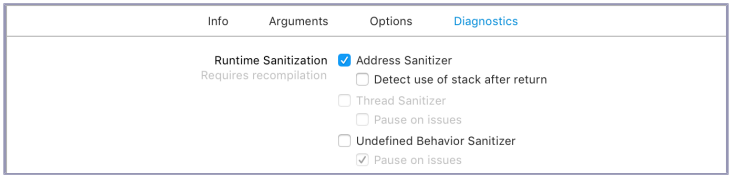
```
void accessDeallocatedMemory(void) {
  UInt32 *buffer = calloc(1, sizeof(UInt32));
  free(buffer);
  buffer[0] = 42;
}
```

Since the app no longer owns the memory pointed to by buffer, this crashes the app instantly. The app crashes out to main() and logs a minimally helpful message:

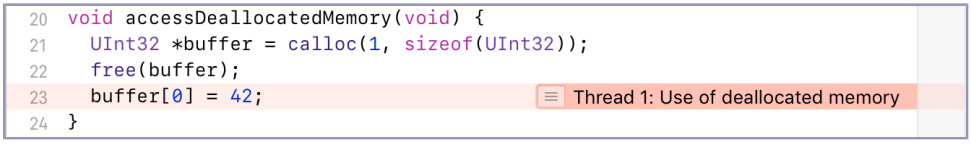
```
malloc: *** error for object 0x614000000880: Invalid pointer dequeued
from free list
*** set a breakpoint in malloc_error_break to debug
```

OK, sure, you could take the advice about setting a breakpoint to debug this, but Xcode 9 provides a much more compelling tool: the *Address Sanitizer*. This is a tool that marks unallocated and freed memory as "poisoned", and can catch accesses to poisoned memory at the moment they happen.

This approach is fairly expensive and slows your code down by 2 to 5 times. Because of this, it's not on by default like the Main Thread Checker. Instead, you have to turn it on, in the same place as before (Scheme Selector > Edit Scheme > Run > Diagnostics), as seen here:



Run the app again, and this time the crash will be caught on the line of code that causes it, as seen in the following figure (you'll also see the offending call in the stack trace in the Debug Navigator, along with a bunch of log messages from the Address Sanitizer in the Console):



The Address Sanitizer also catches memory mistakes that may not crash your app, but are still bad news. One obvious example of this is a buffer overrun, like the following code:

debugging/AddressSanitizerDemo/AddressSanitizerDemo/BuggyCFunctions.c

```
void performBufferOverflow(void) {
  UInt32 *buffer = calloc(1, sizeof(UInt32));
  buffer[0] = 42;
  buffer[1] = 9999999;
  free(buffer);
}
```

In this code, buffer is only allocated enough memory to hold one UInt32, so the attempt to set buffer[1] is out of bounds. When run, this may not actually crash, but since it's writing to an out-of-bounds address, you can't actually know what's being overwritten at that address. Fortunately, the Address Sanitizer catches this programming error too:

```
13 void performBufferOverflow(void) {
14   UInt32 *buffer = calloc(1, sizeof(UInt32));
15   buffer[0] = 42;
16   buffer[1] = 9999999;          Thread 1: Heap buffer overflow
17   free(buffer);
18 }
```

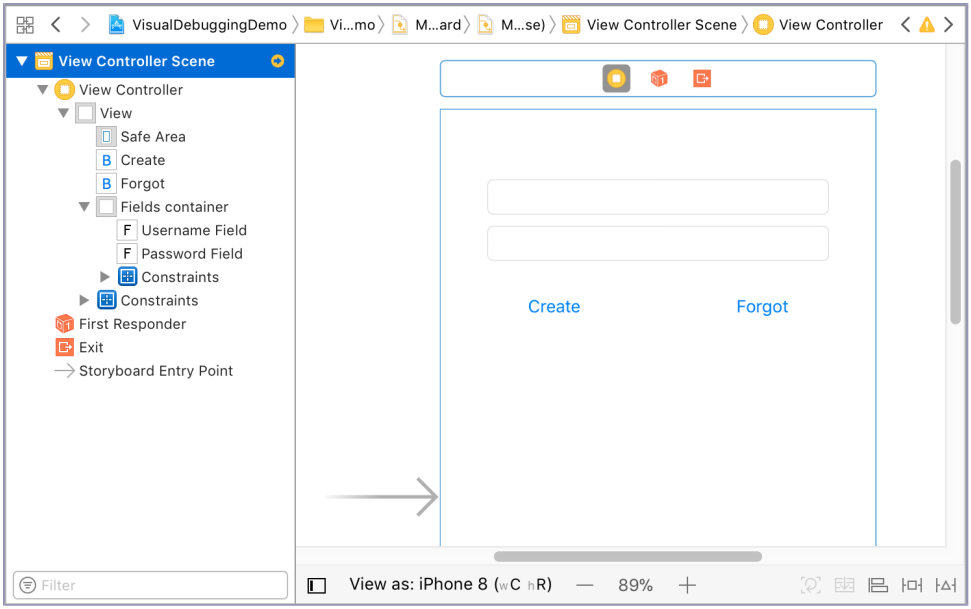
Even if you're not crashing, it's probably a good idea to occasionally run the Address Sanitizer on any C code you've written, to catch overflow errors like this one (just don't forget to switch it off). Overflows are a major source of security problems, since attackers can use overflows to access memory locations they should not have access to. Proper care of your memory allocations and pointers can help put an end to that.

Visual Debugging

LLDB provides a tremendous collection of tools for finding bugs in your code. Only problem is, not every problem exists in code. Consider the storyboard scene in the figure on page 131.

This looks simple enough: text fields for username and password, and buttons for "Create" and "Forgot". Except that, as they're configured in the book's download code, when you tap on the buttons, nothing happens.

How do you debug this? A good first instinct might be to set some breakpoints. If the buttons are connected to IBAction methods (rather than just being connected to segues entirely within the storyboard), you could set breakpoints on those methods to see if they get hit. Cue narrator voice: "They don't get hit." Or maybe you could set a symbolic breakpoint on -[UIControl sendAction:to:forEvent:] to catch any sort of UI interaction, including all button taps (Narrator: "That doesn't work either.")



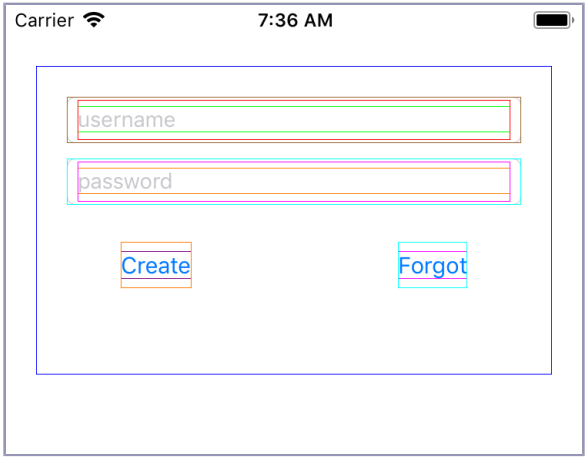
So maybe the problem isn't in the code at all, but is in the storyboard. LLDB won't help in the storyboard, but there are a few things you can check, like whether the "Allows User Interaction" check box has somehow been disabled, although this is a long shot.

Actually, most of the time when you have a button that can't be tapped, it's because something else is on top of it, intercepting all the taps that should get delivered to the button. But if there is indeed a totally invisible UIView stacked atop the buttons, how would you find it? This is where Xcode's *visual debugging* comes in.

There are several tools you can use with Xcode and the Simulator to understand layout and presentation issues. In the Simulator, there is an entire Debug menu with options that will slow down animations so you have time to understand them, and a set of menu items that deliberately mis-color subviews so you can inspect them. For example, the figure on page 132 is from selecting "Color Blended Layers", which lets you reason about where layering of your subviews should and shouldn't be happening. In this case, it shows interesting traits like the insets and rounded corners of the text areas, but doesn't actually show any views blocking the buttons:

Back in Xcode, there are more visual debugging tools to help crack this bug. Under the Debug menu, the "View Debugging" submenu has several options

that pause the app in the simulator and change how it's rendered. Two of these, "Show View Frames" and "Show Alignment Rectangles", will draw boxes around every view and subview, making it easier to reason about their respective frames and layouts. Using Show View Frames, as in the following screenshot, reveals a big view that overlaps the buttons; if it's higher in the z-order, that would account for why the buttons can't be tapped:

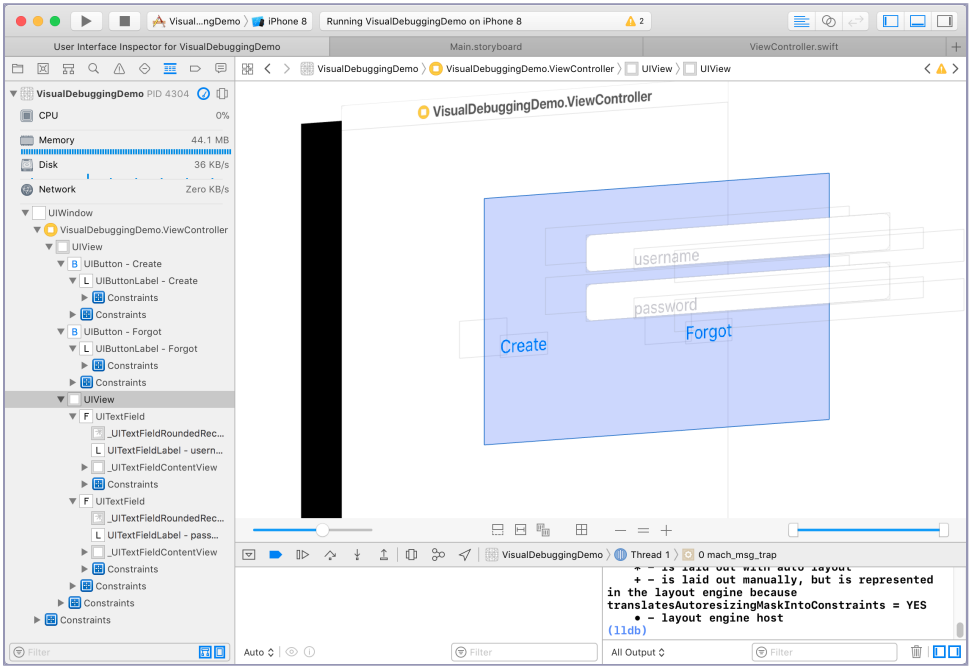


This is where Xcode's most awesome visual debugging tool comes in: *Debug View Hierarchy*. You can launch this debugger either from its button on the debug toolbar (to the right of Step Out), or on the top line of the Debug Navigator by clicking the circle button with the tooltip "View process in different ways" and then choosing "View UI Hierarchy" from the pop-up menu.

The View Hierarchy Debugger pauses the app and takes over both the Navigator and Editor panes. In the Navigator, it shows the tree structure of the views, kind of like the Document Outline in a storyboard. In the editor, it shows the current screen with thin outlines around all its views.

This is where it gets interesting. The Editor view is a 3D representation of all the layers of the subviews. Dragging horizontally pivots your view around the Y-axis so you can see layers from the side, and dragging vertically pivots around the X-axis to look around tops and bottoms. Initially, everything is flat, but a slider at the bottom left increases the Z-axis spacing between the layers, so you can spread things out in that dimension. You can also click individual layers in the Editor area to select them in the hierarchy tree, or select a layer in the Editor area and use Navigate -> Reveal in Debug Navigator (⇧⌘D) to select it in the tree.

By playing around with the controls, you can reveal the offending subview that's blocking the buttons, as shown in the following figure:



This shows that there is indeed a large view with a clear background that serves as a container for the text fields, but also blocks the buttons from receiving taps. There are several possible fixes. You could change the Auto Layout so it doesn't overlap (in fact, to make this demo, I used an unrealistic

height constraint to achieve the blocking). You could also drag the view higher up in the storyboard's Document Outline, to reduce its z-position and thus put it under the buttons.

There are several other interesting controls at the bottom of the View Hierarchy Debugger. There is one button to show where views have clipped their contents, and another to reveal Auto Layout constraints. To deal with visual complexity, a button next to these shows a pop-up that lets you toggle the view between contents, wireframes, or both. In the middle, there are buttons to switch between 2D and 3D, and to zoom in and out if you don't have a trackpad (to be honest, though, you're going to enjoy this feature a lot more with trackpad gestures than with a mouse). Finally, if your view is really complex, there's a slider on the far right with two playheads that lets you filter out which layers to show. Sliding in from the left filters out views starting at the bottom of the z-order, and sliding the right playhead filters out views from the top.

Wrap-Up

By this point, you should be well-equipped to track down and eliminate many different kinds of logical bugs in your application. Just look at all the tools Xcode gives you:

- Basic breakpoints let you stop and step through program flow, while checking on the values of variables at each step.
- The LLDB command line in the Console lets you evaluate expressions and even modify the state of your app while it's running in the debugger.
- For common problems, Xcode comes built-in with tools to catch main thread violations in UIKit and AppKit, and for memory-addressing bugs when you're working with pointers.
- For layout and other UI bugs, you can use multiple view debugging tools in the Simulator and Xcode, including the awesome 3D power of the View Hierarchy Debugger.

These approaches will make your app run correctly, but there's still more work to do. It's possible your non-crashing, good-looking, logically correct app is wasting memory or CPU resources… and that can lead to slowdowns, instability, or reduced battery life. The next chapter shows off how Xcode helps you find and fix those bugs.

CHAPTER 7

Improving Performance

Few things are as annoying as an app that isn't doing anything. You tap, you tap again, you tap a third time… and nothing happens. "What the heck?", you say, "What is this stupid thing doing?"

The irony is, maybe the app *is* doing something. Maybe it's desperately making memory available for some memory-intense action, or chowing through the loops of some deeply nested code. But even if there's a good reason, most users think a slow app is a bad app.

In this chapter, we're going to look at some of the tools Xcode gives us for speeding up our apps.

Managing Memory Mistakes

Performance, as it turns out, isn't just about writing the most efficient algorithms for important tasks. Everything operates in a context, and in the mobile context, memory is perhaps the most important consideration.

You can, for example, trade memory for better performance. Consider a hypothetical piece of hardware on which trigonometric operations like sine and cosine are really slow—don't worry, this is not an "asking for a friend" scenario, since iOS devices are pretty good in this department. If you were writing a game that used a lot of math, this slowness could hurt your entire application. One way you might deal with it would be to compute sines and cosines for every degree once, and then just put them in a big 360-member array. Then your trig functions would just be an array lookup: nearly instantaneous. The trade-off is that you'd sacrifice some memory to do this: 2-byte `Float`s times 360 indexes would cost about half a kilobyte. In this hypothetical example, that's almost certainly a good deal to take.

But a lot of the times, we developers are not nearly so deliberate in our choices, and nor do we use such small amounts of memory. Our cameras take 20-megapixel pictures, which we happily load into memory. All our views are backed by `CALayers` that may have 32-bit ARGB representations of their full-size contents, even if we're viewing them at smaller sizes and thus never seeing the full detail. Throw in some large JSON responses from webservice calls or our own complex data models, and our memory use can add up.

And that's still fine… *if* we're careful about only keeping things in memory for as long as we need them. But when we screw that up, things get bad *fast*. *Memory leaks* are perhaps the most common and most destructive programming error on iOS. They're common because of the inherent challenges of dynamic memory management, and dangerous because these devices don't have endless memory to spare.

Once memory gets tight, this becomes a performance problem. There's a good chance you've seen your phone slow way down while using an app, particularly some of these bloated social networking apps that make heavy use of embedded webviews. Sometimes what's really going on is the system desperately trying to free up memory, terminating suspended apps left and right, as the foreground app chows through more memory.

Use a Real Cache

Our hypothetical array-of-sines example is a de facto *cache*, and there may be times you'd like to cache some data in memory, rather than re-generating or re-downloading it. You might be tempted to use a simple `Array` or `Dictionary`, but you should really consider using a genuine `NSCache` instead.

The `NSCache` class, from the Foundation framework, has a lot of nice memory-management features. Objects are evicted from the cache automatically when memory gets tight, and you can specify a maximum number of objects or a maximum "cost" (in terms you decide) for the entire cache before old contents are removed.

Discovering Memory Leaks

When you have a memory leak, it can be hard to tell that you even have a problem. Let's start with a scenario that will make it obvious, and hone your techniques from there.

The download sample code includes an app called `MemoryLeakDemo` that makes a memory leak really obvious. It's a navigation app whose first scene just has

a button that says “Show Picture”, and a second that loads and displays a picture. The second scene consists of a PictureViewModel and PictureViewController.

The PictureViewModel is designed to retain a gaudy, absurd amount of memory. It uses Bundle.main.url(forResource:extension:) to find an image in the app bundle, loads the raw Data from that location, makes a UIImage from that, and then holds on to both the image and the raw data as properties. *Don’t ever write code like this:*

performance/MemoryLeakDemo/MemoryLeakDemo/PictureViewModel.swift

```
import UIKit

protocol PictureViewModelDelegate {
    func modelDidLoadPicture(_ model: PictureViewModel)
}

class PictureViewModel {

    private (set) var image: UIImage?
    private var imageData: Data?
    var delegate: PictureViewModelDelegate?

    func loadImage() {
        guard let imageURL =
            Bundle.main.url(forResource: "bundle-image",
                            withExtension: "jpg") else { return }
        do {
            let imageData = try Data(contentsOf: imageURL)
            if let image = UIImage(data: imageData) {
                self.imageData = imageData
                self.image = image
                self.delegate?.modelDidLoadPicture(self)
            }
        } catch {
            print ("couldn't load image data: \(error)")
        }
    }
}
```

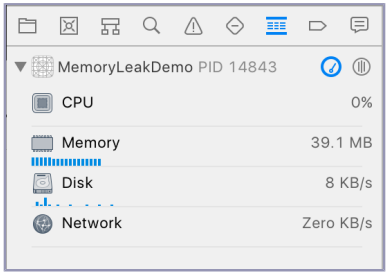
It would obviously be simpler and more responsible to put the image in the asset library, and load it with UIImage(named:). But the point here is not to be responsible. The idea is to waste enough memory that you can create a problem bad enough to be detectable.

Now let’s take a quick look at the PictureViewController. Notice that the view model declared a PictureViewModelDelegate. That’s used to tell a delegate that the image has been loaded successfully. The view controller implements that delegate protocol, using the modelDidLoadPicture(:) callback to load the picture into the UI:

performance/MemoryLeakDemo/MemoryLeakDemo/PictureViewController.swift

```
import UIKit

class PictureViewController: UIViewController {

    @IBOutlet var imageView: UIImageView!
    var model: PictureViewModel?

    override func viewDidLoad() {
        super.viewDidLoad()
        model = PictureViewModel()
        model?.delegate = self
        model?.loadImage()
    }
}

extension PictureViewController: PictureViewModelDelegate {
    func modelDidLoadPicture(_ model: PictureViewModel) {
        print ("modelDidLoadPicture")
        imageView.image = model.image
    }

}
```

When the view loads, the view controller creates a model, sets itself as the model's delegate, and tells the model to load its picture. In the delegate callback, it updates the UI. This looks perfectly reasonable, so go ahead and run the app.



Once the app is running, switch back to Xcode and take a look at the window. Unless you've changed Xcode's "Behavior" preferences, the left pane will automatically switch to the Debug Navigator (⌘7). As seen in the figure, this gives a bare-bones sense of the app's CPU, memory, disk, and network usage as the app runs. At launch, the app may take up somewhere around 30–40 MB; the actual amount can vary depending on the device, since the larger screens of iPads and "Plus"-size iPhones will use more RAM for their views.

Click the "Show Picture" button and notice as the new view slides in and the picture appears, the app's memory usage goes up by about 40 MB. That's to be expected, because you do have that model class holding on to both the raw data and the UIImage, plus there's a whole new view hierarchy to account for.

Now click "Back". The memory doesn't go down. *Uh oh.* Since you don't need that scene anymore, you would hope that the view controller, its model, and the image and data being held by the model would be freed. Try clicking "Show

Picture" again. The memory goes up again. Go back and forth a few times and you can waste quite a bit of memory:

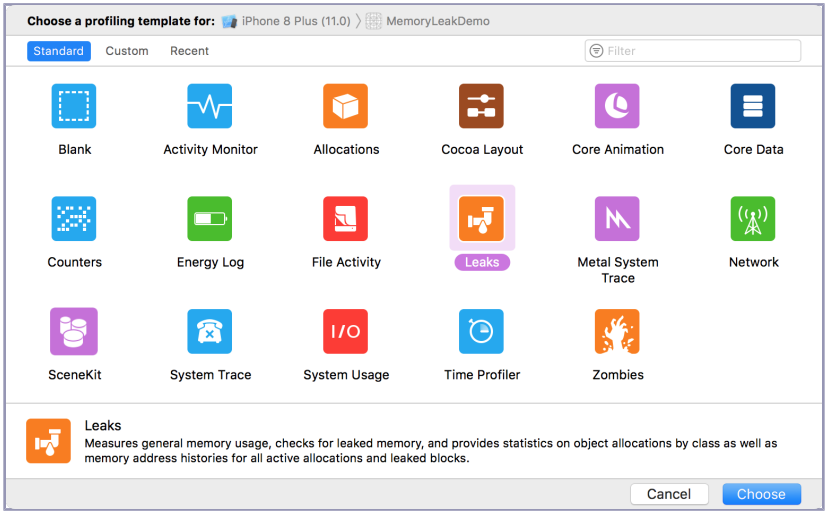
Houston, we have a problem. Some iOS 11-capable devices only have 1 GB of memory, and that's supposed to hold the system and all running apps—you could easily blow through that in less than a minute at this rate. Well, not really, since once you've used too much memory, the system will terminate the app to protect the rest of system. And this is the extreme example of how memory problems are performance problems: nothing performs worse than an app that's been killed.

Now that you know you have a problem, the next step is tooling up to *find* the problem.

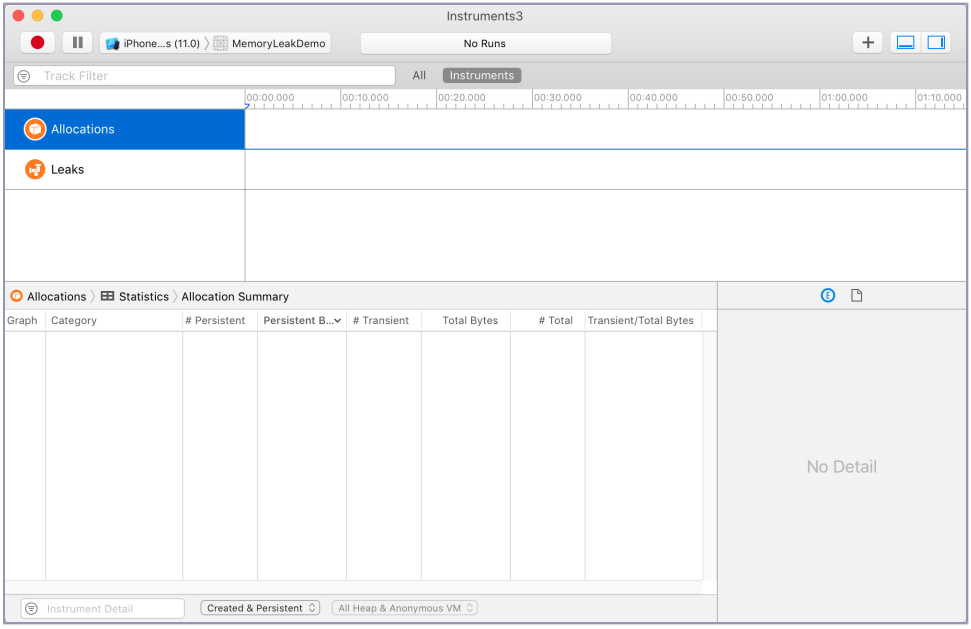
Isolating Memory Leaks with Instruments

For finding all sorts of performance problems, Xcode's go-to tool is *Instruments*. This is a separate Mac application, built into Xcode itself. Instruments is designed to profile your app as it runs, measuring how it uses resources like memory, CPU time, threads, network, and more.

Stop the app if it's still running, and then launch Instruments with menu item Product > Profile (the key command is ⌘I, which you can remember as "Instruments"). When Instruments launches, it shows a window offering icons for the various services it offers, as seen in the following figure (these are *profiling templates*, each of which combines one or more of the actual profiling instruments):



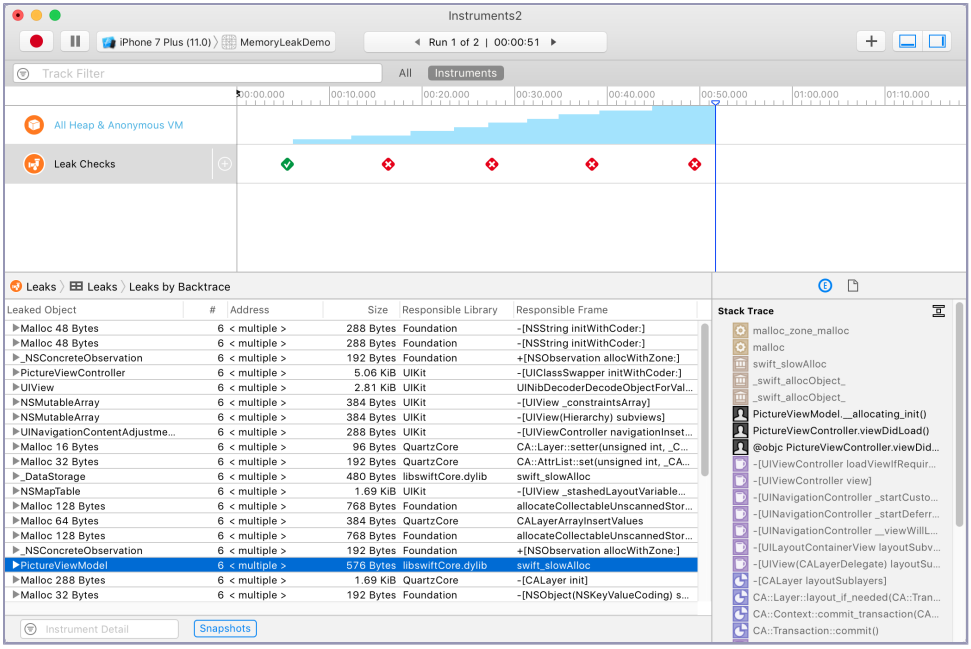
On this screen, choose the “Leaks” template, whose icon shows a droplet of water leaking from a pipe. This opens up a new Instruments window, as shown in the following figure:



The Instruments window is laid out somewhat like Xcode itself: a toolbar at the top with record/stop buttons and a scheme selector on the left, an iTunes-like status box in the middle, and show/hide buttons on the right. Below this is a timeline with two tracks, one for each instrument we’re using. Since you chose the Leaks template, you get both the Allocations instrument (which tracks all memory allocations) and the Leaks (which specifically calls out memory leaks). Finally, at the bottom, there’s a large empty table that will be filled in shortly.

Click the red “Record” button on the left end of the toolbar to launch and run the app. Once the app starts, the playhead on the timeline will start advancing and each track will fill in with a visualization of what the app was doing at that time. For example, the Allocations track will show a bar graph of overall memory use, while the Leaks track will show a green checkmark when no leaks have been detected, and a red x when leaks are found.

While Instruments is recording, switch to the Simulator or your device and interact with the app again, going forward and back, to the picture view and back. After a few times, click the Record button again to kill the app and stop recording. Your Instruments window will look like the figure on page 141.

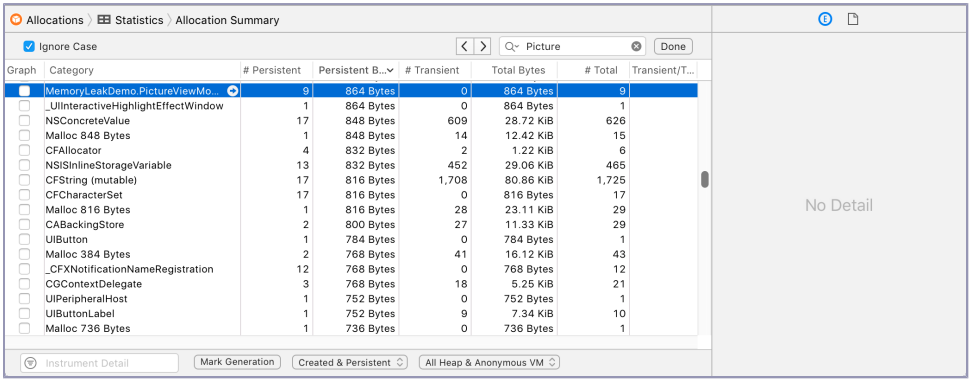


In the Allocations track, you see the dreaded “Staircase of Doom”, a memory use graph that only goes up and never down, clearly showing you have a bad memory leak. In the Leaks track, several red x’s show that you have leaks; mouse over them and a tooltip will appear showing you have dozens or even hundreds of leaks.

You can click either track to populate the bottom pane of the screen with details of that instrument. This area has a hierarchical pop-up controller at its top that works kind of like the scheme selector. Click the Leaks track, and the control will show as “Leaks > Leaks > Leaks by Backtrace”, while the table fills in with a list of every type that has leaked, how many instances have leaked, and other details.

Most of the types that have leaked are built in Foundation and UIKit classes. But you shouldn’t take that as evidence of memory leaks in iOS itself; it’s more likely these are caused by this app leaking memory, and holding on to instances of these types. So the essential strategy is: *search for your own classes,* because those are probably what’s leaking. You can scroll up and down in the table, or use Edit > Find (⌘F) to search for one of your classes by name. Search for “Picture” and sure enough, you’ll find `PictureViewController` and `PictureViewModel`. Click either and a detail pane at the bottom right shows a call-by-call stack trace of when the leaked object was created.

It can also be helpful to search for your types in the Allocations instrument. Click the Allocations track and then do a find for "Picture" in its Statistics table. This table shows the number of "Persistent" instances of the type, meaning those that are still in memory, versus "Transient" instances, which were present at one time but have since been freed.



From the looks of this figure, no instance of PictureViewController or PictureViewModel has ever been freed; they're all persistent in memory. These classes are clearly leaking.

So the next question is: *why?* Fortunately, Xcode and Instruments can help you figure that out too.

Fixing Memory Leaks

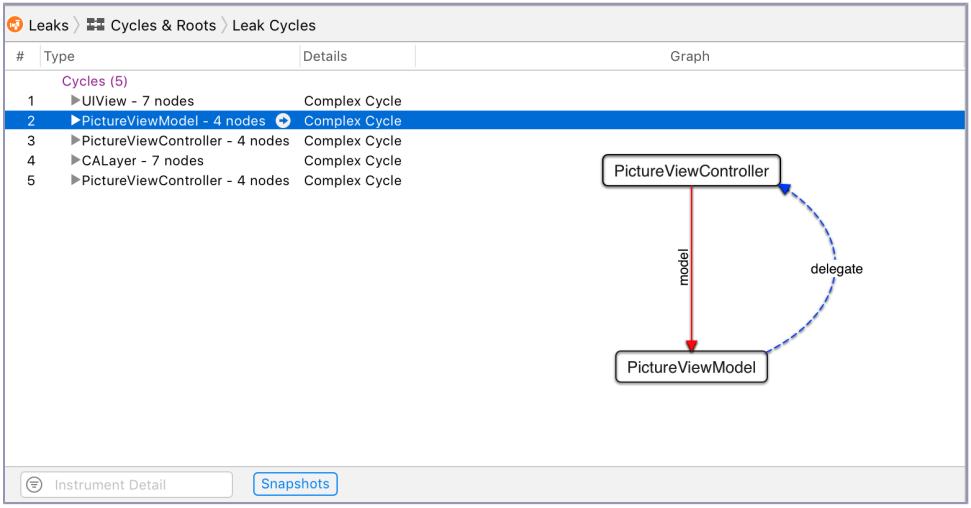
Instruments has implicated your classes as leaking memory. Now you need to figure out why. And that means thinking about what causes a memory leak in the first place.

Automatic Reference Counting (ARC) should take care of this for Swift and Objective-C reference types (i.e., classes) by keeping track of how many valid references there are to any object. Once the count of incoming references drops to 0, the object gets freed. Somehow, one or both of these objects are failing to ever get down to 0.

What should be happening is that there is one reference to the view controller: the navigation controller will have a reference to it in its stack. But that means when the user goes back in the UI, the view controller is popped off the navigation stack, and its count should then go to zero. Obviously, this isn't happening. As for the model, the only object that knows it exists is the view

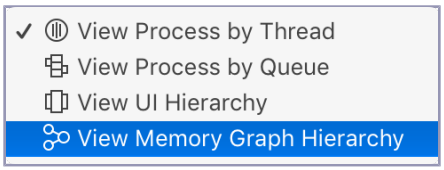
controller (which has the model as a property), so it's a good bet the model leak is a side effect of the controller leak.

Instruments can help with this. Select the Leaks instrument in the timeline, and then atop the detail area, change the second member of the Leaks > Leaks > Leaks By Backtrace control to "Cycles & Roots". This display shows cases where there are cycles of object references, rather than one-directional hierarchies. The table shows both PictureViewController and PictureViewModel. Click one, and it shows a graph of the objects' relationship, as in the following figure:

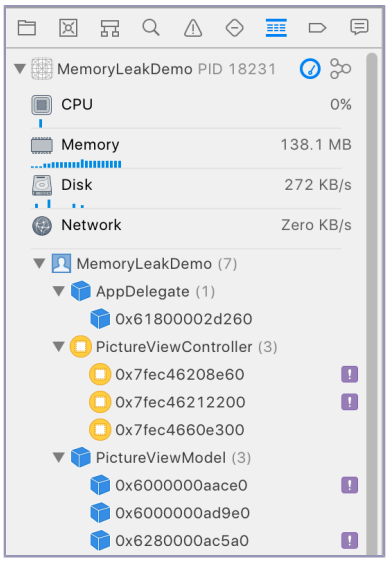


Now you have a good lead on what the problem is. Reference cycles are almost always a mistake, and here you can see what is almost certainly the problem: the PictureViewController's model property is a reference to the PictureViewModel, but the model's delegate property refers back to the view controller.

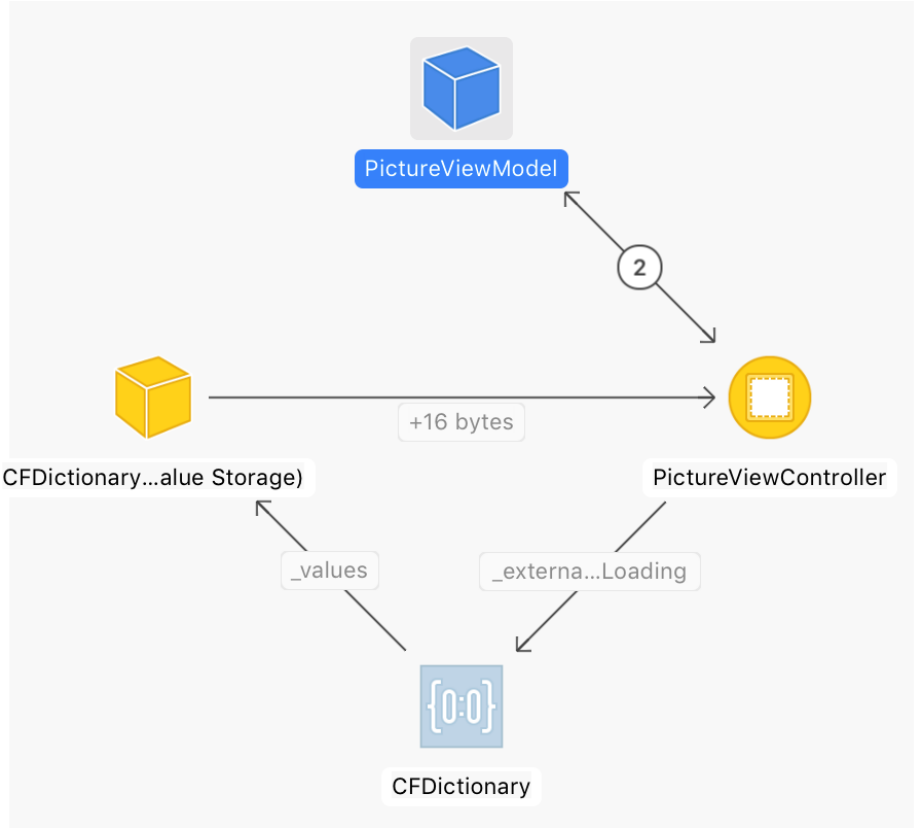
You can get an even better view of the problem back in Xcode. Run the app in Xcode, and go back and forth a few times to leak some memory. Then, in the left pane's Debug Navigator, next to where it says "MemoryLeakDemo" and a PID (process ID), notice that the icon on the far right of this line is actually a pop-up button. It starts as "View Process By Thread", but select "View Memory Graph Hierarchy" instead.



This stops the app, as if on a breakpoint (in fact, it can be resumed with the breakpoint controls in the debug area, as usual). The empty space in the Debug Navigator fills in with a hierarchical list of objects, grouped by framework or module. The first of these groups is the current app. Expand the group to see types defined by the app, such as your PictureViewController and PictureViewModel.



Each instance of an object is shown by its address in memory, and some of them have a purple exclamation point icon next to them. These indicate your leaks. Click one of them to bring up an object graph in Xcode's content pane, as seen in the following figure:



This graph shows the reference cycle between the model and the view controller, with a "2" in the line to indicate there are two references here. Click the "2", which will reveal a pop-up that names both model and delegate as the offending parties in a reference cycle, reaffirming what Instruments already tipped you off to.

So how do you fix it? The problem with a reference cycle is that the controller holds a reference to the model (keeping its reference count from ever reaching 0), and the model holds a reference back to the controller (which keeps its count above 0). One of these references has to be eliminated for the cycle to break. The way you do that is with a *weak reference*, one that refers to the other object but doesn't count toward its total reference count.

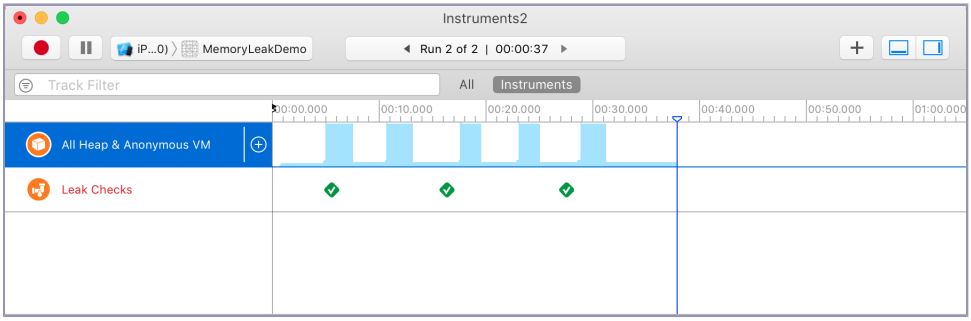
This happens a lot with delegates, and the common practice in both Swift and Objective-C is for the reference back to the delegate to be the weak one. So, in PictureViewModel.swift, you need to change the declaration of delegate as follows:

performance/MemoryLeakDemo/MemoryLeakDemo/PictureViewModel.swift
```
weak var delegate: PictureViewModelDelegate?
```

This will force you to make one other change: you can't declare an instance of the PictureViewModelDelegate to be weak if it's not necessarily a reference type (the protocol could, in theory, be implemented by extensions on an enum or struct), so you need to change the protocol declaration to say any implementation must be a class:

performance/MemoryLeakDemo/MemoryLeakDemo/PictureViewModel.swift
```
protocol PictureViewModelDelegate: class {
```

Apply these fixes and run again. Now the Debug Navigator should show the memory dropping back down when you go back to the first scene. For an even better view, try it out again in Instruments:



As seen in the preceding figure, the Allocator instrument shows memory use going up and then back down, rather than the Staircase of Doom you saw before, and the Leaks instrument is all green checkmarks. Problem solved!

How the Strong-Delegate Memory Leak Turns Up in Closures

A leak caused by forgetting to make a delegate property weak is one of the most common coding mistakes in the Apple frameworks. A Swift-specific counterpart turns up a lot when you're using completion-handler closures. Consider the following code:

```
foo.doSomething { completed in
  self.handleCompletion(completed)
}
```

In this example, the doSomething() method takes a closure, which will pass in the variable completed when it runs. The closure's implementation simply calls self's handleCompletion() method, passing in the completed parameter.

The problem is that foo will hold on to this closure until it's time to run the completion handler, and that means it'll create a strong reference to self, forcing it to stay in memory when it might otherwise have been freed (which, in turn, might have freed foo).

The fix here is two-fold. First, you need to tell the closure that in capturing the self variable, it should use a weak reference. You do that with the syntax [weak self] at the beginning of the closure. Secondly, since self is now weak inside the closure, you have to deal with it being an optional. You could use optional-chaining (e.g., self?.handleCompletion(completed)), but it's better practice to try to unwrap the optional up-front, a technique called the "Weak-Strong Dance". The corrected code looks like this:

```
foo.doSomething { [weak self] completed in
  guard let strongSelf = self else { return }
  strongSelf.handleCompletion(completed)
}
```

Some people dislike the ugly strongSelf variable, and the fact it's still possible to accidentally refer to the weak self. One variant of this technique is to call the local variable `self`, with backticks, which lets you override the usual restriction on naming a variable self.

Optimizing CPU Use

Fixing memory problems is helpful, but there are still plenty of times when the problem really is that our apps are asking the CPU to do too much in too little time. We wouldn't be buying new iPhones every couple years if we weren't grumpy about how slow our old ones had become.

But before we tell our users to do the same thing, can we be sure that we're making the most efficient possible use of the hardware they have? Are there

things we can do to make our code faster? Fortunately, Instruments can help us investigate these kinds of performance challenges too.

Building a Chroma-Key App

To figure out performance optimization, you'll need some code that does a lot of heavy lifting. To do this, the sample code download has a *chroma key* application. This is a technique used in TV and movies: you take a single color and make all instances of it transparent. If you have an image or a video of an object or actors with that color as a background, you can then put them in any scene you like by just changing out the background behind the now-transparent pixels. Sometimes this will be called a "green screen" or "blue screen" technique, because the background will be a pure green or blue, and the foreground objects and actors will be careful not to use that color.

For this exercise, I've taken a sample image of one of my anime figures and put it on a green background, as seen in the following figure (to simplify the code, I tweaked the background so that instead of a range of mostly green colors that you'd get from an actual camera image, all the background pixels are perfect greens: RGB values of `(0, 255, 0)`):

So, how *do* you create a chroma-key image? Given a source image, how do you create an image with the green pixels turned transparent? To make things easier, I made sure the incoming image is a PNG with an alpha layer. That means that when you access the raw pixel data, it will be in BGRA format, where you have one byte (8 bits) each of blue, green, red, and alpha.

The alpha is the key here: you can take that data and just copy it as the basis of a new image, except that every time you find a pure green pixel, you can just copy it over with a `0.0` alpha value, making it transparent. Make a new image from that, and boom, chroma key effect. Not so bad, right? Let's see what that code looks like:

performance/ChromaKeyDemo/ChromaKeyDemo/ViewController.swift

```
Line 1  func applyChromaKey(sourceImage: UIImage) -> UIImage? {
        -     guard let sourceCGImage = sourceImage.cgImage,
        -         let colorSpace = sourceCGImage.colorSpace,
        -         let provider = sourceCGImage.dataProvider,
        5         let pixelData = provider.data as Data? else { return nil }
        -
        -     var chromaKeyData = Data(count: pixelData.count)
        -     for pixelIndex in 0..<pixelData.count/4 {
        -         let pixelOffset = pixelIndex * 4
        10        let blue = pixelData[pixelOffset]
        -         let green = pixelData[pixelOffset + 1]
        -         let red = pixelData[pixelOffset + 2]
        -         let isChroma = isColorChroma(red: red, green: green, blue: blue)
        -
        15        chromaKeyData[pixelOffset] = blue
        -         chromaKeyData[pixelOffset + 1] = green
        -         chromaKeyData[pixelOffset + 2] = red
        -         chromaKeyData[pixelOffset + 3] = isChroma ? 0 : 255
        -
        20        print ("pixel \(pixelIndex) isChroma: \(isChroma)")
        -     }
        -
        -     let bitmapInfo = CGBitmapInfo(rawValue:
        -         sourceCGImage.bitmapInfo.rawValue |
        25        CGImageAlphaInfo.last.rawValue)
        -     let chromaKeyCFData = chromaKeyData as CFData
        -     guard let chromaKeyDataProvider =
        -         CGDataProvider(data: chromaKeyCFData),
        -         let chromaKeyCGImage =
        30        CGImage(width: sourceCGImage.width,
        -                 height: sourceCGImage.height,
        -                 bitsPerComponent: sourceCGImage.bitsPerComponent,
        -                 bitsPerPixel: sourceCGImage.bitsPerPixel,
        -                 bytesPerRow: sourceCGImage.bytesPerRow,
        35                space: colorSpace,
        -                 bitmapInfo: bitmapInfo,
        -                 provider: chromaKeyDataProvider,
        -                 decode: nil,
        -                 shouldInterpolate: false,
        40                intent: CGColorRenderingIntent.defaultIntent)
        -         else { return nil }
        -
        -     let chromaKeyImage = UIImage(cgImage: chromaKeyCGImage)
        -     return chromaKeyImage
        45 }
        -
        - private func isColorChroma(red: UInt8, green: UInt8, blue: UInt8)
        -     -> Bool {
        -     return green == 255 && red == 0 && blue == 0
        50 }
```

This code goes fairly deep into the Core Graphics woods—and this isn't a Core Graphics book—but here's a general idea of how it works. This method takes a UIImage called sourceImage, and returns a new UIImage with the green pixels turned transparent. The guard let on lines 2–5 makes sure that the image is really backed by a bitmap (called sourceCGImage) and that it can get to the image's pixel data.

If everything checks out, the loop on lines 7–21 are where the real work happens. It creates a Data to hold the modified pixel data, and then loops through all the pixels. There are four UInt8s for each pixel, arranged "BGRA", meaning blue, green, red, alpha. It copies these to corresponding indexes of the new Data, except that for the alpha, it calls a helper method isColorChroma() to see if this is a green pixel that should be made transparent. Finally, lines 23–41 are just the rather verbose means of creating a CGImage from raw bitmap data, mostly matching sourceCGImage's traits, while ensuring the new image has an alpha channel for transparency.

The sample app has a single view with two UIImageView subviews. The first is the original image with the green background, as loaded from the asset library, and the second will be the chroma-key image. Here's how you do that:

performance/ChromaKeyDemo/ChromaKeyDemo/ViewController.swift

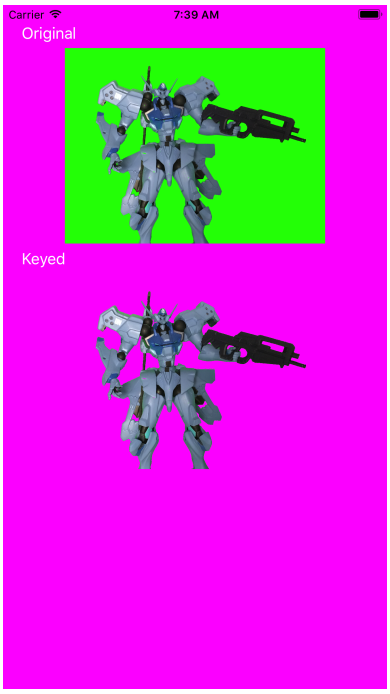
```
Line 1  override func viewDidLoad() {
2       super.viewDidLoad()
3       originalImage = UIImage(named:"tsf-pure-green")
4       originalImageView.image = originalImage
5       let inTime = NSDate()
6       let keyedImage = applyChromaKey(sourceImage: originalImage)
7       let elapsed = inTime.timeIntervalSinceNow * -1
8       print ("apply chromaKey took \(elapsed) seconds")
9       keyedImageView.image = keyedImage
10  }
```

Notice that this code provides a simple performance check here. You get the current time on line 5, make the call to applyChromaKey(), and then get the timeIntervalSinceNow() immediately afterwards. This value is in seconds, and since inTime is in the past, you have to multiply by -1 to get the elapsed time.

Anyway, this is all we need, so now we can run the app. The good news is that it works! See the figure on page 150.

```
apply chromaKey took 53.680808007717 seconds
```

The bad news is that it takes *a minute or more* to run, even in the Simulator. This code obviously has a *terrible* performance problem.



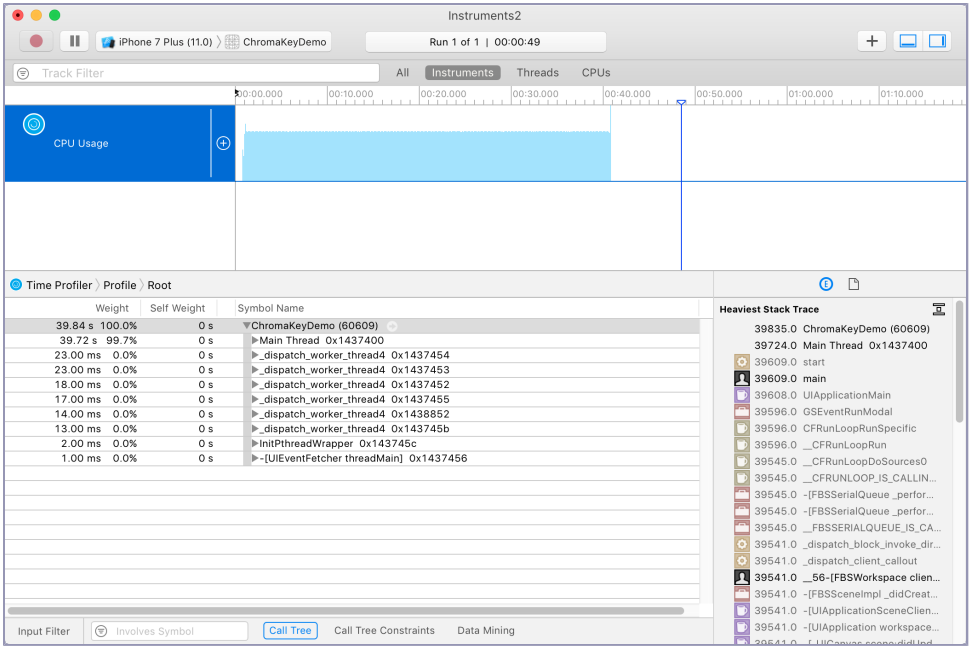
CPU Profiling

Should this really take a minute or longer? It's moving a lot of data around in memory, but still, that seems like way too long. So what's the hold up? Fortunately, Instruments can help you out.

Launch Instruments (⌘I), and this time when the icons for the profiling templates appears, choose "Time Profiler". This is an instrument that helps figure out what the app is doing with its CPU time, by inspecting every function or method call and how long it takes. It then groups these together in a hierarchy, so you can see who called whom and how much CPU time it took.

Click the Record button in the Instruments window to start a run, wait for the app to come up in the Simulator window and show the images, and then click the Stop button in Instruments. Because of how badly the sample app is written—you'll see why in a moment—Instruments may become unresponsive for several minutes while it shuts down the app and finishes up its work. But when it's ready, you'll see a display as shown in the figure on page 151.

In the timeline, you can see a visualization of how much work the CPU was doing as the app ran. As you might expect, it goes full blast for 40 to 60 seconds while it's creating the chroma-key image, then drops to nothing as it

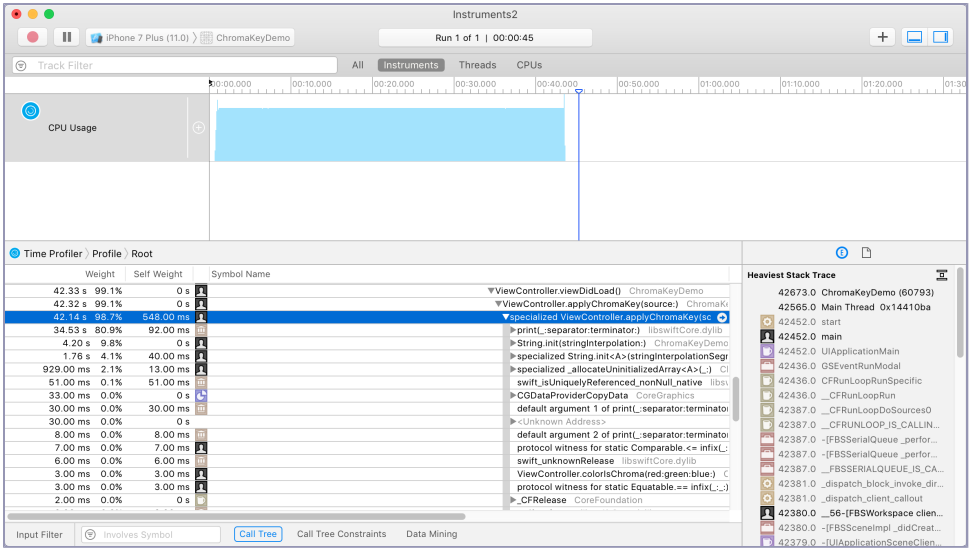


idles. That doesn't help a lot here, because the timing of the problem is so obvious, but in a complex app with many scenes, you could interact with the app at length and look for hotspots where it seemed busier than it should be.

In the bottom part of the window, there's a table called "Time Profiler > Profile > Root", that shows how much time was spent in each of ChromaKeyDemo's threads. Nearly all of it was in the "Main Thread", which figures, because that's the thread that builds the UI and does all the startup work.

Notice that this table is actually built of disclosure triangles. You can drill down into each of these threads to get more and more detail. For example, expand "Main Thread" and you can see that the internal method start() accounted for about 99% of the CPU time, and _dyld_start() almost none.

So you can keep drilling down like this, opening the disclosure triangle for whichever function or method call has the highest percentage of CPU. Or, you can cut to the chase. When a row is selected in the table, a detail pane on the right shows the "Heaviest Stack Trace", meaning the one chain of calls that took the most time. In these stacks traces, system calls are shown in disabled (gray) text, while your own calls are black. Scroll down that list and you'll find the deepest call is specialized ViewController.applyChromaKey(source:), and by clicking it, you can expand all the disclosure triangles to get to that call in the Profile table as shown in the figure on page 152.



In the tree, expand the line for applyChromaKey() to show the calls it makes and look at the next line: print(_:separator:terminator). *Whoops! I left a debugging print() statement in the sample code.* If you missed it, it's line 20 in the original code listing. Worse yet, this log message accounts for fully 80% of the runtime of the app. D'oh!

In fact, it's even worse than that. The next line down is String.init(stringInterpolation:), presumably part of the logging, taking another 10% or so. And there's another String function after that. What Instruments shows here is that the app is spending effectively *all* of its time cranking out debugging messages.

It kind of makes sense: logging messages aren't free, but more importantly, for a 3264 × 2448 image, printing a log message for every pixel means it is cranking out nearly *8 million* useless log messages.

Go back to Xcode, remove the print() statement from viewDidLoad() and run again. The minute-long delay disappears and the app comes up fairly quickly. Time profiling has found your worst code, and allowed you to eliminate it. Mission accomplished.

Rethinking the Code

Eliminating all your wasted time is a win, but is it enough? On the Simulator, the elapsed-time logging message shows that the call to applyChromaKey() takes about half a second. *But the Simulator doesn't matter.* Your users will be

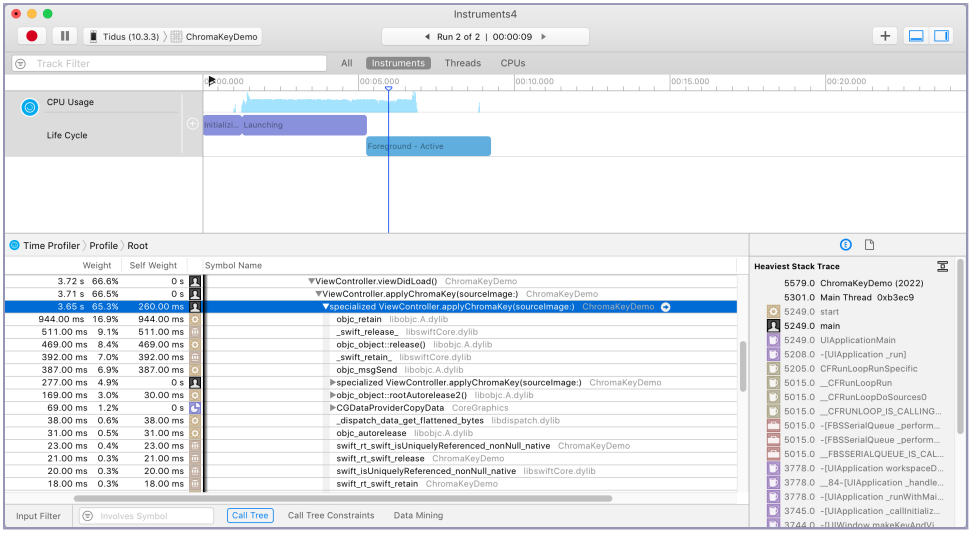
running the app on devices, so the only true test of performance is on an actual iOS device.

Run the app on a device and the story isn't nearly so rosy:

```
apply chromaKey took 5.37090402841568 seconds
```

On an iPhone 5s, applyChromaKey() takes about five to seven seconds. That's still *way* too much.

What do you do? Go back to Instruments! Fortunately, Instruments is perfectly happy to probe and profile an app running on a device instead of the Simulator. This is something you should always do, to profile how the app runs on real hardware. Again, launch Instruments (⌘I), and choose the Time Profiler template. Since the last run in Xcode was to the device, Instruments' scheme selector should show your device and not the Simulator (and you can change it at any time anyway). Click the record button to run and profile the app on your device, and stop recording once the app comes up.



The display for a time profile run on the device is a little different than the simulator, with a second track in the timeline showing when the app is launching and when it's in foreground. As before, the table at the bottom lets you drill down into where your performance hotspots are.

Again, the apps spend most of its time in applyChromaKey(), but this time, there are no easy wins to be found. In the figure, you can see that 65% of the time is spent in this method, with a significant amount of that time spent on internal Swift and Objective-C calls.

Since many of these are "retain"- and "release"-type calls, you might hypothesize that the overhead of accessing your pixels through the Swift `Data` type is costing too much. You might even consider rewriting `applyChromaKey()` in C to get rid of that overhead, since Core Graphics is a C API anyway. It sounds a little painful, but it *might* help.

But you know what? It was pretty naive to just assume walking the pixels manually was a good way to solve this problem. It works, but at the cost of iterating through a `for` loop *8 million times.* A loop like that should almost always raise a red flag, and make you ask yourself "is there a better option?"

When I tweeted a screenshot of this chapter[1] for kicks, developer Bruce Payan saw the horrible `for` loop and replied with a helpful tweet of his own[2]:

> Something tells me that the Accelerate framework would be useful… or better yet a Metal kernel shader. Need code?

Bruce brings up a couple high-performance options that aren't in everyday use, but that you should seek out when you think they might help. *Accelerate* is a collection of low-level C functions (callable from Swift) that mostly operate on large blocks of data, using CPU-specific features when available. It might be possible to rework the loop to work on big chunks of the pixel data, rather than walking it one pixel at a time. *Metal* would let you move the graphics work on to the GPU itself, which could be a boon, since GPUs are so performant, and working with pixel data is the whole point of this function in the first place!

Another option for getting the GPU to help out, but without having to learn the Metal Shader Language, is *Core Image*. This is a framework with hundreds of built-in filters to perform various manipulations on images. Apple's built-in filters are implemented in Metal, and it's also possible to add your own with Metal or OpenGL code.

For the purposes of this example, it's possible to stick with Swift and use a built-in filter. Because instead of jumping right in and doing it by hand, looking through the "Core Image Programming Guide" in Xcode's documentation would have revealed that the docs *already have sample code for a chroma-key filter.* Perfect!

Here's an `applyChromaKeyCoreImage()` method that uses the same parameter and return type as the old method, but uses Core Image instead (this also requires an `import CoreImage` at the top of the file to compile):

1. https://twitter.com/invalidname/status/888211301545234433
2. https://twitter.com/MacKallion/status/888232981789155328

performance/ChromaKeyDemo/ChromaKeyDemo/ViewController.swift

```
Line 1  fileprivate func applyChromaKeyWithCoreImage (sourceImage: UIImage)
        -> UIImage? {
        guard let sourceCGImage = sourceImage.cgImage else { return nil }
        
5       let cubeSize = 64
        let cubeSizeDivisor = Float(cubeSize - 1)
        var cubeDataArray : [Float] =
            [Float](repeating: 0,
                    count: cubeSize * cubeSize * cubeSize * 4)
10      for blueIndex in 0..<cubeSize {
            for greenIndex in 0..<cubeSize {
                for redIndex in 0..<cubeSize {
                    let index = 4 * ((blueIndex * cubeSize * cubeSize) +
                        (greenIndex * cubeSize) + redIndex)
15                  let alpha : Float =
                        greenIndex == (cubeSize - 1) &&
                            redIndex == 0 &&
                            blueIndex == 0
                            ? 0.0 : 1.0
20                  cubeDataArray[index] = Float(redIndex) /
                        cubeSizeDivisor * alpha
                    cubeDataArray[index + 1] = Float(greenIndex) /
                        cubeSizeDivisor * alpha
                    cubeDataArray[index + 2] = Float(blueIndex) /
25                      cubeSizeDivisor * alpha
                    cubeDataArray[index + 3] = alpha
                }
            }
        }
30
        let sourceCIImage = CIImage(cgImage: sourceCGImage)
        let cubeData = NSData(bytes: cubeDataArray,
                              length: cubeDataArray.count *
                                MemoryLayout<Float>.size)
35      let cubeFilterParams : [String : Any] = [
            "inputCubeDimension" : cubeSize,
            "inputCubeData" : cubeData
        ]
        let chromaCIImage = sourceCIImage.applyingFilter("CIColorCube",
40                              parameters: cubeFilterParams)
        let chromaUICIImage = UIImage(ciImage: chromaCIImage)
        
        UIGraphicsBeginImageContext(chromaUICIImage.size)
        chromaUICIImage.draw(in: CGRect(origin: CGPoint.zero,
45                                      size: chromaUICIImage.size))
        let chromaUIImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()
        return chromaUIImage
    }
```

Since Core Image needs its own book to understand—Simon J. Gladman's *Core Image for Swift [Gla16]* is a great place to start—here's just a high-level look at how this works. The trick is to use a CIColorCube filter, which maps from one set of RGB values to a set of RGBA values.

The map is like a three-dimensional "cube" where the axes are red, green, and blue, with a maximum of 64 indexes along each axis. In C, this is a three-dimensional array of floats, so you can write it like cubeData[redIndex][greenIndex][blueIndex]. Swift doesn't have multi-dimensional arrays, so lines 5–29 do the next best thing, by calculating indexes of a one-dimensional array of Float. Within this array, color values map over directly, unless it's at the index (0, 63, 0). That's the pure green value, so that color maps to a color with zero alpha, making it transparent.

Lines 31–41 create a CIImage from the source image, create the color-cube CIFilter, and then create a new CIImage that represents the image that results from applying the filter. You can create a UIImage from this, but it's a little weird because that image won't actually have a bitmap representation, just a reference to the CIImage object, which is an image-processing chain, not a bitmap. To avoid confusing callers, lines 43–48 create a genuine bitmap representation, and returns that.

So, this gives up a little code readability by using the exotic Core Image framework rather than just walking the pixels. Is it worth it? Run the app and see. On the Simulator, there's little difference (Core Image may even be slower), but on the device, it's breathtaking:

```
apply chromaKey took 1.29453003406525 seconds
```

The first time it runs, the time drops from 5–7 seconds to about 1. Better yet, subsequent runs are even faster. With the GPU spun up, your next run could be well under a second; the fastest I saw on my iPhone 5s was about 0.65 seconds. So, by using a different approach, the time spent in this function has been cut by 80% to 90%.

In practice, this could actually be even faster, because the job of setting up the filter only needs to be done once; you could stash the filter in a lazy property and reuse it on subsequent calls. There are also Core Image–specific ways to improve creating the bitmap (by keeping a CIContext around instead of drawing in the UIKit graphics context), to make it faster still.

Wrap-Up

In this chapter, you've seen how Instruments lets you find the worst offenses in your code—things you might not have realized were problems when you wrote the code, but which devour memory or CPU time when they actually run on the device. Some of the other Instruments templates help with specific technologies like Core Animation or Scene Kit, while the Allocations, Leaks, and Time Profiler instruments used here are applicable to pretty much any app.

These are powerful tools, and by watching the timeline and drilling down into details, you can achieve huge gains in the performance of your apps.

CHAPTER 8

Automated Testing

Once your app has compelling features and has been scrubbed of bugs, you'd naturally assume that part of the code is all set, and you don't need to worry about it as you develop new features. Problem is, you *do* need to worry about it. It's possible that your new code—or changes to the storyboards, resources, build logic, etc.—might create new bugs in old code.

Many developers defend against introducing new bugs by using *automated testing*: code that exercises the existing codebase to make sure it behaves as intended. If code changes or side effects cause the behavior of the old code to break, they turn up as a failed test and immediately call attention to themselves.

Xcode provides two systems for testing your code. *Unit tests* allow you to test the behavior and logic of your code, isolated from the app as a whole. *UI tests* bring up the app and perform simulated user interactions such as taps and text entry, allowing you to automate testing the user's interaction with the app. In this chapter, you'll find some helpful ways to use these features.

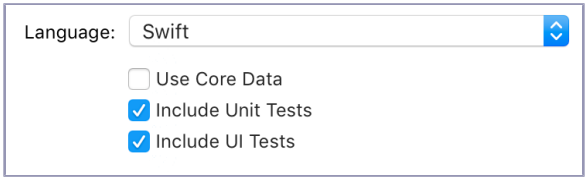
What Goes in Your Tests

Xcode uses the XCTest framework for writing tests. For unit tests, this provides assert-style methods to test for expected conditions. If these conditions are not met, the test fails. It also allows tests to wait for asynchronous results, such as long-running operations and network loads. For UI tests, XCTest provides an API to programmatically access and interact with the user interface elements.

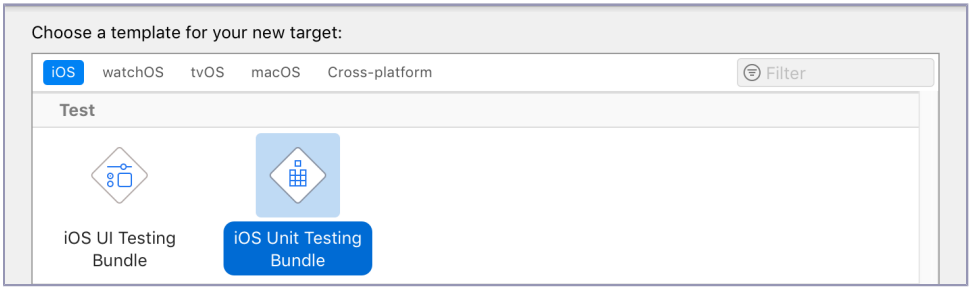
This is a book about the tool, not about the frameworks, and one could easily write a whole chapter just about XCTest. So, to learn how to write tests, see the chapter "Testing the App" in *iOS 10 SDK Development [Cla17]*.

Adding Test Targets

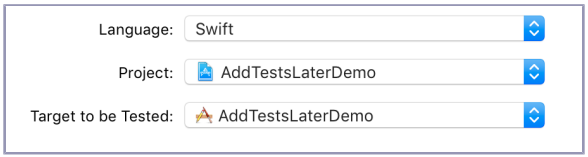
When you first create a project, Xcode asks if you want to create unit test and UI test targets to go with it. These are shown as check boxes while configuring the project (you'll also get the option to create test targets when you add frameworks, app extensions, and other kinds of code targets within the project):



But what happens if you chose not to create test targets, and then need them later? Fortunately, this is easy to deal with. Select the project from the top of the Project Navigator, and at the bottom of the Projects and Targets list, press the plus (+) button to create a new target (you can also use the menu item Editor > New Target...). This brings up the usual sheet of target templates. Making sure you've selected the right tab for your current platform (iOS, macOS, or tvOS; automated tests are not supported on watchOS), scroll down to the "Testing" group and select either the UI Testing Bundle or the Unit Testing Bundle:

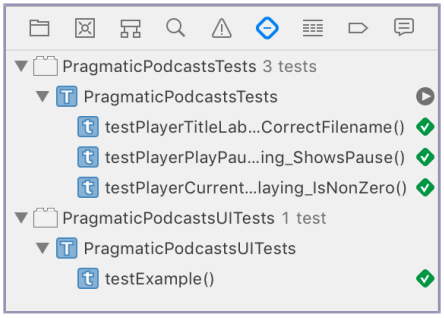


After you choose a test target type, the sheet shows you various options for your test target, including the usual metadata like a name and organization identifier, but also what language you want to write your tests in. You also need to select which project and target within the `.xcodeproj` the tests will work with. This is how you can test a framework target independently of any app or app extension targets that use it:

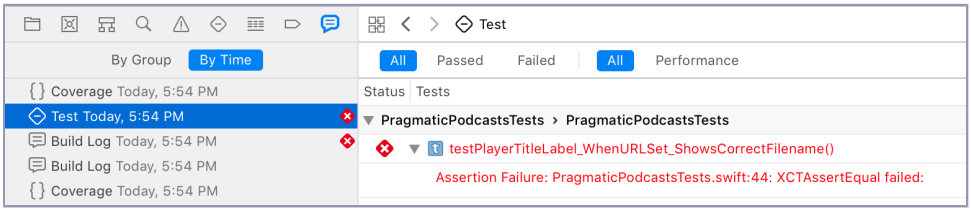


Once added, the test target will have its own Info.plist metadata file, and a single test file with a few trivial tests, and comments on how to build out the test suite further. You can run individual unit tests by clicking the diamond icon in the left gutter next to the test method name, or run all tests in a file by clicking the diamond next to the test class declaration. Usually, though, you'll create many test files, and you can run all of them with the menu item Product > Test (⌘U).

Once you have a significant number of tests, you'll probably want to stop looking so much at individual test source files and instead at the Test navigator (⌘6). This shows a hierarchy of all the test targets for the current app target, organized by test class and test methods within them. You can mouse over any test method or class to expose a run button. But usually the main purpose of this navigator is to review test results, particularly if you ran all of the tests with ⌘U. Passed tests will show a green diamond icon with a white checkmark, failed tests will show a red diamond with an X. The following figure shows a small group of tests that have all passed:



If a test fails, just double-clicking the failed test in the Test navigator won't show you the reason for the failure; instead, it takes you to that test's source. The source editor will show an error overlay with a summary of the error, but the message often gets truncated. If you don't understand why a test failed, you need to go to the Report Navigator (⌘9), and in the list of reports (which includes build logs, test logs, and run logs), click the most recent test. As seen in the figure, this shows the tests in the editor area, and lets you expand failed tests to see a full log of the problem:

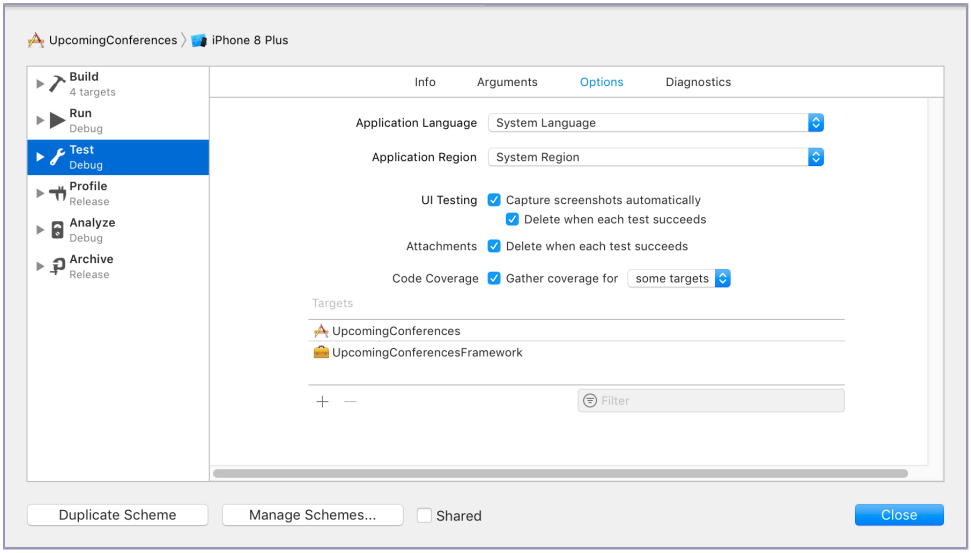


Viewing Code Coverage

You might have an intuitive sense of what code needs the most automated testing, but as your projects grow, it's possible you'll overlook important parts of the code that should be tested. Helper methods, convenience initializers, extensions on existing types… there's lots of code you can bang out quickly to get a feature done, and not really realize its implications for unit testing.

Xcode's tool to help with this is a UI to show *code coverage*: how much of the codebase is exposed to automated testing.

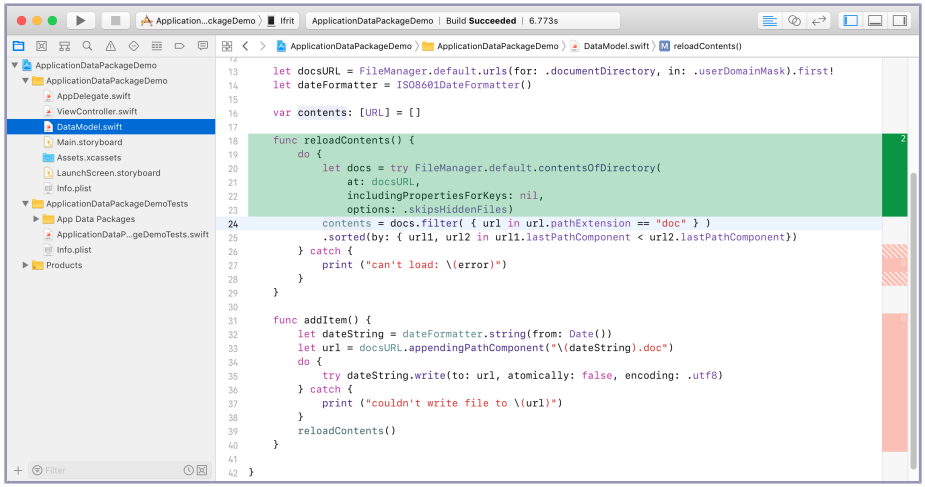
Code coverage is *not* enabled by default. To use it, you need to edit the current scheme, select the Test action, go to the Options tab, and select the "Code Coverage" check box. By default, Code Coverage will be evaluated for all your targets; if you don't want to collect or show it for certain targets, use the pop-up to select "some targets", and then add the targets to the table with the plus (+) button, as seen in the following figure:



Once you run unit tests after enabling code coverage, a visual representation of the coverage is shown in the right gutter of the source code editor. The gutter has red regions denoting code sections that are not hit by tests, and green showing those that are covered. Uncovered areas are always shown, while covered areas only appear as you mouse over the right gutter.

There's also a number that shows how many times each section of code is covered by the tests. This helps explain how the code coverage tool actually works under the hood. It's not doing some fancy LLVM profiling to analyze

the code and figure out what tests call what code; it simply counts up which lines of code get executed during the actual test runs:



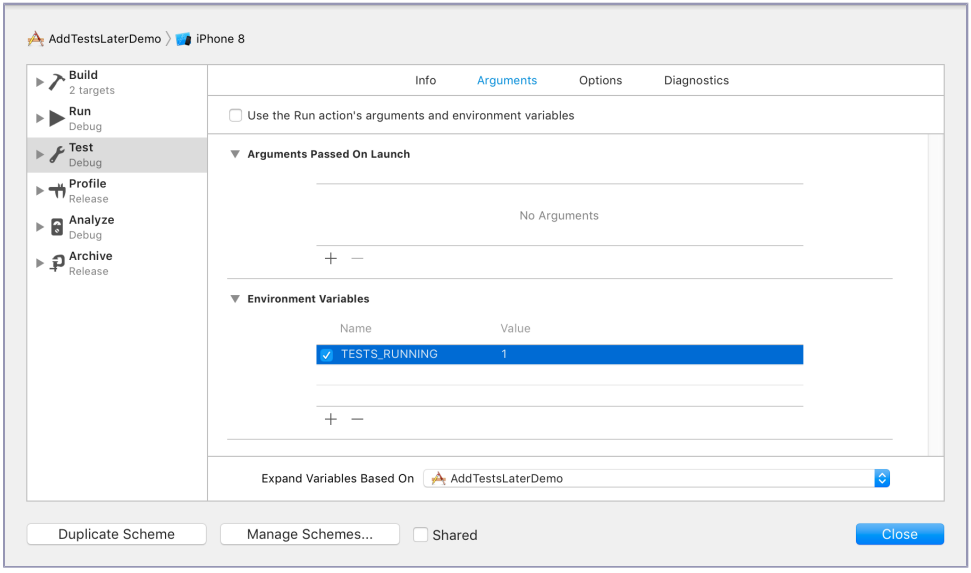
This also illuminates one important side effect of how the code coverage tool works: since the app itself is brought up to a normal running state before any of the tests are run, all code in your normal startup path is indicated as having been tested. That means a completely empty iOS app project will still show the AppDelegate's application(_:didFinishLaunchingWithOptions:) and the ViewController's viewDidLoad() as being covered by tests, simply because the test runner will *have to* run these methods as part of starting up the app, even though they're not actually being tested yet.

Using Environment Variables in Tests

If you find it surprising that the test runner brings your app up all the way just to run a single test—possibly in a class that would work just fine without the rest of the app—that's actually a good thing to be aware of. Chances are, if you have a large, complex app, your startup flow involves a lot of work that you might not want to run during tests. In fact, if the app uses the network, there's a good chance the tests won't run without a network connection. In effect, this behavior may make your test suite more of an end-to-end integration test than you ever intended, particularly when the whole point of unit testing is to test small pieces of code in isolation.

One way to deal with this is to identify parts of your startup flow that don't need to be performed when you're unit testing. Find a way to make it to your first view without checking for a login token, downloading updates, or doing other expensive tasks. Those are still worth testing, of course, but in isolation… not as a prerequisite to every unit test you'll ever run.

Once you've developed a minimal path through your launch flow, you'll need a way to execute it only when you're running tests, and not in normal app launch conditions. To do this, go to the scheme selector and choose the "Test" action, as shown in the figure. Under the "Arguments" tab, you can set either command-line arguments, or environment variables. To start, you'll need to de-select the "Use the Run action's arguments and environment variables". Then, use the plus (+) to create an environment variable with an unambiguous name, like TESTS_RUNNING or I_CAN_HAZ_MINIMAL_STARTUP_PATH:



Then, in your AppDelegate's application(_:didFinishLaunchingWithOptions:), and in any other startup-path code, just look for that environment variable. In Swift, the ProcessInfo class has a processInfo singleton object representing the current process, and the object has a dictionary named environment with all the environment variables. So your check basically looks like this:

testing/AddTestsLaterDemo/AddTestsLaterDemo/AppDelegate.swift

```
if let _ = ProcessInfo.processInfo.environment["TESTS_RUNNING"] {
  // avoid expensive startup tasks
} else {
  // normal app launch
}
```

In Objective-C, you'd look for the existence of the environment variable with [[[NSProcess info] processInfo] environment] objectForKey:@"TESTS_RUNNING"] and verify it's not nil. Either way, it's a simple check to see whether you should or shouldn't perform normal but expensive startup options.

Running Tests on the Command Line

One thing that may have occurred to you is that all the tests run up to this point have been performed in the Xcode GUI, either by clicking gutter buttons in the editor view, run buttons in the Test navigator, or with the menu item Product > Test (⌘U). That's all well and good, but it's obviously inadequate if you want your tests to be part of a continuous integration (CI) process, one which automatically runs tests for you, either periodically (like every night), or maybe every time new code is checked in.

Fortunately, it's simple to have Xcode run tests from the command line or in a shell script. It's actually a lot like automated building of your project, which you saw way back in *Building on the Command Line*, on page 104. In fact, it uses the same command, xcodebuild. The difference is that xcodebuild implicitly takes an action, which defaults to build. To run tests, you just need to use xcodebuild test.

Except… it's not actually that simple. The test action doesn't settle into sensible defaults, so just using xcodebuild test doesn't work:

```
⇛ xcodebuild test
❮ xcodebuild: error: The test action requires that the name of a scheme in the
  project is provided using the "-scheme" option. The "-list" option can be used
  to find the names of the schemes in the project.
```

The hint is nice, as it tells you that xcodebuild -list will give you the name of all the schemes, one of which you'll include with the command next time. As it turns out, that's only half the story…

```
⇛ xcodebuild -scheme AddTestsLaterDemo test
❮ xcodebuild: error: Failed to build project AddTestsLaterDemo with
  scheme AddTestsLaterDemo.
      Reason: A build only device cannot be used to run this target.
```

Right. So you need to not only provide a scheme, but also specify either an actual iOS device by name, or the identifier for one of the installed simulators. You can use the action showdestinations to show strings for all the available simulators, and then pick the one to run on:

```
⇛ xcodebuild -scheme AddTestsLaterDemo -showdestinations
```

Notice that you have to include which scheme you're interested in, since you may have some simulators that are incapable of running a given scheme (maybe your app is iPhone- or iPad-only, or you have simulators older than the app's minimum supported iOS version). The output of this command is a list of simulators, which are identified by id, OS version, and name. It's a long list, so here's just a clip:

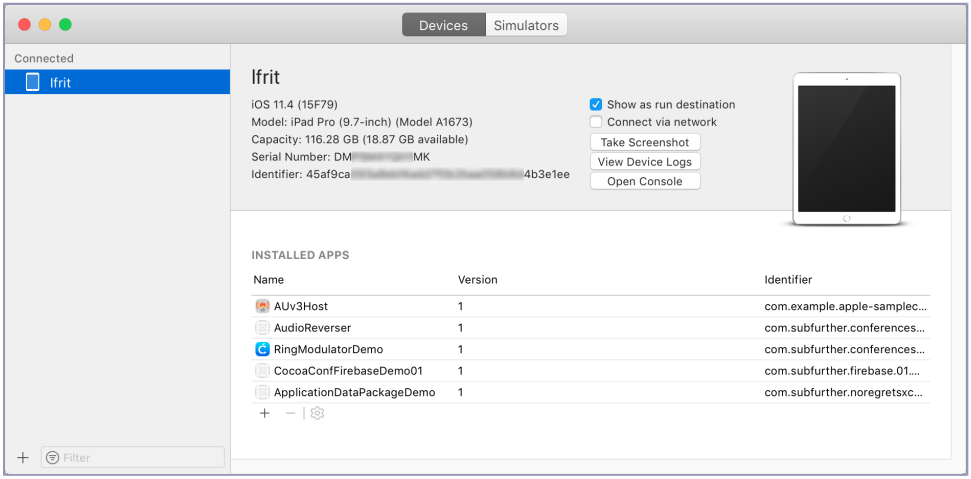
```
❮ Available destinations for the "AddTestsLaterDemo" scheme:
    { platform:iOS Simulator, id:A1EF4A02-5A4E-4B88-8D11-E931C8FD7809,
      OS:11.4, name:iPad (5th generation) }
    { platform:iOS Simulator, id:68D194AF-D16A-4A5E-95A2-70748375DA07,
      OS:11.4, name:iPad Air }
    { platform:iOS Simulator, id:9BF0DC53-036E-4A40-A19C-E2A62A3C044D,
      OS:11.4, name:iPad Air 2 }
    { platform:iOS Simulator, id:DB0B5527-0155-4C80-89B1-1FDE004A11E6,
      OS:11.4, name:iPad Pro (9.7-inch) }
```

The way to compose the `-destination` option is to provide a set of criteria that `xcodebuild` can match. You could use the `id` string, but it differs from computer to computer, so that approach won't work for teams. Instead, you should compose your destination as a combination of platform, OS, and device name, like the following command (which has added a backslash line-continuation character to fit the book's limited width):

```
⇛ xcodebuild -scheme AddTestsLaterDemo -destination \
⇛     platform='iOS Simulator',OS=11.4,name='iPhone 8' test
```

This will run the tests on the specified simulator. If all of the tests pass, `xcodebuild` returns 0, like any other command-line process does when it ends normally. On the other hand, if any tests fail, the return value will be non-zero (usually `65`), so you can watch for that in build scripts and CI systems.

What if you want to run your tests on a device? That's when you use the `id`, since now you're talking about one specific iOS device, as opposed to one instance of a simulator. You can get the device identifier by bringing up the Devices and Simulators window (either from the Window menu or with ⇧⌘2), as seen in the figure. Plug in your device via USB, select it from the Devices tab, and copy-and-paste the "Identifier" string:



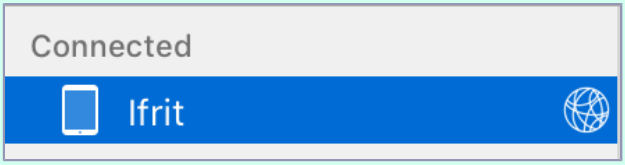
With the device identifier in the clipboard, you can paste it into the Terminal as you enter the xcodebuild command, which will look like this:

```
⇛ xcodebuild -scheme AddTestsLaterDemo -destination id="YOUR_DEVICE_ID" test
```

Where We're Going, We Don't Need… Cords

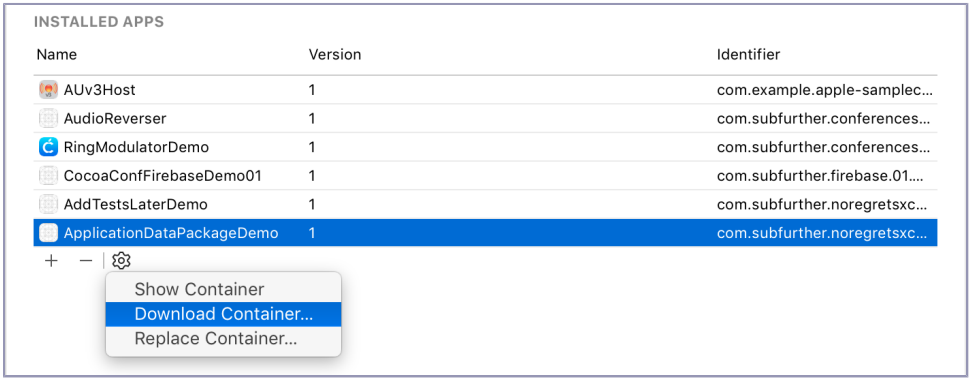
One really cool feature is hiding here in the Devices and Simulators window: the check box marked "Connect via network". Selecting it allows Xcode to run, test, and debug the app over Wi-Fi, without being connected via a USB cable.

Once you enable Connect via network, the device can be unplugged and will still appear in the Devices and Simulators window and the scheme selector—as long as it's within Wi-Fi range—and will have a little globe icon to indicate it is connected wirelessly.



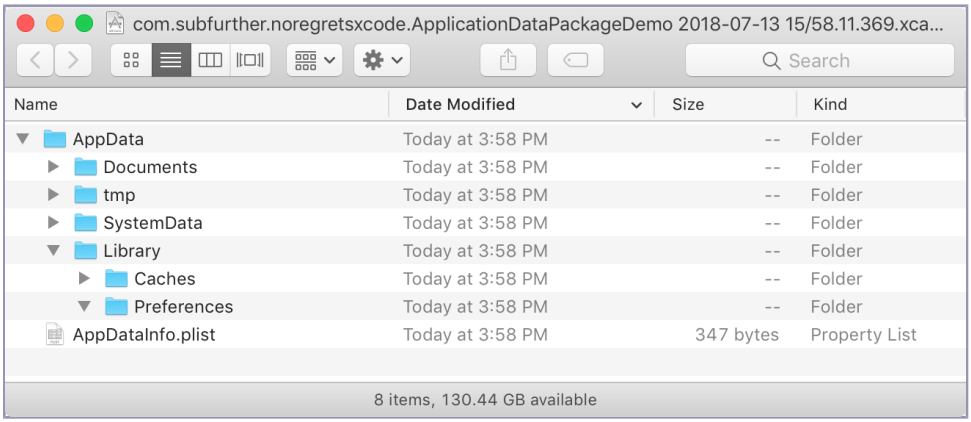
Packaging App Data for Tests

The Devices and Simulators window has a few other neat tricks up its sleeve. When you select a device, a table at the bottom of the window shows all your developer-installed apps. At the bottom of the table, there is a gear button that shows a pop-up menu with commands to show, download, and replace the app's "container", as seen here:



This feature provides access to your apps' filesystem as it exists on the device. To try it out, choose "Download Container…" and choose a destination on

your Mac's filesystem. This saves a file of type .xcappdata. This file is the app's bundle, so you can explore it in the Finder with "Show Package Contents":



The contents are an AppDataInfo.plist with metadata about the app and when you retrieved the app data, and an AppData folder. AppData is the filesystem as visible to your app at runtime: there is a Documents folder for long-term storage, the Library folder with its Caches and Preferences sub-folders, and so on.

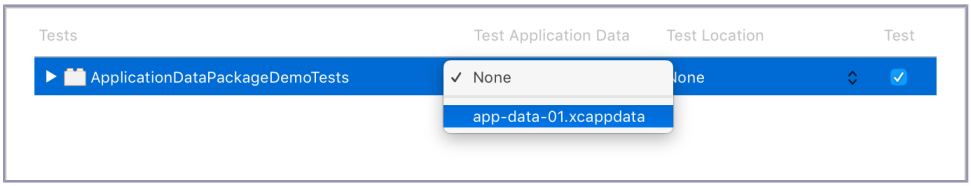
This is useful enough when debugging—allowing you to get files from a misbehaving device—but it also has a surprising use for testing. Imagine if you have code that works with local files, such as user-saved documents, or a Core Data database that gets populated by regular use of the app. To test those features, you'd need data to test on, which can be difficult to populate from within the test runner.

Instead, Xcode lets you use a dumped app data file as the basis of your tests. The basic idea is that you save the app data bundle as part of your Xcode project, and the test runner re-installs the data onto the device as the app's local files, prior to running tests:

To do this, add the .xcappdata file to your project, alongside your other test files. You may even want to create a group for app data, distinct from the test classes. Go to the File inspector (⌥⇧⌘1) and make sure that it's not included in any targets; after all, you don't want to compile this file or copy it to an app bundle, you just want the project to know about it, and presumably keep it under source control with the rest of the project.

Go to the scheme selector again, edit the scheme, select the Test action, and go to its "Info" tab. The table in this tab shows all the test bundles for the current target, and one of the columns for each bundle is "Test Application Data". This is usually set to "None", but by selecting your app data file (as

shown in the figure), you're telling the test runner to install this data into the app prior to running tests.



Running with "canned" data like this allows you to write tests that target specific contents of the local filesystem, like documents or database entries. Just run the app enough times to populate it with interesting local data, grab the app data bundle, and set it up as your test data.

App Data Bundles Are Device-Only?

Notice that you cannot create an app data bundle from an app installed to the iOS Simulator, since the Simulators tab of the Devices and Simulators window doesn't even show your installed apps. Moreover, while it seems like it should work to install test app data from a device to a simulator, it appears that it only works when running tests on device. So if you're going to use this approach, you may want to put your canned-data tests in their own test bundle, so you can keep all your other tests running in both simulator and devices.

Wrap-Up

This chapter has shown off some of the interesting ways that Xcode supports unit testing. After you've added test targets—either when creating your project or by adding testing bundles later—you can run specific tests from the source editor or from the test navigator, or just run all your tests with Product > Test. Once you've run tests from inside Xcode, the code coverage tool gives you a visual representation in the source editor of how much (or how little!) of your code is exposed to tests.

You can also run tests from the command line or from a shell script, which is essential for configuring full-on continuous integration systems for your app. And for consistency, you can use app data bundles to pre-load an app with local files prior to running tests on it.

Testing keeps your app running, but apps also need to be kept secure from external threats you might not have even thought of. In the next chapter, you'll see how those systems work, and the steps you need to take to hold up your end of the arrangement.

CHAPTER 9

Security

You remember that one time when Apple was totally pwned by hackers?

Probably not, right? As of early 2018, it appears that neither Apple or any of its devices have ever been victimized by widespread hacking efforts. There was a scandal with some sensitive celebrity photos obtained from iCloud in 2014, but that turned out to be the result of phishing attacks, not a security weakness in iCloud. Dig in to other stories on the web and they often turn out to only affect jailbroken devices, or they have other non-technical explanations.

It's kind of remarkable that Apple's stuff has held up, considering what an attractive target they must be: hundreds of millions of credit cards stored in the iTunes Store, over a billion devices connected to the internet, and the personal and financial data of all its users, stored on devices and in iCloud.

It's a credit to Apple's focus on security, protecting users' privacy and personal information that their defenses have held up over the years. The focus on security is pervasive throughout the Apple platforms, and as a developer, you can't help but run up against them sometimes.

In this chapter, you'll see how Xcode manages the security of your apps. You'll see how security pervades your use of the platform APIs, and the role it plays in putting your apps onto devices and submitting them to the App Store.

Sandboxing and Entitlements

As the legacy computing platform, macOS is a little more hands-off with security than iOS, watchOS, or tvOS. Users have always expected greater latitude running apps on their computers and have been willing to accept a little more risk. Even then, Mac apps are subject to security restrictions that protect the user from malicious code. Yes, even your code.

You can see this when you write even the most trivial Mac app. By default, new Mac app projects in Xcode are *sandboxed*, meaning they're subject to a restrictive set of rules about what they can and can't do.

To see sandboxing in action, consider the following method, which writes the string `Foo` to a file named `foo.txt` in the user's `Downloads` folder:

security/SandboxEntitlementsDemo/SandboxEntitlementsDemo/AppDelegate.swift

```
private func writeToDownloadsFolder() {
  let fileManager = FileManager.default
  guard let downloadsURL =
    fileManager.urls(for: .downloadsDirectory,
                     in: .userDomainMask).first else {
      print ("can't find downloads ƒ")
      return
  }
  let outFile = downloadsURL.appendingPathComponent("foo.txt")
  do {
    try "Foo".write(to: outFile, atomically: false, encoding: .utf8)
  } catch {
    print ("write failed, error: \(error)")
  }
}
```

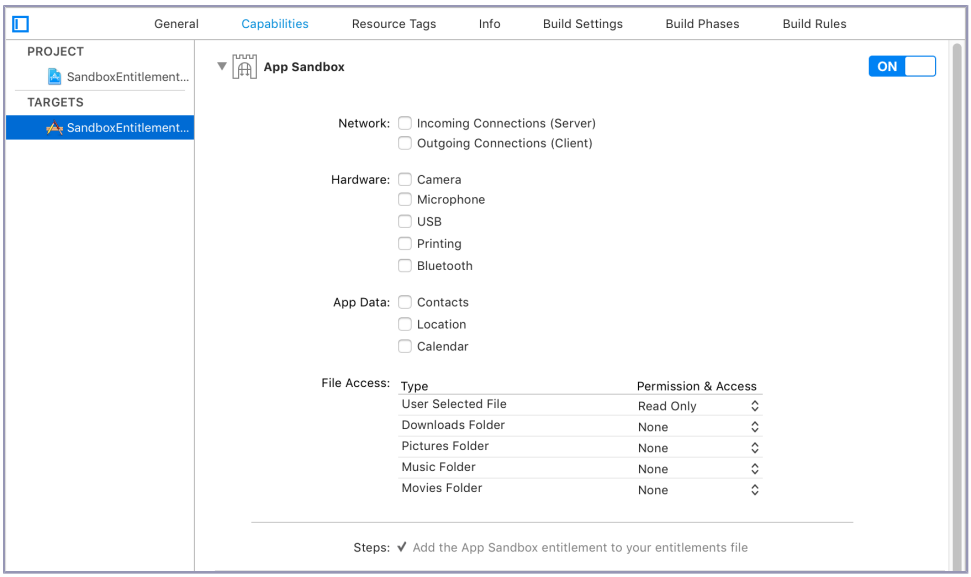
Thing is, when you run this, *it doesn't actually work.* Instead, a message like the following appears in the Console:

```
write failed, error: Error Domain=NSCocoaErrorDomain Code=513 "You don't
 have permission to save the file "foo.txt" in the folder "Downloads"."
UserInfo={NSFilePath=/Users/cadamson/Library/Containers/
 com.subfurther.noregretsxcode.SandboxEntitlementsDemo/Data/Downloads/foo.txt,
 NSUnderlyingError=0x6100004400c0 {Error Domain=NSPOSIXErrorDomain Code=1
 "Operation not permitted"}}
```

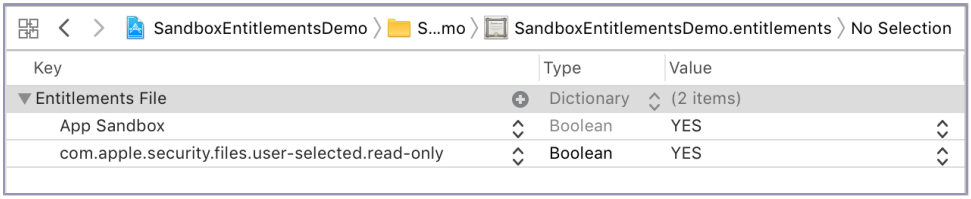
The error explains that the operation isn't permitted. But how do you get permission? To find out, select the project in the File Navigator and click the "Capabilities" tab. This shows a list of features of which your app can take advantage. The first one, as seen in the figure on page 173, is "App Sandbox" and it's turned on by default.

Every check box is a feature that sandboxing denies you unless you specifically enable it for your app by selecting it. Looking at the list, this means you can't make network connections, use microphones and cameras, access the contacts or calendar databases, etc., unless you specifically enable them.

Notice the "Steps" at the bottom of the pane. It reads, "Add the App Sandbox entitlement to your entitlements file." It's checked, so this means it's something



that Xcode has already done for you. Look in the File Navigator and you'll see a file named SandboxEntitlementsDemo.entitlements (or whatever your target is named):



This is a familiar property list editor, showing two properties: app sandbox is turned on, and the property com.apple.security.files.user-selected.read-only means you can only access files the user has deliberately exposed to your app (through user interface actions like drag-and-drop or using the NSOpenPanel)—even then, they're read-only.

Switch back to the Capabilities tab. Notice that "File Access" is a table, and one of its lines is "Downloads Folder". Switch that to "Read/Write" and Xcode will write a new line to the entitlements file: com.apple.security.files.user-selected.read-only. More importantly, if you run the app now, it will succeed.

Notice there are only a few pre-defined folders you can enable with entitlements. Beyond this, most folders are off-limits; for example, there's no entitlement that lets you write to the Desktop folder unless the user chooses to do so with a save sheet and the user-selected entitlement mentioned earlier is

set to read/write. There's much more information about the app sandbox in the "App Sandbox Design Guide" in Apple's developer documentation.

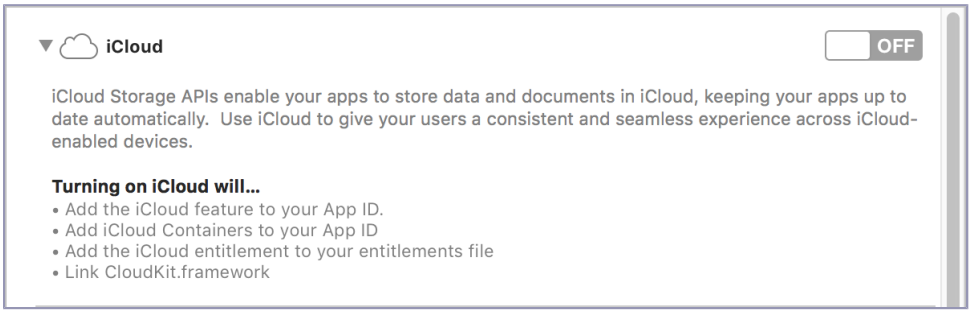
The important thing to take away from this example is that the entitlements file specifies the things your app can and cannot do. In many cases, the Capabilities tab will create entries in the entitlements file for you, however, there are some entitlements that are not exposed by Xcode; you'll have to enter those into the entitlements file manually.

App IDs and Your Developer Account

You may think from the previous example that the Capabilities tab is little more than a fancy editor for the entitlements file. But it turns out, Capabilities go a lot farther than that. If you expand some of the other capabilities, you'll see they have more steps than simply adding entitlements for you.

This is even more pronounced with projects for iOS, tvOS, and watchOS where there's a tighter connection to the App Store and Apple-provided services. For example, consider an app that uses iCloud. The book's download code has a sample app that uses iCloud to log the last time it was launched. Then, when you launch it, it shows the last time it was launched on any of your devices.

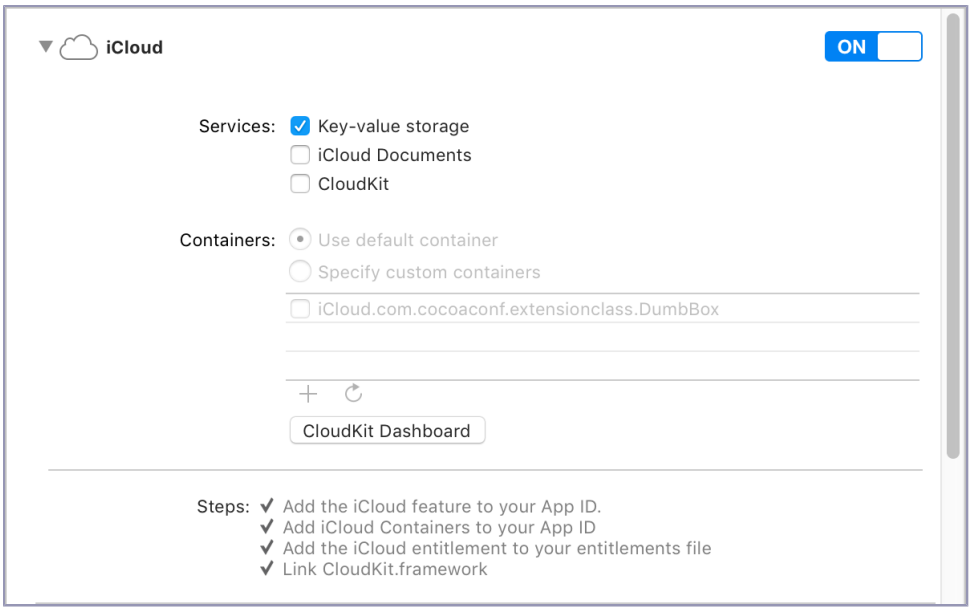
For an iCloud app like this to work, you need to enable iCloud from the Capabilities tab. The following figure shows the default state of that capability:



This time there are four steps, only one of which is an entitlement:

- Add the iCloud feature to your App ID.
- Add iCloud Containers to your App ID.
- Add the iCloud entitlement to your entitlements file.
- Link CloudKit.framework.

Setting aside the question of what an "App ID" is for a moment, go ahead and flip the switch to "On". This causes the UI for this capability to change radically, as seen in the figure on page 175.



You now have check boxes to select which iCloud features you want to use (key-value store, documents, and CloudKit) along with iCloud-specific UIs to pick an iCloud container for your cloud documents; there's also a button to configure CloudKit. The sample code project only uses key-value store, so you don't need to worry about the rest of this.

But what about the "App ID" mentioned before? The idea of the *App ID* is to uniquely identify an app you write. Obviously, Apple needs to know what apps are using its services like iCloud storage or Apple Pay; using a unique ID for each app is how they do it.

To view and manage your App IDs, you need to visit your developer account on https://developer.apple.com/account. Go to the section titled "Certificates, Identifiers, and Profiles". The "Identifiers" section is where App IDs live (along with other kinds of identifiers, like those used for Wallet items and push notifications). Click the "App IDs" list item to show your App IDs.

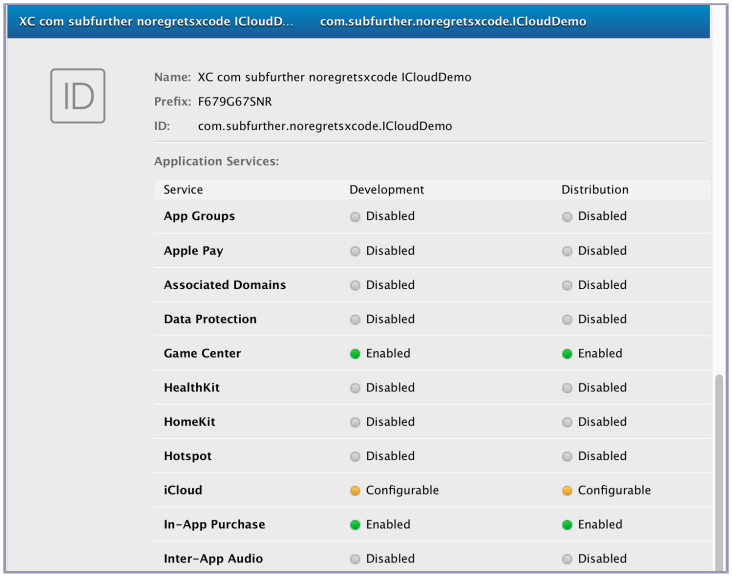
The table of Apple IDs shows a name in the left column, and a bundle identifier (from when you set up the project in Xcode) in the right. The surprising thing is that you already have an entry for your app here:

XC com subfurther noregretsxcode ICloudD…	com.subfurther.noregretsxcode.ICloudDemo

The name is programmatically generated from your bundle identifier, and starts with either `Xcode iOS App ID` or simply `XC`. This is what the Capabilities tab was talking about in its list of steps: by turning on the iCloud capability,

Xcode logged into the developer site and created this App ID for you. Pretty freaky, right?

Click the row and it will expand out to show all the features currently used by the App ID, as seen in the figure. There's also an "Edit" button way at the bottom to let you add or configure these features on another page, or rename the App ID to something other than Xcode's default:



You don't need to do anything else here for a simple iCloud app, other than to just understand that this is how App IDs work: if you're going to use one or more Apple-provided services, your App needs to have an App ID. In addition, you'll also need an App ID to distribute an app through any of the App Stores, so even if you weren't using any of these services, you'd need to set up an App ID manually on this page.

Automatic Code Signing

While you're still in the Certificates, Identifiers, and Profiles section of your developer account, you might pause to wonder: "so what are certificates and profiles?" It turns out, these are the other two major pieces of the app security story. The difference is that while the entitlements file and App ID are about what your app is and what it can do, certificates and profiles are all about *you*.

Certificates identify you as a developer who's known to Apple. They actually work with the Keychain back on your Mac. The idea is that there's a chain of trust: your certificate is digitally signed by a certificate issued by Apple World Wide Developer Relations, and that certificate is signed by Apple's root

Certificate Authority (CA). Since all Apple devices trust the Apple root CA, they trust the WWDR certificate, and any certificate it signs, including yours.

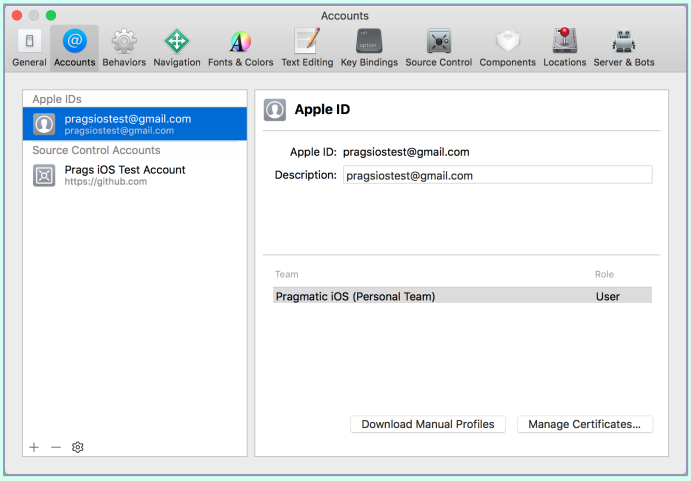
This "chain of trust" works the same way as SSL certificates for https websites. The big difference is that rather than just comparing certificates with a web server, Xcode actually uses the certificate to digitally *sign* your app, producing a file that could only have come from you.

Certificates identify you, but in and of themselves, that's all they do. *Provisioning profiles* are the documents that let you actually do stuff. There are two kinds of profiles:

- *Development profiles* allow you to install and run apps from Xcode to your own devices. In this case, your development certificate identifies you as someone known to Apple as a developer, and the profile contains a list of devices you're allowed to run on.
- *Distribution profiles* let you submit apps to Apple for sale on the store. These use your distribution certificate (if you have one, since it's part of the paid Apple Developer Program) to identify you as someone approved to submit apps to the App Store for review.

Sign In to Your Developer Account

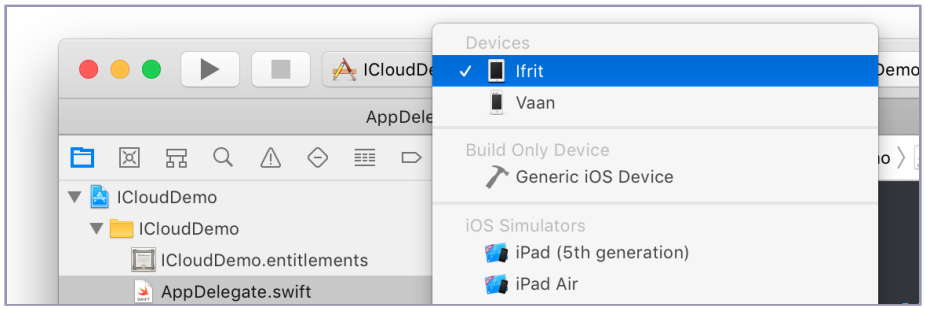
Since certificates are based on your Apple ID, from here on out you'll need to be signed into your developer account if you aren't already. Go to Xcode's Preferences, select the account tab, and look at the left pane. There should be a section that reads "Apple ID" with your logged-in Apple ID. If not, click the plus (+) button at the bottom of the window to add your Apple ID account to Xcode.



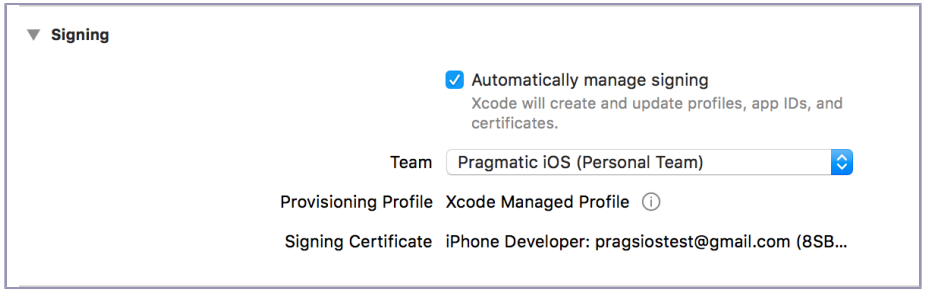
In early versions of Xcode, certificates and provisioning files had to be downloaded from the site and installed into Xcode in order to work. Starting with Xcode 8, and steadily improved since then, there's now an automatic signing system that's enabled by default for new projects. This system will create any needed certificates and profiles on the site and download them into Xcode.

Automatic Code Signing for Development

With automatic code signing, you can usually run apps on an iOS device simply by connecting it via USB to your Mac and selecting the device from the scheme selector. In this figure, two devices are connected—an iPad named "Ifrit" and an iPhone named "Vaan" (yes, I have a *Final Fantasy*-themed system for naming all my devices):



Before you try to run on the device for the first time, select the project file in the File Navigator and go to the "General" tab. In the section titled "Signing", you'll see that your project defaults to "Automatically manage signing". There's an entry for a "Team" that should show your name as a "Personal Team" if you created the project yourself; with someone else's project, it will show "Unknown" or "None" and you'll have to select your own name manually. The following figure shows how signing has been set to my "Pragmatic iOS" personal team:

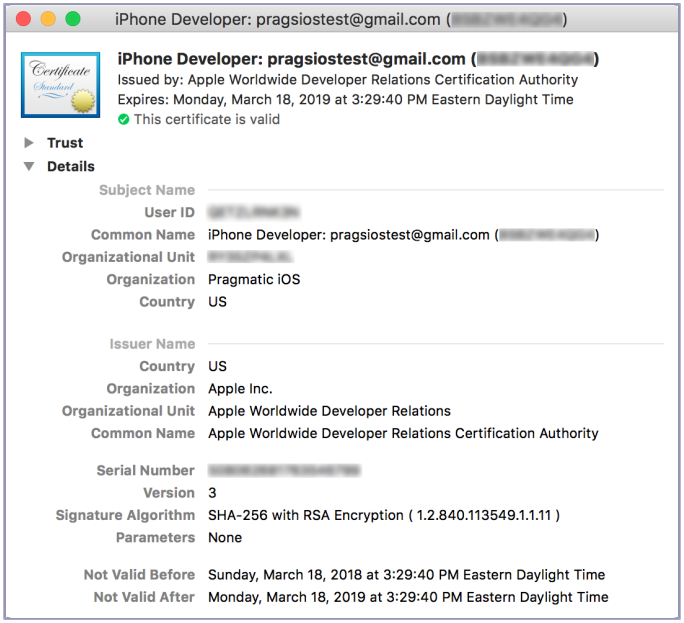


Once you select a team, Xcode will start trying to fix any issues that would prevent you from running the app on the device. If you don't have a provisioning profile or development certificate, it will create them for you on the Apple

Developer website and download them; the progress of this is shown as a spinner at the bottom of the Signing section:

Assuming everything works, you should now be set. The "Provisioning Profile" will say "Xcode managed profile". If you click the little circle-i info button next to it, you'll get a pop-up showing the auto-generated App ID, certificates, and even the default capabilities and entitlements. The next line down, "Signing Certificate", will also be populated as "iPhone Developer" followed by your name in parentheses.

The certificate is not exclusive to Xcode—it's actually added to the Keychain on your Mac. Open the Keychain Access app from the /Applications/Utilities folder, look under "My Certificates" and you'll find this "iPhone Developer" certificate. By doing File > Get Info, as seen in the figure, you can see details like how it was signed by Apple WWDR, forming the chain of trust described earlier:



Automatic Code Signing for Distribution

The process is different for code-signing an app for distribution. That's because the purpose is different: instead of ensuring that you've got the right credentials to install an app on your own device, distribution signing is about providing the app to other parties. That often means submitting to Apple for App

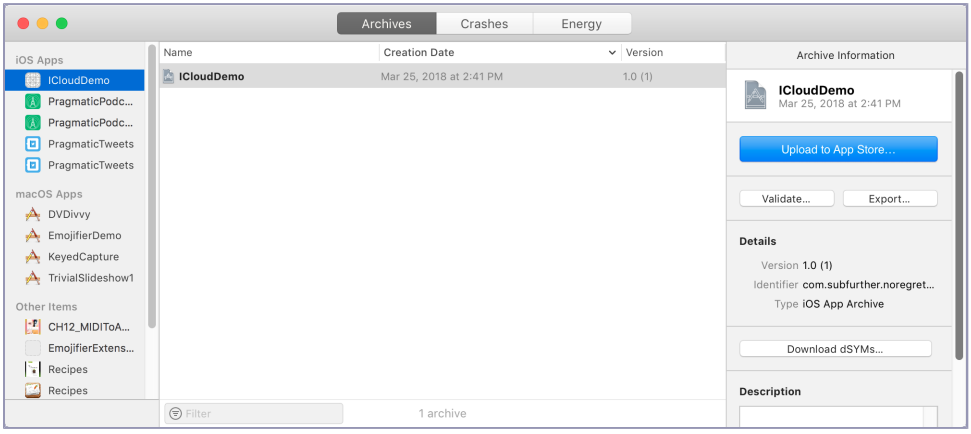
Store distribution, but it's also used for in-house ad hoc testing, enterprises distributing apps to their own employees, and (for macOS apps) "Gatekeeper" distribution.

Pay Up, App Distributors

Keep in mind that distribution certificates are only available through the paid Developer Program. If you're on the free program, you'll have a development certificate for installing to your own devices, but you won't be able to create a distribution certificate. Thus, you won't be able to do the kinds of signed distribution techniques in this section, like uploading to the App Store.

To distribute an app, you start by *archiving* your app, with the menu command Product > Archive. For iOS apps, this menu item will be disabled until you set the scheme selector to an iOS device or the "Generic iOS Device", rather than a simulator. Same goes for tvOS and watchOS.

The archive command performs a release-configuration build—see *Using Configurations*, on page 10, or just keep in mind that release builds will generally be configured for higher performance and smaller code size, at the expense of debuggability—and, if successful, opens up the Organizer window (⇧⌘6), as seen in the figure:

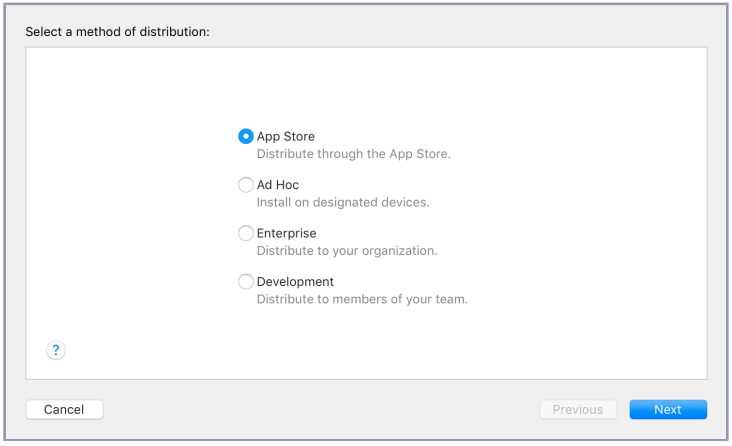


The Organizer keeps track of all the projects you've ever built with it on this Mac—the left column in the figure actually shows "Pragmatic Podcasts", "Pragmatic Tweets", and "Recipes" apps from earlier Pragmatic Programmers iOS SDK programming books. For each one of these, there's a list of archived builds, listed by version number and build number (which you saw how to manage in *Managing Your App's Version Numbers*, on page 12).

The created archive is a bundle that contains your app and a bunch of metadata. For Swift apps, it will also contain Swift dynamic libraries (.dylib) files for frameworks you use (which may change once the Swift abstract binary interface is stabilized, as planned for Swift 5). Thing is, you'll probably never need to deal with the bundle itself. Instead, everything you need is here in the Organizer.

With a build selected, there are three buttons on the right with actions you can take. "Upload to App Store" is prominently positioned, larger, and colorized… almost like Xcode is trying to call your attention to it, right? You can also use the "Export" button, which slides in a sheet (seen in the figure) with different kinds of distribution:

- "App Store submission" (to export an App Store build but not actually upload it)
- "Ad Hoc" (to distribute to a fixed number of devices within your company or testing group)
- "Enterprise" (to distribute throughout your company)
- "Development" (for manual installation)

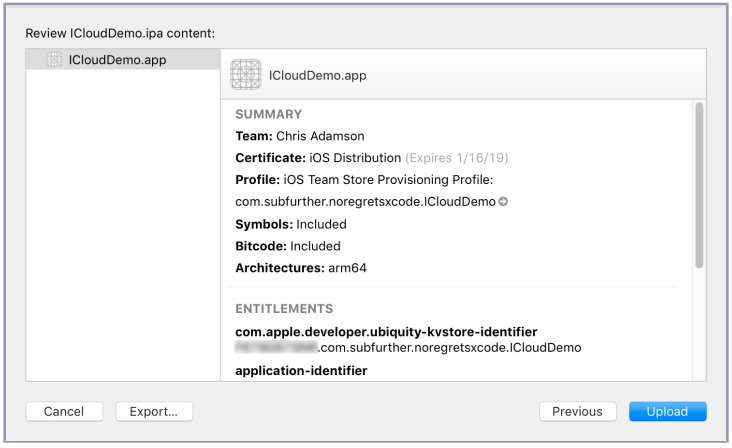


After you select one of the options and click Next, there are a few other options, like whether you want to use Bitcode, or use App Thinning to exclude resources for devices your app isn't meant to run on.

When you use "Upload to App Store", always choose the option to "Upload your app's symbols to receive symbolicated reports from Apple". With this option enabled, any crash reports you receive from Apple (shown in the Organizer's "Reports" tab) will have meaningful method and function names, rather than just offsets in memory.

For App Store distribution, your next step is dealing with app signing. Remember, this is different than development signing: rather than proving to your device you're who you say you are, you're proving it to the App Store. There are two choices here: automatic signing will create any needed App IDs, certificates, and profile for you and download them from the developer site. Manual signing is more work and will be discussed in the next section.

After signing the app, the sheet will show you one final summary (shown in the figure) of the certificate and entitlements it's using.



At this point, Xcode will re-sign your app with your development certificate, proving to Apple's reviewers that this app really comes from you.

At this point, Xcode has taken you as far as it can to submit your app to the App Store. For the upload to succeed, you'll also have to create an entry for the app on the iTunes Connect website, where you connect the App ID to a SKU that you'll use to track the app through its distribution to the public. That's not an Xcode topic really, so look at Apple's iTunes Connect guides to learn about all the metadata and legalese you need to set up for your app.

The process of code-signing your apps for development or App Store distribution is pretty straightforward with automatic signing turned on: enable the features you need in the capabilities tab, let Xcode create App IDs, certificates, and profiles, and you should be able to build and run or upload to the App Store without a hitch.

Manual Code Signing

On the other hand, if you want complete control of what's really going on, then you'll need to understand how manual signing works.

Think about what automatic code-signing provided: it created an App ID, created a development certificate, and then created a provisioning profile to indicate which entitlements you needed and which of your devices it could run on. Manual code signing is simply the process of performing each of these tasks yourself.

OK, "simply" is the wrong word here, because the whole code-signing process has long been the bane of many iOS developers. But it *can* be understood and made to work. Usually.

Development Teams

Arguably, there's not much benefit to a single developer using manual signing, because it mostly amounts to manually doing what Xcode is willing to do for you automatically. Manual signing makes somewhat more sense—although this is increasingly debatable—to *teams*.

Organizations in Apple's Developer Program can form teams on the developer website. Teams are made up of individual developers, and it's perfectly fine to use your existing Developer Program membership (even a free one) as a member of an employer's or client's team. Using the developer website, the organization's agent can create teams, add members, and assign them roles with various abilities. Enterprise memberships, which are meant for companies that develop and distribute apps internally instead of through the App Store, work the same way.

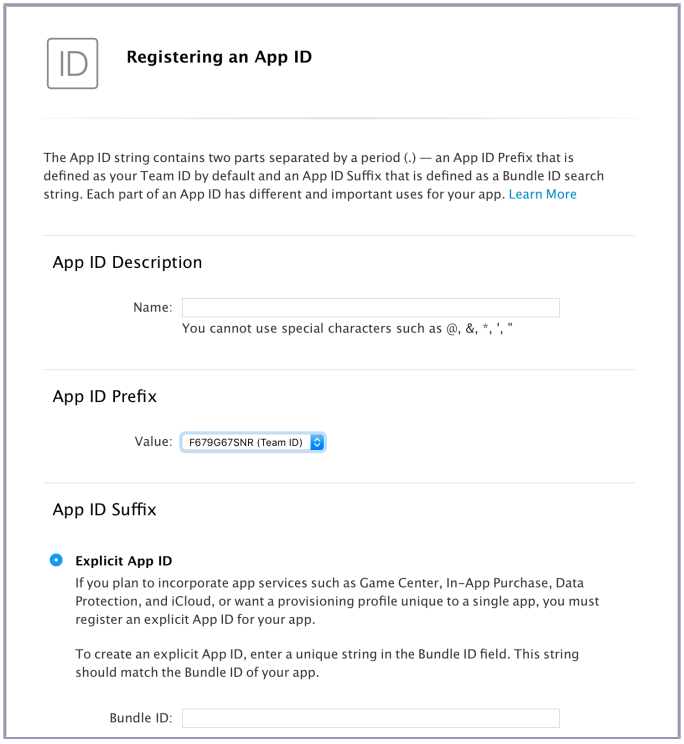
This is what the "Team" refers to in the signing section of your project's General settings. As an individual, you are your own team. But when you join an organization, you may need to be clear whether you're signing an app as yourself or as the team.

Most of the work of manual signing is done on the Apple developer website, in the "Certificates, Identifiers, and Profiles" section seen earlier. The certificate is easy: if you've ever done automatic signing, then you have a certificate on the website (and in your keychain). If you had to do everything from scratch for some reason, you can create a certificate on the website, which you'd then download to your Mac and double-click to add to your keychain.

Creating an App ID

The next piece of manual signing is the App ID, again only if it hasn't already been created for you by automatic signing. For that, you need your app's bundle identifier, which you can get in Xcode from the project settings—it's near the top of the General tab, right before the version and build numbers. You implicitly created this when you created your project, and it's typically a reverse domain name followed by a unique string for the app, like `com.pragprog.caxcode.iCloudDemo`.

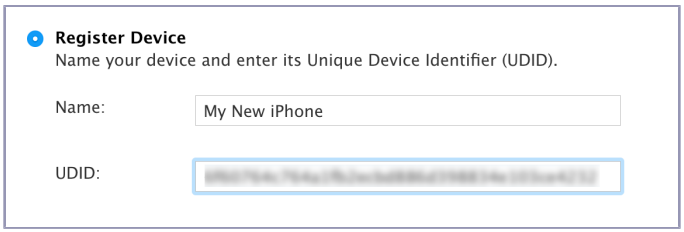
On the website, go to App IDs and click the plus (+) button to start creating an App ID. As seen in the figure, you'll need to give your App ID a name, an App ID Prefix (if there are more than one, choose the one that says "Team ID"), and your bundle identifier (under "Explicit App ID"). As you scroll down, you'll see all the features that can be enabled for an App ID; choose the ones you need, like iCloud, HealthKit, or whatever's appropriate for your app.



Registering Devices

The next piece of the puzzle is the specific devices you want to enable for development. Each account type can register a certain number of devices—for individuals, it's 100 each of iPhone, iPad, Apple TV, and Apple Watch. You need to specify these so that you can create profiles (in the next step) to run on those devices. You can reset your list once per year, resetting each time your developer membership renews.

On the website, the "Devices" section lists all devices registered to your account. To add a device, click the plus (+) button. Here you need to enter a name for the device (it doesn't have to match the device's actual name in the iOS Settings or Watch Settings apps), and a "UDID", which is a unique identifier string for the device as shown in the figure on page 185.



You can get the UDID as the "Identifier" in the Devices & Simulators window (⇧⌘2), which was introduced back in *Packaging App Data for Tests*, on page 167. Just select the long alphanumeric string and paste it into the web form. You can also get it from iTunes by bringing up the device information and then clicking on the Serial Number line, which will change to show the UDID. While this text isn't selectable in iTunes, doing a File > Copy (⌘C) will copy the UDID.

Creating Development Profiles

Now you have all the pieces in place for a profile: an App ID (which indicates the features it uses), and devices you want to run the app on. Under "Provisioning Profiles", you can click "Development" to start creating a new development profile, by clicking the plus (+) button.

The first step is to choose which kind of profile you want to install (iOS development, tvOS development, or some flavor of distribution). Choose iOS development.

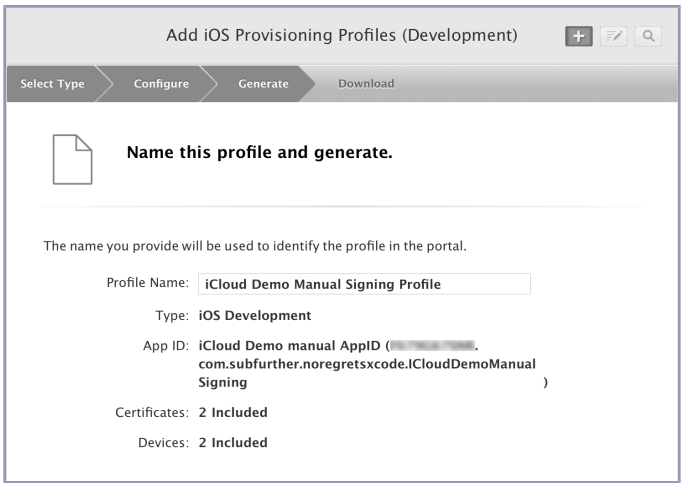
Next, there are three pages to set up the profile. First, you choose which App ID to use. Next, you choose which development certificates to use. You may have multiple certificates, if you've used automatic signing on multiple Macs. In this case, each certificate with your name represents a different computer, so select them all. On the third page, choose all the devices with which you want this profile to work.

When you're done with these pages, the site shows you a summary of your profile, as seen in the figure on page 186.

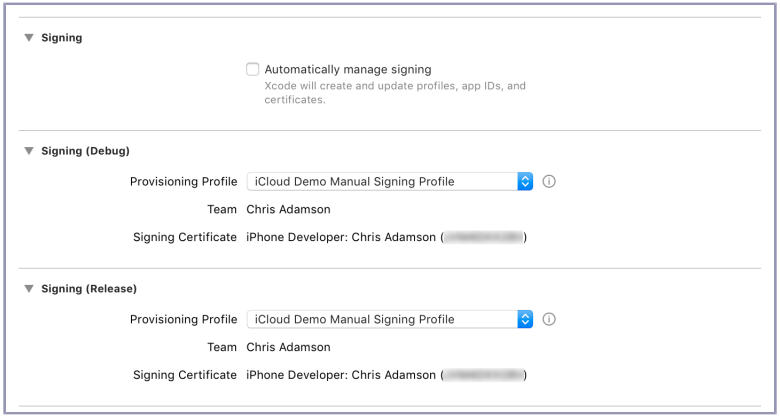
Click "Continue" to create the profile. On the next page, you'll get a confirmation that the profile was created, along with a download button. Download the file, find it in your `Downloads` folder, and install the profile into Xcode by either dragging the file to Xcode's dock icon or just double-clicking it.

Using the Development Profile

At this point, the development profile is ready to use. Go to your target's general properties and turn off automatic signing. This will create new signing



groups for “Debug” and “Release” (plus any other configurations you’ve created). Each will have a pop-up menu labeled “Provisioning Profile”; use it to select the manual profile, as shown in the figure:



There’s a little circle-i button next to the pop-up. You can click this to see a pop-over that shows details about the profile, such as the App ID, certificates, entitlements, and capabilities that the profile is configured with.

More importantly, your app is now ready to build and run. The manual profile provides everything an iOS device needs to install and run the app, secure in the knowledge that it’s from a known, Apple-approved developer.

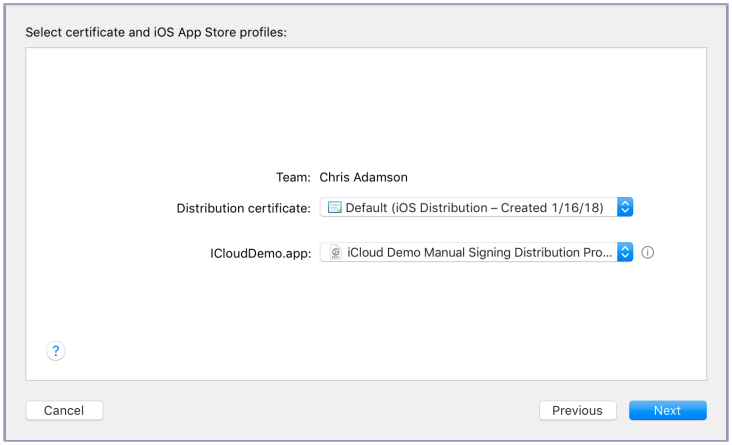
Manual Distribution Signing

When you’re choosing profiles for the Debug and Release configurations, you should use your manual development profile for both. You might think,

"doesn't release mean I should use a distribution profile?", but remember that signing for distribution only happens in the Organizer, like when you're about to upload to the App Store. In fact, your app is re-signed with your distribution credentials at that point.

But there is a distribution step to worry about: if you build your app with manual signing, you also have to use a manual process for distribution. And that means you'll need a distribution profile. You set this up on the site almost exactly like the development profile, except that you don't select specific devices, and you only have one certificate to choose: the distribution certificate for the account you've logged into (whether that's your personal account or an organization). Once you've created the profile, download it and install it into Xcode as before.

Then, when you reach the last stage of the Organizer's "Upload to App Store" or "Export" workflow, you'll need to choose your local copy of your distribution certificate, and the distribution profile you just downloaded, as seen in the figure:



If you think manual development and distribution signing seems like a lot of unnecessary work… it's actually worse than you think. The manual development profile must be present on every copy of Xcode that your development team uses. Furthermore, anything hard-coded into a manual profile will need to be replaced when any of its contents change. That means if you add a new capability listed in the App ID, add another iOS device you want to run on, or add a new team member (or just if someone gets a new Mac), you'll have to create a new profile to account for the change.

Updating profiles is one of the things that automatic signing takes care of for you, so really, you probably don't want to do manual signing unless you have some obscure build or workflow problem that automatic signing can't deal

with. And, yeah, it does happen. For example, if you manage apps for multiple clients and have their distribution certificates in your keychain, you'll probably need manual signing to make sure you sign the right app with the right certificate, rather than letting Xcode choose. You might also need manual signing if your project is doing something tricky, like if you build for different teams by configuration/scheme and each needs to be signed differently. Or you might be worried about your organization using up your test device slots when every developer's devices are added automatically. And honestly, some people just prefer the manual approach, because it lets them know exactly what's going on.

Wrap-Up

The security features provided by Xcode strike a lot of developers as annoying. Maybe a good way to think of them is to remember that you're not the one being protected. In fact, you are actually the one being protected against, as anyone with a copy of Xcode is potentially an attacker. What's being protected is Apple and the end-users of iOS apps. These security features ensure that you're who you say you are: a developer known to Apple and qualified to be in the Developer Program. Code signing also prevents your app from being altered by an attacker after you sign it. And thanks to this stringent authentication, your app can be safely permitted to use backend features like iCloud and push notifications. After all, think of the havoc that would be possible if features like that were open to just anyone in the world via a web API: bad guys could do a *lot* with that kind of power and access to the billions of iOS devices out there.

Keeping that in mind, you can see how the pieces fit together: local entitlements files tell a development device which features of the device you're planning on using. And on the Mac, Sandboxing uses these entitlements to limit your app to using just the features appropriate to it, rather than having carte blanche to use the entire filesystem, capture devices, and other sensitive features. And for the App Store, the combination of App ID, certificates, and profiles combine together to certify to Apple that your app should be able to make use of device features, and the end-user is protected because Apple knows it's really you submitting the app.

So that protects the user from you, but who's going to protect the source code? You could be one filesystem crash or drunken coding binge away from laying waste to all your work. In the next chapter, you'll see how Xcode's support for source code management will help you maintain good coding practices.

CHAPTER 10

Source Control Management

Having coded, designed, debugged, and tested your application, it would be a disaster if something were to happen to it. But it's not external threats you need to worry about; the greatest danger to the well-being of your app is *you*. Failing to back up your code means you risk being wiped out with a single hard drive failure.

But that's only one possible risk. What if you go on a late-night tear through your code—born of inspiration and optimism—only to wake up the next day with an app that won't build, no less run, and not enough undos in the world to get you back to a state where it worked? At this point, you're out of luck.

Any professional development requires the use of *source control management*, the formal system of tracking changes in your codebase. Properly used, good SCM practices will save you from disasters, help you understand the history of your work, and facilitate working with others.

Xcode has fairly deep support for *Git*, the mostly widely used source control system today. In this chapter, you'll see how it integrates with Xcode, what it does for you, and why you sometimes may need to go outside of Xcode for advanced features.

Git-ing up to Speed

While this chapter introduces SCM concepts gradually, it's not meant as a complete introduction to either SCM generally or Git specifically. For that, check out *Pragmatic Guide to Git [Swi10]*.

Creating and Cloning Git Repositories

In SCM, your code, storyboards, images, and all other assets are stored in a *repository*. The repository, or repo for short, tracks the history of every file—when they were created, renamed, deleted, modified (where you added or

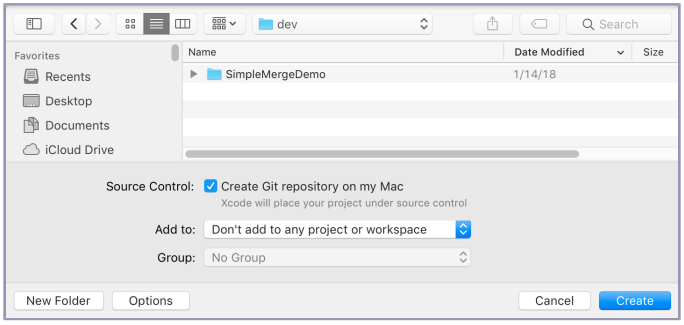
deleted lines), and so on. The distinctive feature of Git is that it is a *distributed* source control system, meaning that instead of having a single canonical repository on a server somewhere, everyone working with the codebase has their own, and shares update history amongst themselves.

Subversion Subverted

Xcode's SCM features are designed to be implementation-agnostic; in fact, earlier versions also supported the older *Subversion* source control system. However, Apple has removed most support for Subversion in Xcode. It's still possible to use Subversion-controlled projects with Xcode, but all the source control will need to be performed from the command line or with a separate Subversion application.

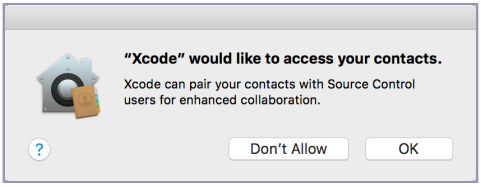
Creating a Repository

Creating a Git repository in Xcode is straightforward: it's one of the options when you create a new project. When the file dialog appears asking where to save the project (as shown in the figure), there's a check box for the option "Create Git repository on my Mac". This option will create a Git repository just for that project in that folder.



The repo won't be publicly visible to the internet or even to other Macs on your network, but it will allow you to track history as you start work on your project. Plus, you can always use this local repository as the basis of a shared repo later, so there's almost no reason not to go ahead and create the repo along with your new project.

If this is your first time creating a repo with Xcode, you may get a request to access your Contacts. This happens because every commit you make will have your name and email associated



with it. If you use multiple email addresses or prefer to work under a different name, and you later make this project public, this might disclose information you'd rather keep private. To prevent this, you can explicitly set your name and email for Git. *Prior* to creating your project, enter the following commands in Terminal:

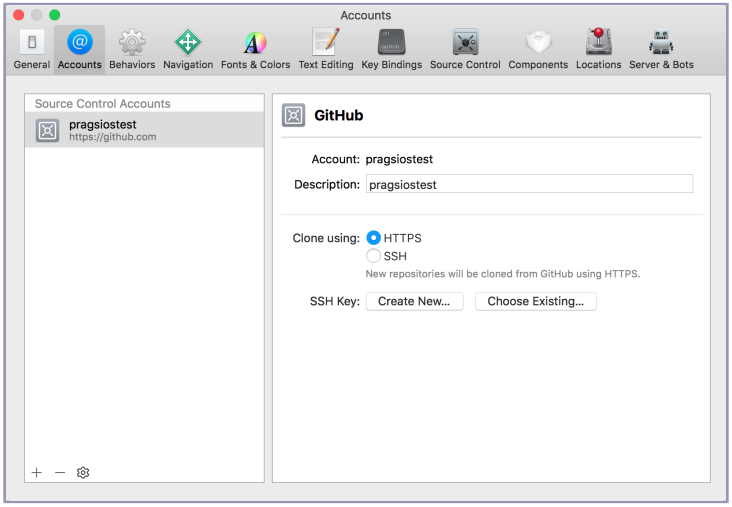
```
git config --global user.email "myaddress@example.com"
git config --global user.name "My Name"
```

These commands will apply to all future Git projects you create or import. You can also do this for a single repository. Using Terminal, `cd` to the project's top-level directory and issue the same commands as shown here, just without the `--global` flag.

Cloning Repositories

Actually, creating a repository isn't nearly as common as using a repo someone else has created. For example, you may have started a new job and need to check out the existing code, or you want to try out an open source project.

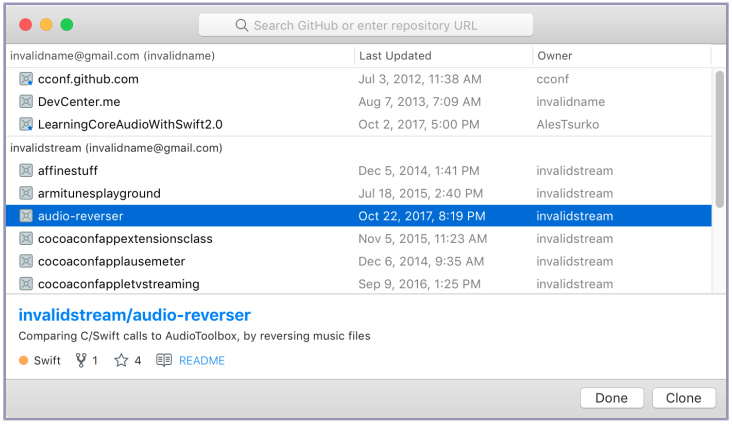
By far, the most popular host of Git repositories is GitHub.[1] Xcode 9 builds GitHub integration right into the app. To use it, start in Xcode's Preferences and go to the Accounts tab. When you click the plus (+) button at the bottom-left to add an account, it shows four options, one of which is GitHub. When you select it, Xcode asks for your GitHub username and password—create these on GitHub's website if you haven't already. Once you've entered them, Preferences shows your account details, as shown in the following figure:



1. https://github.com/

GitHub uses secure connections to exchange code, so you'll need to choose an existing SSH key, or generate a new one now. If you've ever used GitHub on this Mac—from the command line or the GitHub app, for example—you'll already have an SSH key in the hidden .ssh folder in your home directory, typically named id_rsa. Click the appropriate button in the right pane to either create a new key or choose an existing one.

Once your GitHub account is set up in Preferences, you can download and use any GitHub project of which you're a member. The basic idea in Git is that you *clone* a repository, capturing its state and a reference back to the original repo, and copy it to your system. Use the Source Control -> Clone... to bring up a dialog with all the GitHub projects of which you're a member. The following figure shows some of my projects; to clone one, select it and click "Clone":



On the other hand, if you're not officially a member of a project, it won't show in the list. But maybe there's an open source GitHub project you want to clone and try out. Even if you're not a member, you can check it out by just pasting its full URL (ending in .git) into the search bar at the top of the Clone window.

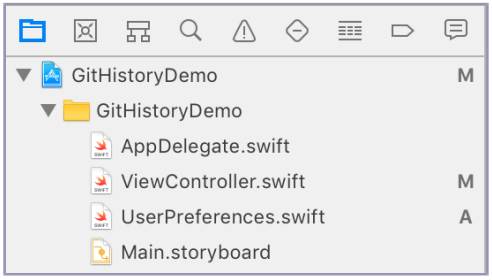
You can also clone a Git project from the Terminal. cd to the directory where you want to clone the project, and just type git clone and the URL of the project, as indicated on the GitHub web page. For example, to clone the audio-reverser project from the earlier figure, you would type:

```
git clone https://github.com/invalidstream/audio-reverser.git
```

This copies the repository to your system, and when you open the Xcode project inside this folder, it will work just as if you'd cloned it from the Source Control menu. This is also the technique you'll need to use if your project is hosted on a Git server other than GitHub, like an in-house company server.

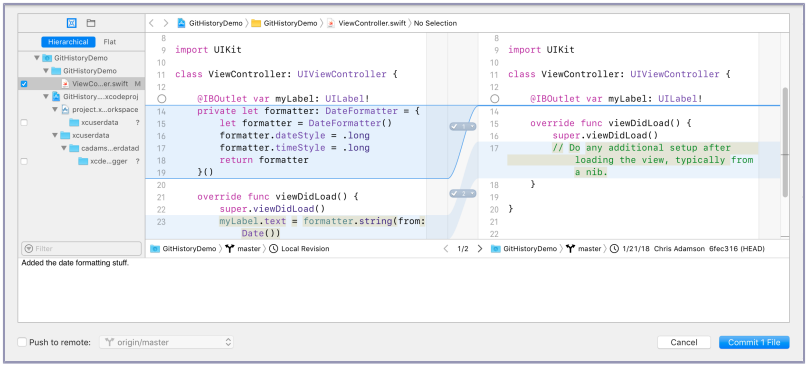
Using Your Git Repository

When a project is under source control, you'll notice some new behaviors as you work with it. In the File Navigator, single-character indicators are used on the right side of every line to show source control status. As seen in the figure, files you've added are marked `A`, and those that you've modified are marked `M`. At the bottom of the navigator, on the right side of the filter, a box icon that looks like a tiny bank safe will filter the navigator to only show files with source control status:



Committing Changes

Your changes pile up until you're ready to *commit* them to Git. This means that instead of a stream of constantly changing files like you might see in a backup utility, your Git history will consist of only those milestones you chose to mark as significant enough to commit. To commit your changes, use the menu item Source Control -> Commit. This slides out a sheet like the one in the following figure:



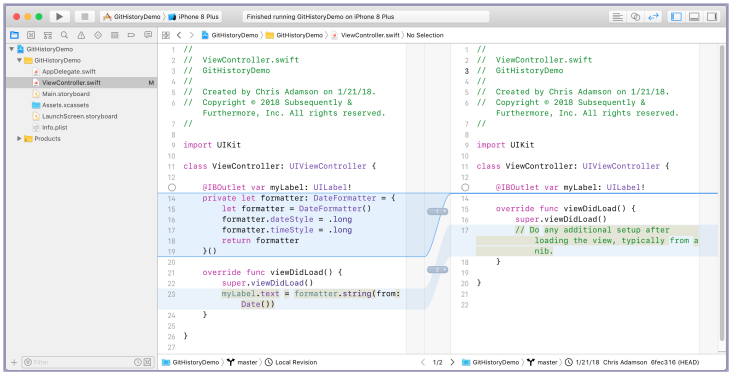
The UI here shows a hierarchical list of files on the left, somewhat like the File Navigator, with check boxes for which files will be committed; you can keep a file out of the commit by unchecking it. You can inspect the changes for each file by selecting the file. This brings up a side-by-side view of the

"local revision" of the file on the left—meaning the files in their current state—and the previous commit on the right. Your changes are highlighted with connections between the two sides showing where code has been added or removed. Each distinct change is numbered with a button in the gutter between the two source views. You can click the pop-up on this button to leave an individual change out of a commit, or even discard the change and roll back to the previous version.

The bottom of the view contains a text view for entering a text comment describing the changes you made for this commit. These messages can be really helpful in the future when you're looking at commit history to figure out what you were doing and why. So, try to develop a habit for writing good commit log messages; otherwise, future you is going to see a Git log that just says "Fixed bugs" 30 times in a row, and future you may not be happy about that.

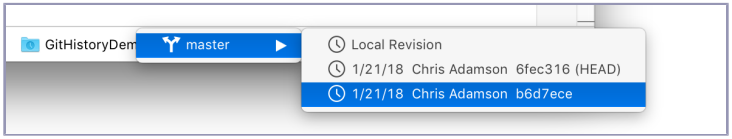
Viewing Git History

Once you've made more than one commit to your repository, the *Version Editor* will become useful to you. This editor is shown using the clock icon in the toolbar, in the group of three buttons that also includes the Standard and Assistant Editors. You can also show the Version Editor with the menu item View -> Version Editor (⌥⇧⌘↖). There are three modes to the Version Editor, which can be exposed with a long-press, or by clicking it again when it's already selected. The first mode is *Comparison View*, shown in the following figure, which presents a side-by-side view similar to what the commit sheet shows:

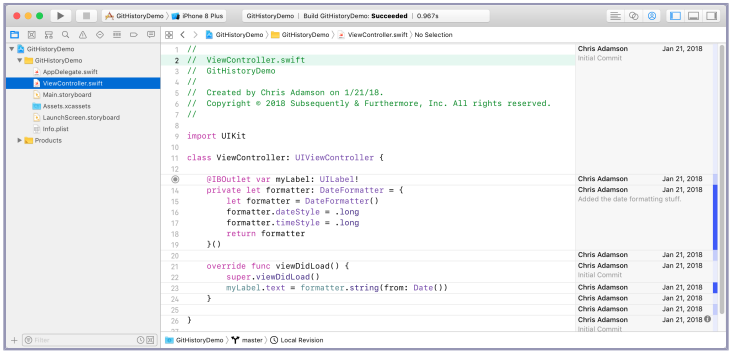


At the bottom of each pane, there's an indicator showing the current branch (to be covered shortly, and which defaults to master), and the specific revision, identified by a date, committer name, and a short hexadecimal id. The revision indicator is actually a revision *chooser*, a pop-up menu, as shown in the

following figure. While it defaults to the previous commit, you can choose older commits to see a comparison of the current state of your code to states further in the past. And, as with the commit sheet, you can use the gutter buttons on any of the differences to discard individual changes:

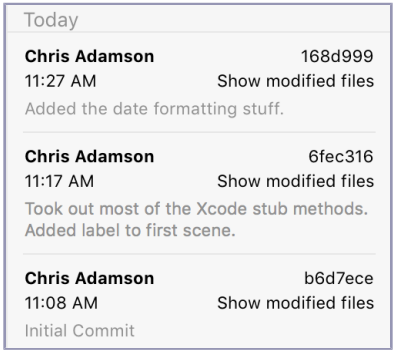


The second mode is *Authors View*, which indicates the date, committer name, and log comment for every line of code in the file. These changes are on the right side of the following figure:



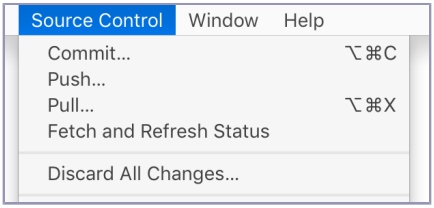
As its name indicates, this lets you know who's responsible for every line of the file. In fact, it's the equivalent of command-line Git's `blame` command, just with a less judgmental name. If you mouse over one of the entries in the right-side gutter, it'll show a round-i "info" button that presents a pop-up with more details (like the commit id), and a button to switch to comparison view with the selected commit in the right pane.

The third mode is *Log View*, which just shows a simple history of commits by date, committer name, and log message, as seen in the following figure:



Branching and Merging

If this was all there was to source control, it would amount to little more than a deliberate file backup system. Where it gets interesting, however, is when multiple developers are involved, each working on their own set of changes.



When you clone a repo from GitHub, you create your own history in a copy of the original. At some point, you may want to reconcile those histories. To start that process, you first do a *pull*. With a pull, you'll get changes from the remote origin repository and incorporate them into your repo. In Xcode, you initiate a pull request with the Source Control -> Pull menu item. Looking in the opposite direction, the remote knows nothing about your new commits. For that, you need to *push* your changes to the remote repo. In Xcode, you'll use the menu item Source Control -> Push to push your changes.

But what if the changes back at the origin are incompatible with yours? What if the changes you want to push are incompatible with the changes already there? This is where things get a little messy. The real problem is, there are potentially many people working on their own copies of the repository, and there has to be some way to manage those changes.

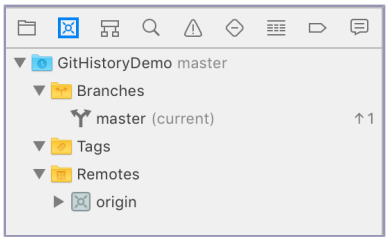
The main ways source control deals with this is branching and merging. *Branching* is the idea of having more than one series of commits. Instead, starting at some point in the commit history, you go off and make a different set of commits. This allows multiple developers to start at the same point and go in different directions with their work.

At some point, you bring your changes back to the original branch by *merging*, which means to reconcile the changes with the current state of the original branch. If you've changed files that nobody else has touched, this is trivial. Git can also figure out what to do if two branches modified the same file, but in different parts of the file. It's only when the same parts of the same files have concurrent changes that things get tricky.

Working with Branches

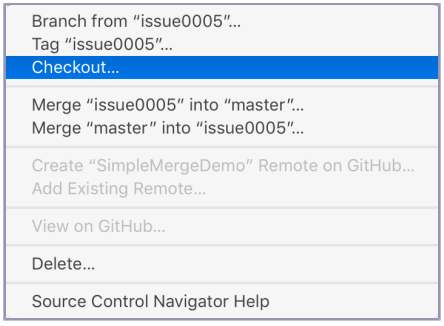
When you start work in a Git repository, all your work is on a branch named `master`. Many large projects will immediately create branches off of this, like a `release` branch that represents an always ready-to-release version of the code. Then they'll create a `development` branch off of `release` for new work that will be

merged into release when it's ready. And to this end, one common practice is to make each issue—a bug or a new feature—a branch off of development which lets developers work in parallel.



In Xcode, the Source Control Navigator (⌘2) is where you work with branches in your repository. It shows three folders: Branches, Tags, and Remotes. Tags are names assigned to mark specific commits, like when you release a 1.0 version, while remotes are remote repositories to which your repo is connected—usually the one from which you cloned it.

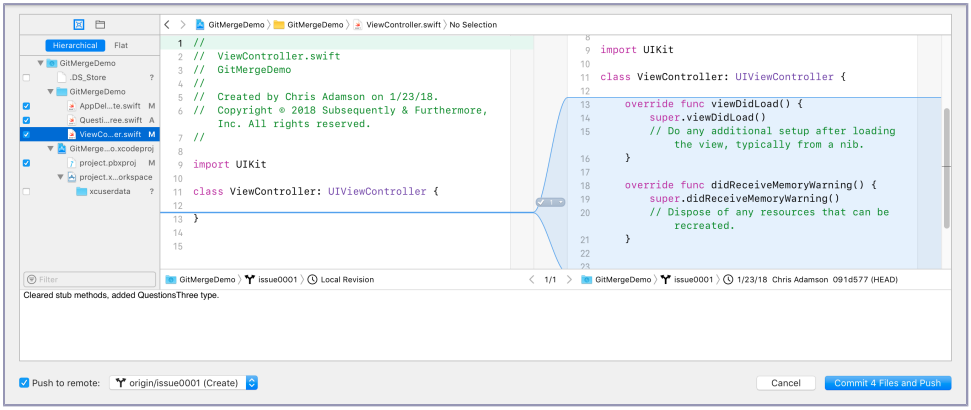
The interesting part is the branches. You start with master marked as your current branch. If you cloned a repo that already has other branches, you can switch to one by selecting it and bringing up the menu with the gear button at the bottom of the navigator (you can also show this menu by right-clicking or ctrl-clicking the branch name). To switch to that branch, select the "Checkout…" menu item, as shown here (this switches all files in the current working copy to their versions at the head—i.e., the most recent revision—of that branch):



This menu is also where you create new branches: just select an existing branch name—which will be master if you just created the repo—and choose "Branch from *branch name*…". A sheet slides in to let you give the branch a name, and includes a reminder that any code changes you haven't yet committed on your current branch will be available on the new branch.

Once you're on a branch, everything works as normal in Xcode. When you've reached a good milestone—maybe you've finished the task, or you just want to store your work for safe keeping—you can commit your code with Source Control -> Commit. A sheet shows a side-by-side diff of your changes, much like the Comparison View you saw earlier. There's also a text area at the

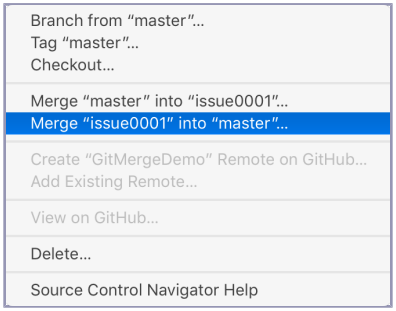
bottom where you can enter a log message describing what's in this commit. In the following figure, the first commit to branch issue0001 is being performed. The difference between the left pane (local file) and the right pane (previous state of the branch) shows that the big change in this commit is deleting the default methods Xcode puts in ViewController.swift:



This sheet also has a check box at the bottom-left that pushes the commit to the remote repository if there is one. In the screenshot shown here this check box is selected, but if you leave it unchecked, all your changes to the branch remain local to your machine, although you can manually push them with the menu item Source Control -> Push.

Dealing with Merges and Conflicts

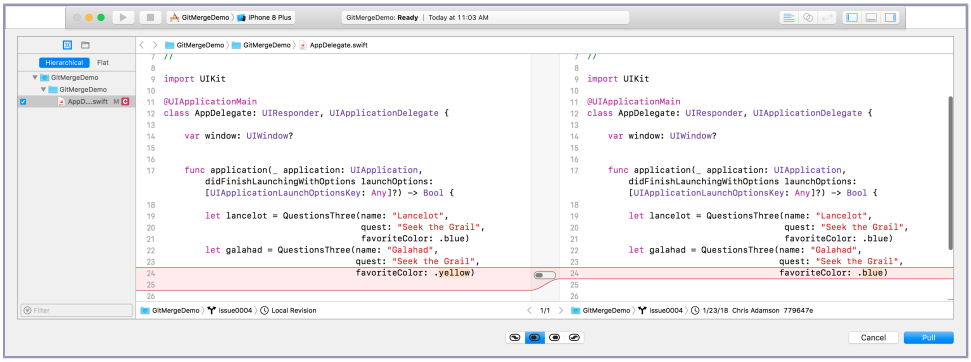
When you're done with your work on the branch, you usually want to merge it back into the branch you started with. In the Versions Navigator, select the source branch to which you want to apply your changes, bring up the menu with a right-click, control-click, or the gear button, and choose "Merge *my-branch* into *source-branch*". For example, in the preceding figure, issue0001 was branched off of master, so when work on that branch is ready to go, you'd merge issue0001 into master:



When you merge, Git checks to see if there are any *conflicts*—situations where it's impossible to automatically reconcile two versions of the same file. In a purely local scenario, this is usually trivial, since you're the only one making changes. When you're working on a team, however, it's another story.

Usually, this is an issue when you pull in changes from the remote repository with Source Control -> Pull. This updates any files that have changed since you cloned the repository or did your most recent pull. If there's a conflict on your current branch, or even the branch you started with, Xcode goes into a conflict-resolution mode.

The following figure shows a conflict in AppDelegate.swift—on line 24, the remote repository now has .blue on that line (right-pane), but I've changed my mind and my local version now reads .yellow (left-pane):

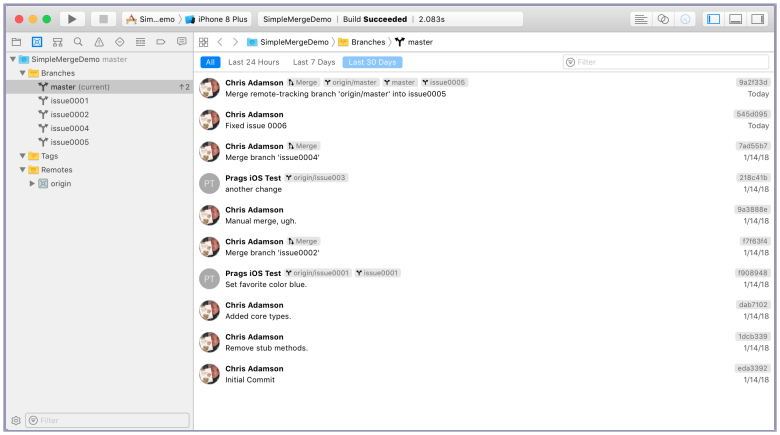


How is Git supposed to know which one is right? *It can't*, so you have to choose. The switch icon in the gutter for each conflict initially shows a question mark, and at the bottom of the sheet, there's a set of four buttons. These buttons indicate how you want to resolve the conflict: use both versions (putting the left side first), use only the left side's change, use only the right side's change, or use both versions with the right side first. In the figure, the left side has been chosen as the resolution of this conflict.

Once all conflicts have been resolved, you can complete the pull. Probably the first thing you'll want to do is build the app, just to sanity-check your merge decisions. If things are broken, you can use the Version Editor to compare your current state against older, working revisions, and try to figure out which lines of code are causing problems.

As you continue with this practice—put new work on branches, merge them back to the source branch when done—you'll eventually develop a history of commits. This is something you can view with the Versions Navigator—when

you select a branch, it shows the history for that branch in the Editor area, as seen in the figure:



Dealing with Xcode's Git Limitations

Having explored Xcode's Git support, it's important to acknowledge that it doesn't seem to be a popular choice among developers. I've repeatedly asked my 3,000 or so Twitter followers if anyone uses *only* Xcode for their Git needs, and to date, no one has said yes.

That presents two obvious questions: what does Xcode not do well enough (or at all), and what do people use instead?

Looking at the Git Index of Commands,[2] there are about 50 Git commands, some of which clearly don't exist in Xcode, and probably wouldn't make sense considering its UI. For example, you can't `bisect`, which is a targeted series of checkouts meant to find which version introduced a behavior change. You can't `stash` changes away while you switch branches. You can't `cherry-pick` or `rebase`, which are powerful but dangerous ways of rewriting your commit history. For hard-core Git users, Xcode leaves quite a bit to be desired.

Many developers use either the `git` command line, or Mac GUI applications for managing Git repositories, like GitHub Desktop[3] (which only works with GitHub-hosted repositories), Tower,[4] SourceTree,[5] and others. Some developers use these apps exclusively and ignore Xcode features, while others use Xcode's version comparison tools or even perform pushes, pulls, and basic branch

2. https://git-scm.com/book/commands
3. https://desktop.github.com/
4. https://www.git-tower.com/
5. https://www.sourcetreeapp.com/

management from within Xcode. It's your choice of how much you do or do not let Xcode involve itself in source control

In this section, you'll see a few real-world techniques for using Git with Xcode projects but without Xcode itself.

The .gitignore and .gitconfig Files

Xcode project files are actually bundles—directories that contain a file hierarchy of metadata for both the project itself and your use of it. This is how Xcode remembers things like what tabs you had opened the last time you worked in a project. Thing is, if we have many developers working on a project, none of them need to know about your tab settings or other preferences, so putting these files under source control is just noise to anyone looking at the Git history and a (small) waste of space.

To deal with this, you can create a file named .gitignore and place it at the top level of the repository (i.e., as a peer to the .xcodeproj bundle). This file contains filenames and patterns with wildcard specifiers (*), and any matching files are versioned by Git. A starter .gitignore for an Xcode project can be as simple as the following:

```
.DS_Store
xcuserdata/
```

.DS_Store is the macOS path for local metadata, like file tags. All your user-specific settings are tracked by files in xcuserdata, so excluding that directory keeps those files out of Git. These are the big two you need for Xcode 9. You can also get language-specific .gitignore files for Objective-C and Swift from GitHub's own gitignore project at https://github.com/github/gitignore. The names are actually a little misleading: rather than being language-specific, most of the files they screen out are bookkeeping files and build results used by older versions of Xcode, plus files created by third-party tools like CocoaPods, Carthage, and Fastlane.

Dot Files

Keep in mind that since .gitignore starts with a period character, it's invisible by default. Text editors like BBEdit and TextMate will warn you before saving a file that starts with a period, while the Finder won't even allow you to rename a file to one that starts with a period. Of course, there's always Terminal…

You can also create a file named .gitconfig to set Git properties to be used by all Git actions, whether via Xcode, the git command line, or something else.

A .gitconfig in your home directory is used as a default, but you can override it with .gitconfig files in project directories. Most of the settings apply to the command line git, but one common use is to set a universal default name and email:

```
[user]
        name = Prags iOS Test
        email = pragsiostest@gmail.com
```

There are lots of things you can set in .gitconfig; if you're going to use the command line, GitHub's own "gitconfig" project has a sample file to get started: https://github.com/gitconfig/gitconfig.

Merging from the Command Line

If you use one of the apps mentioned earlier, they'll provide their own UI for resolving merge conflicts, using a UI similar to Xcode's. However, if you use the command line, the default behavior is to just leave you with a modified file that includes markers to show both sides of the conflict:

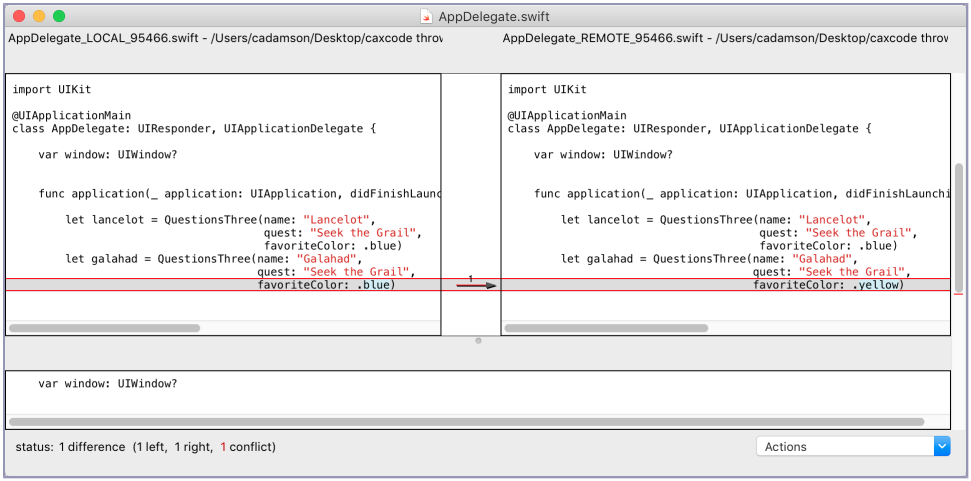
```
        let galahad = QuestionsThree(name: "Galahad",
                                     quest: "Seek the Grail",
<<<<<<< HEAD
                                     favoriteColor: .blue)
=======
                                     favoriteColor: .yellow)
>>>>>>> issue0005
```

The idea is that you're supposed to manually edit these files with a text editor to resolve the conflict. You know, *like a savage*.

On macOS, what you want to do is to edit your .gitconfig to set your *merge tool* to opendiff, like this:

```
[diff]
        tool = opendiff
[merge]
        tool = opendiff
```

With this setting, once you perform a merge that creates conflicted files, you can use the command git mergetool to open the built-in file merge app, /Applications/Utilities/FileMerge. As shown in the figure on page 203, it uses a side-by-side display to compare the versions and evaluate the conflicts, and just like Xcode, you choose right, left, or both to resolve them. When used from the command line, you close the window and quit each time you finish resolving one file, and if there are more files in conflict, FileMerge launches again. And it has to be said: this is surely the ugliest GUI of any app Apple ships today. Still, better than editing those >>>>>>>> lines, right?



Merging Project and Storyboard Files

Two Xcode file types are particularly susceptible to Git merge conflicts: project files and storyboards. They're both text files, so Git thinks it can merge incompatible changes, and sometimes it can. But when you have to manually resolve a conflict, it's both ugly and risky.

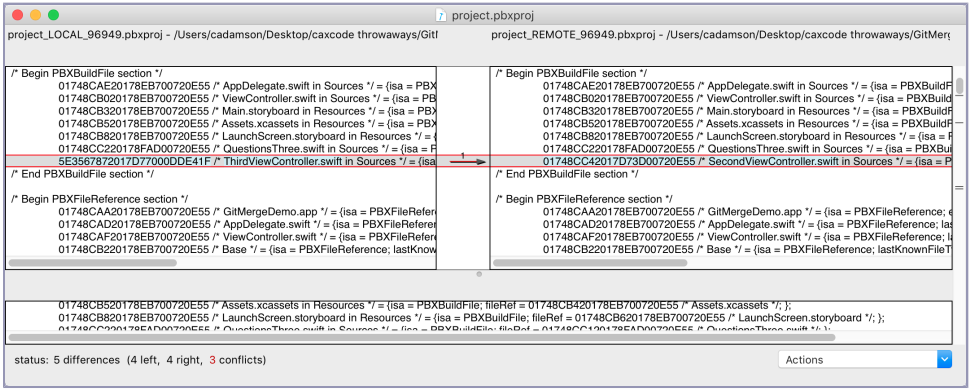
Merging Project Files

The easier of the two is project files. Project files aren't magic; look at your project file with the Version Editor instead of the Standard Editor and you'll see it kind of looks like a property list, with curly braces as containers, semicolons and commas as separators, and C-style /* ... */ comments. It's at least as human-readable as JSON, maybe more so.

When you get a merge conflict, it's usually a fairly simple matter of two branches adding new files to the same group, and they happen to get sorted into the same place in the file. But what makes it scary is that when you work from the git command line, the conflict markers basically corrupt the .xcodeproj file and leave it *unable to open*, because its contents now look like this (note that line breaks have been added to accommodate the book's formatting):

```
<<<<<<< HEAD
  5E3567872017D77000DDE41F /* ThirdViewController.swift in Sources */ =
    {isa = PBXBuildFile; fileRef = 5E3567862017D77000DDE41F
    /* ThirdViewController.swift */; };
=======
  01748CC42017D73D00720E55 /* SecondViewController.swift in Sources */ =
    {isa = PBXBuildFile; fileRef = 01748CC32017D73D00720E55
    /* SecondViewController.swift */; };
>>>>>>> issue0007
```

This is actually an easy problem to fix: both developers added new files, they happen to have landed in the same place in the file, so you just want to keep both of them. Do a git mergetool and accept both sides of this conflict.



The other option for dealing with a project file merge conflict is to simply accept all of one side's changes and ignore the others. What will likely happen is that the project won't build, because references to the new files won't be found—but that's an obvious and easy to fix error. In this conflict, if you just took issue0007's changes, then you'd lose ThirdViewController.swift from the project file, but then any code that calls ThirdViewController would fail to compile, and you'd know to just re-add that file in the File Navigator. Build again, commit, push, done.

Merging Storyboard Files

Storyboard merge problems are one of the most hated aspects of iOS and Mac development, and some developers justify not using storyboards entirely because of bad experiences with corrupted storyboard files.

First, you can largely eliminate this problem with a preventative measure: *use storyboard references*, as discussed in *Storyboard References*, on page 54. You can't have merge conflicts when different branches are modifying completely different files. This is a step many people overlook as their storyboards naturally grow along with the project, but you should really stop every once in a while and refactor large storyboards into smaller files by using references. There's no right number of scenes to have in a storyboard, just enough so that you're never tempted to have two developers in the same file at the same time. Or, if you do find you need to make concurrent edits like that, split the storyboard up into smaller files first.

Second, some storyboard conflicts are trivial. Storyboards are frequently marked as modified just because a user opened one. To see why, right-click

or control-click a storyboard file to open it as "Source Code" instead of the usual "Interface Builder - Storyboard". Notice that the storyboard is just a big XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<document type="com.apple.InterfaceBuilder3.CocoaTouch.Storyboard.XIB"
    version="3.0" toolsVersion="13122.16" systemVersion="17A277"
    targetRuntime="iOS.CocoaTouch" propertyAccessControl="none"
    useAutolayout="YES" useTraitCollections="YES" useSafeAreas="YES"
    colorMatched="YES" initialViewController="BYZ-38-t0r">
```

Since the systemVersion and toolsVersion (i.e,. the version of Xcode) are baked into the file, any differences between macOS and Xcode versions among the project's developers, including betas, appear as edits to the .storyboard file. The good news is, this is a simple string substitution that doesn't corrupt storyboard files. At worst, it just looks like a big revert war in the Git log until everyone standardizes their versions. Although, sure, it's still *really* annoying.

Other elements of the XML are prone to simple conflicts. <color> elements sometimes change based on the color space provided by a developer's computer. Older versions of Xcode used to have <rect> conflicts when the numeric values for coordinates or sizes would be integers or floating point based on whether the last person to touch the file had a Retina display. Seriously, you'd have revert wars where a value might change between 30 and 30.5 because one developer had a MacBook Pro and another had a MacBook Air. Fortunately, Xcode 9 seems to use integers everywhere, so this has actually improved.

Still, it's possible for a merge conflict to fundamentally corrupt the XML structure, and make the storyboard file not open. There are still several ways to recover from this:

- Do your best to make sense of the XML structure and edit it manually. Personally, I've successfully done this once or twice and it's not fun, but if you're looking at the two sides in FileMerge, you can sometimes understand what Git was thinking and set it right. Take ten minutes and give it a shot.
- Forget about making the merge work, and just re-perform the easier set of storyboard changes. To do this, revert to the branch that made the most changes, then check out a second copy of the project to a separate location, and open the other branch that led to the conflict. Open the storyboard in that project, select and copy any new scenes and segues, then paste them into the first project. This will look like new changes on the first branch, and you'll want to test them carefully.

It could be argued that major conflicts in project files and especially in storyboard files aren't source control problems at all but rather engineering and project management problems: your storyboards are too big, you're staying on branches too long, or your issues are too big and need to be broken down into multiple smaller issues. If you don't want to have an ugly storyboard merge, adopt development practices that make them unlikely to occur in the first place, right?

Wrap-Up

Source code management in Xcode is kind of a mixed bag. It's deeply built into the IDE, such that just looking at the File Navigator serves as a reminder of your changes since the last commit, with all those `M` for "modified" and `A` for "added" files. And you can totally drive a basic pull/branch/commit/merge/push cycle entirely within Xcode. Yet… a lot of developers only use a few of these features, or even none at all, and keep their SCM work separate, using either a dedicated Git app or the `git` command-line tool.

It's up to you to determine to what degree you want to depend on Xcode's Git support. Personally, I like the visualization tools and find Xcode is effective for going through old revisions and using the Authors View to find where things went wrong with a complex codebase. And yet, I still tend to use the command line, sometimes because I have to (for fancy repository tricks like `git rebase`), but maybe also because old SCM habits are hard to break.

CHAPTER 11

Platform Specifics

All of the features you've seen so far are largely applicable to any of the various Apple platforms; they can all be coded in Swift or Objective-C; their UIs can all be built with storyboards—in fact, watchOS *requires* storyboards; and they are all amenable to being debugged with LLDB and profiled with Instruments. However, Xcode has some features that are specific and meaningful only on certain platforms.

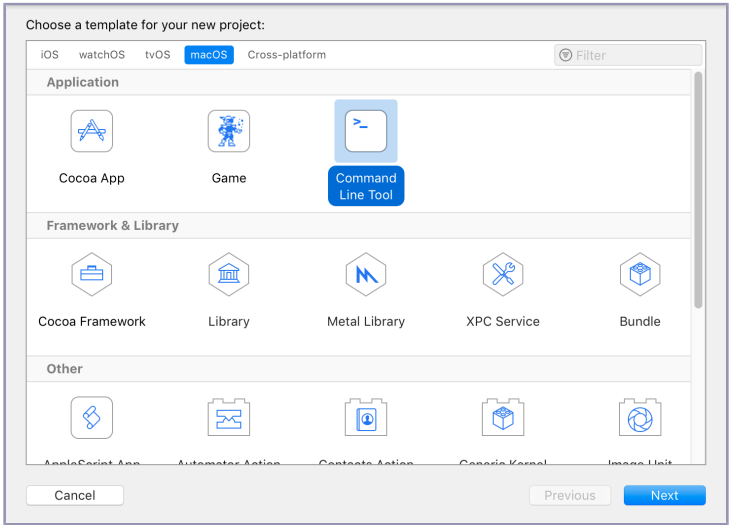
For this chapter alone, you'll look at Xcode not in a platform-agnostic way, but from the specific point of view of a macOS developer, a tvOS developer, and a watchOS developer. You'll see the specific features and needs that are unique to these platforms, and what Xcode offers when you're focused on a single platform outside of "iOS."

Command-Line Applications on macOS

macOS is the only Apple platform with a user-visible command line, and thus it's the only one where the idea of a *command-line application* makes sense. If you've worked with Terminal, you're likely familiar with Unix's standard built-in commands (`ls`, `chmod`, `more`, and so forth), and perhaps more substantial programs like the command-line version of `git`, or `ffmpeg` for converting audio and video files between formats.

On the surface, you might think these apps are different from double-clickable Mac apps, because they exist as a single executable file that must be invoked from a command line or shell script. But as you saw back in *Understanding App Bundles*, on page 89, the macOS app bundle is just a folder structure that contains an executable file, along with other resources like graphics, localizations, and metadata.

If you want to write a command-line application of your own, Xcode makes it easy. Start a new project with File -> New -> Project (⇧⌘N) and when the template sheet slides in, choose the “macOS” tab, and the “Command Line Tool” template, as shown in the figure:



The next step of the project setup asks for the usual details like a product name, team, and organization info, but one interesting difference is the number of languages available. With this template, you have *four* languages from which to choose: Swift, Objective-C, C++, and C; whereas full-blown Cocoa apps only include the first two. There’s also a significant difference in how a command-line app works in Swift versus the C-based languages (which you’ll see momentarily), so choose carefully. That said, Xcode will always let you mingle languages, so your main file can be Swift and make use of helper code in Objective-C source files, or vice versa.

Command-Line Apps with C-Based Languages

If you start with a C-based language, Xcode sets you up with a `main()` function that takes two parameters. These two parameters are used to receive arguments from the caller (which can be the command line, a shell script, or some other kind of invocation). The first parameter is an `int` indicating how many arguments you received, and the second is an array of `char*`, that is to say C strings, with the arguments themselves.

If your app accepts arguments, you need to first look at the list of arguments being sent to see if they make sense. Many command-line apps start by scanning the arguments, and if they don’t make sense, they print a usage message to the standard output and terminate.

For example, here's an Objective-C command-line application that plays the system beep (if there's an argument, it's parsed into an int, and that number of beeps is played):

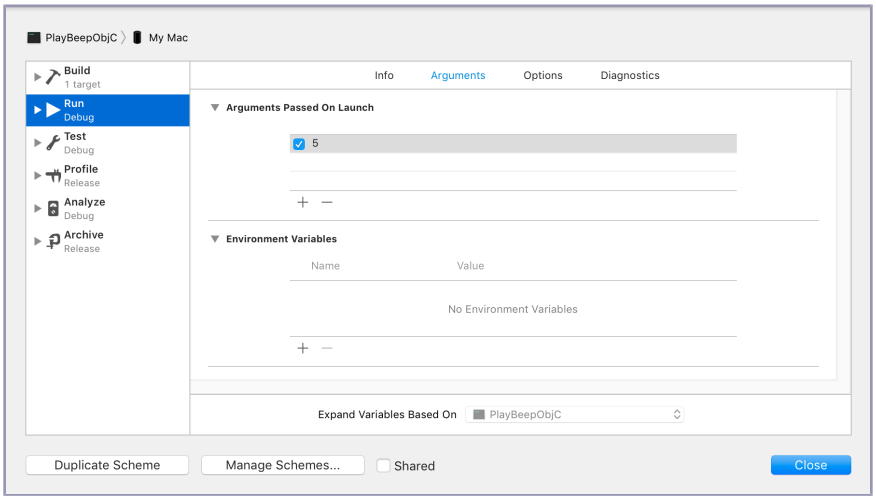
platforms/PlayBeepObjC/PlayBeepObjC/main.m

```
Line 1  #import <AppKit/AppKit.h>
-
-   int main(int argc, const char * argv[]) {
-     @autoreleasepool {
5
-       int beepCount = 1;
-       switch (argc) {
-         case 1:
-           break;
10       case 2: {
-           int count = atoi(argv[1]);
-           beepCount = count ? count : 1;
-           break;
-         }
15       default:
-           NSLog (@"Usage: PlayBeepObjC [beep-count]");
-       }
-
-       for (int i=0; i<beepCount; i++) {
20       NSBeep();
-
-         // keep app alive long enough to hear the beep
-         NSDate *wake = [[NSDate alloc] initWithTimeIntervalSinceNow:0.5];
-         [NSThread sleepUntilDate: wake];
25     }
-
-     }
-     return 0;
-   }
```

The processing of command-line arguments happens on lines 6–17. What may look unusual at first is that the code is looking for *two* arguments, with the second argument indicating the beep count (i.e., argv[1]). The reason for this is that the path to the executable is always passed as the first argument, and any further arguments follow that.

So how do you run your command-line app with an argument while you're developing and testing it? You *could* build the project, then go to the Project Navigator, expand the "Products" group, right-click the executable, do "Show In Finder" and then double-click the file, or navigate to that directory in the Terminal. But that's *awful*. Not only is it a lot of work, it also cuts Xcode out of the loop, meaning you can't easily debug the application.

A better way is to go to the scheme selector (on the left side of the toolbar) and select "Edit scheme…" From here, select the "Run" action, and notice that its details include an "Arguments" tab. As seen in the following figure, the top section lets you create arguments to pass to the application, such as `5` to request five system beeps. A second section lets you set environment variables that the app can read at runtime; these might ordinarily be set by the user's command-line shell, or perhaps a launcher script that calls your app:



Once you've added arguments to the scheme selector, running the app from the Xcode "Run" button will pass those arguments to the app when it launches. Better yet, since running the app from Xcode launches it via LLDB, you can set breakpoints and employ all the debugging techniques from back in Chapter 6, *Debugging Code*, on page 109.

The other thing you may notice is that command-line apps exit once they leave the `main()` function, so unless you're doing something special to keep your app running, the "Stop" will turn back into a "Run" button once the app terminates.

Command-Line Apps with Swift

Xcode's support for a Swift-based command-line app is similar to what it's like for C-based languages, but Swift itself is very different in this context. For starters, you don't need a `main()` function; you can just write code freely (you saw this with the Swift version of "Hello, World" back in *Compiling Code*, on page 89).

That said, you may be wondering how to get the command-line arguments since you're not receiving them as function parameters to a `main()` function

Using Frameworks in Command-Line Apps

One thing that may surprise you about this example is that it imports AppKit, for the sake of using Cocoa's NSSound class. Don't assume that just because an app is command-line based that you can't use the rich collection of Apple-provided frameworks—you totally can. For example, the built-in /usr/bin/afconvert command uses the AudioToolbox framework to let you convert between audio formats and encodings, in the form of a command-line app that you can then call from scripts or other automated processes.

You can even inspect what libraries and frameworks other executables are using, thanks to otool, a command that inspects executable files. In this case, executing otool -L PlayBeepObjC reveals that the app uses the Foundation, CoreFoundation, and AppKit frameworks.

To be technical, the modern otool is just a shim that calls through to the open source objdump, but otool's syntax is easier to remember. For example, the equivalent to otool -L filename is objdump -macho -dylibs-used filename.

like in C. In Swift, this is handled by an enumeration in the Swift Standard Library called CommandLine. This has a property named arguments, which provides the command-line arguments as an array of type String. So, you can write an equivalent command-line beeper with much less code than you needed in C:

platforms/PlayBeepSwift/PlayBeepSwift/main.swift

```
import AppKit

var beepCount = 1
let args = CommandLine.arguments
switch args.count {
case 1:
  break
case 2:
  if let count = Int(args[1]) {
    beepCount = count
  }
default:
  fatalError("Usage: beepcount [number-of-beeps]")
}

for _ in 0..<beepCount {
  NSSound.beep()

  // keep app alive long enough to hear the beep
  let wake = Date().addingTimeInterval(0.5)
  Thread.sleep(until: wake)
}
```

Edit the arguments in the scheme selector, and this works just like the Objective-C version from before.

Daemons, Agents, and Installers

Aside from programs meant to be invoked directly by the user on a command line, one common use of this style of programming is for *daemons* and *agents*. Daemons are apps that launch when the system starts up, while agents run in user space and launch when the user logs in. These are different from *launch items* that a user may have configured in their System Preferences, because daemons and agents generally don't have UIs, and instead just run in the background.

You create daemons and agents by creating configuration files and putting them in special directories like `/System/LaunchDaemons` or the user's `Library/LaunchAgents`. This file contains the path to the executable, arguments, and other metadata. Xcode doesn't help with this at all; look up "Daemons and Services Programming Guide" on https://developer.apple.com to get started.

Since daemons and agents aren't the kinds of things sold on the Mac App Store, you need to create an *installer package* to get both the configuration file and the executable file onto the user's machine. This is a bundle containing your files and an XML *distribution definition file* that tells the built-in Installer app where to put the contents.

Unfortunately, Xcode doesn't create installer packages, and Apple has eliminated "PackageMaker", its Mac app for making `.pkg` files. Instead, you need to use the command-line utilities `pkgbuild` and `productbuild`. For more information, see their `man` pages to get started, as well as the "Distribution Definition XML Schema Reference" on Apple's developer site to write the distribution definition file that you import into `pkgbuild`.

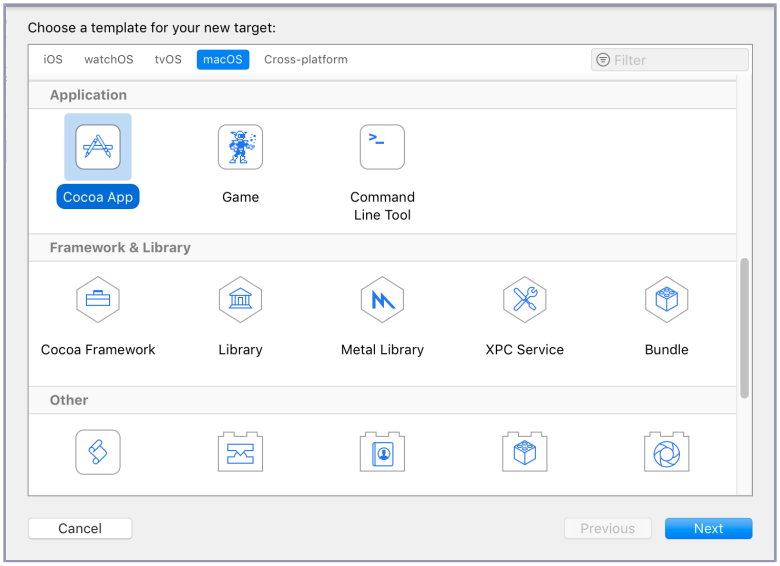
Multi-Platform Projects

On the other hand, what if you have some great iOS code, and you'd like to bring it to the Mac? Well, you can just wait until 2019, when Apple brings UIKit to the Mac, right? *Wrong!* You can bring most of your code to the Mac without waiting for Apple to do the job for you by leveraging the power of Xcode targets.

To be clear, UIKit on iOS (and tvOS) and AppKit on macOS are two fairly different things. After all, they should be—iPhones don't have windows or menu bars, and Macs don't rotate or have touchscreens. But beneath those top-level UI frameworks, most of the underlying frameworks are the same: both have Foundation, Core Location, StoreKit, and other application-logic frameworks. In fact, both platforms' UI frameworks are built on Core Graphics, Core Text, and Core Animation. Some classes and methods differ or are absent altogether, but overall the two platforms are a lot more alike than they are different.

You might intuit that to make an app share code between platforms, you'll want to make heavy use of frameworks; after all, that's a key strategy for sharing code between apps and app extensions. You saw this technique back in *Creating App Extensions and Frameworks*, on page 18, which had a sample project to share code representing upcoming developer conferences between an app and its extension.

Adding macOS to this project is surprisingly simple. You start by going to the top-level project and adding a target. When the sheet slides out, choose the macOS tab, and then the "Cocoa App" icon, as seen in the following figure (you will need to provide a name that's unique from other targets in the project, like `UpcomingConferencesMac`):

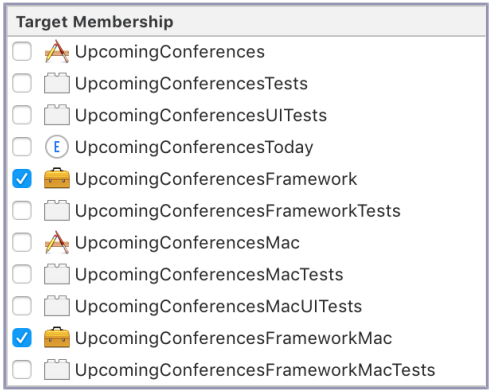


You now have a project that can build for both platforms… but only because the macOS project doesn't have any meaningful code. The point of this exercise was to share code, after all, and none of the existing code is being built for macOS.

There's a quick and dirty way to fix this, and a good way to fix this. You can add any file to any target either by checking its "Target Membership" in the File Inspector, or by manually adding it to the Compile Sources build phase. So the quick way would be to manually add any needed sources to the macOS app target. The problem is, these source files are still in a group for an iOS framework, which makes no sense for any macOS developers on your project.

A better way is to have any source files used by both platforms in their own group. You start by adding a macOS framework target, just so the Mac architecture resembles its iOS equivalent. Then you can create a group named

"UpcomingConferencesFrameworkShared" and move any files you need for both platforms' frameworks in there. Finally, just use the File Inspector to add each shared file to both framework targets:



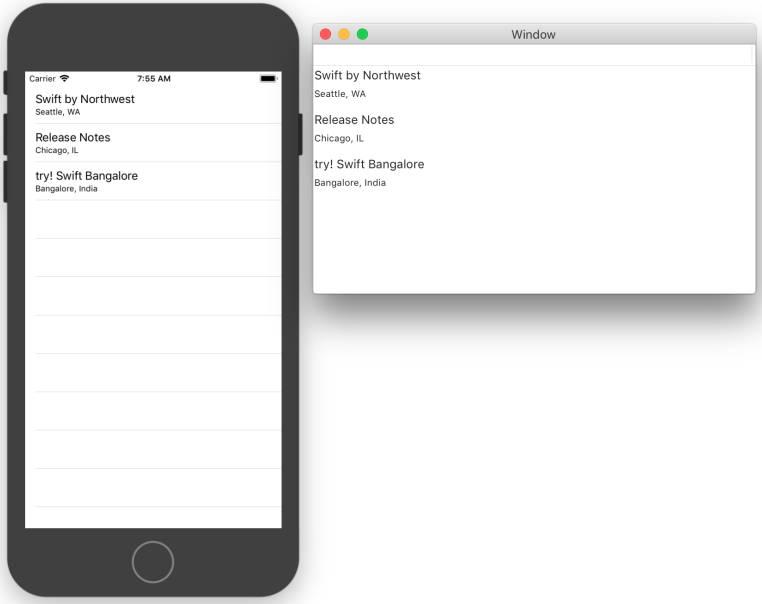
And that's basically all there is to it. Your macOS ViewController can bring in the same business logic code with import UpcomingConferencesFrameworkMac, and then you can limit your Mac-only code to focus on just the specific UI differences between the platforms.

You will want to think about how to put as much reusable code as possible into platform-agnostic source files to get the biggest bang for your buck. For example, instead of implementing UITableViewDataSource and NSTableViewDataSource with largely redundant code (like returning section counts and cells at index paths), you could have both implementations delegate to a common "view model" class, as advocated by the *Model-View-View Model* (MVVM) architecture. Here is how the downloadable sample code provides a platform-agnostic table model:

platforms/UpcomingConferencesPlusMac/UpcomingCo … meworkShared/UpcomingConferencesTableMod¬el.swift

```
public class UpcomingConferencesTableModel {

  public init() {}

  public let numberOfSections = 1

  public func numberOfRowsInSection(_ section: Int) -> Int {
    return UpcomingConference.allConferences().count
  }

  public func conferenceForCellAt(indexPath: IndexPath)
    -> UpcomingConference {
      return UpcomingConference.allConferences()[indexPath.row]
  }
}
```

By delegating to this model, you can trivially create native tables on iPhone and Mac from the same data and business logic, as seen in the following figure:



Setting up your own targets and architecting cross-platform code is a little more work, but it sure beats waiting for Apple to solve your problems for you, and you can make a more polished and native-feeling app for your Mac users by adopting the real native UI classes.

Multi-Platform iOS and tvOS Apps

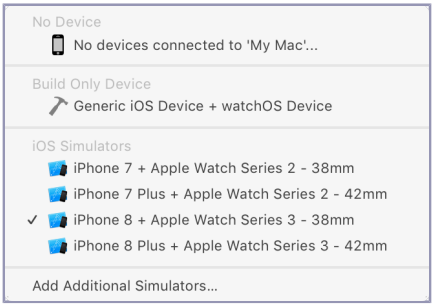
One particularly good use for multi-platform projects is to add a tvOS app target to your iOS project. Because tvOS uses UIKit, it's an even easier port than going to Mac. Plus, you can market your iOS and tvOS apps together, so that a purchased iOS app will be available for download on the user's Apple TV, without them having to go to the App Store on their TV.

However, for the App Store to know that the tvOS and iOS apps are the same product, they need to share a bundle identifier. So, while your targets need different names in Xcode, the bundle identifiers need to match. You can fix this on the TV app target's general setting page, changing the bundle identifier to remove the "TV" (or whatever you added to make the TV target unique), thereby matching the iOS app target's bundle identifier.

Working with watchOS Simulators

The idea of a multi-platform project won't come as a surprise to Apple Watch developers, because a watchOS project must always have multiple platform targets: one for the iOS app, and at least one for watchOS. In fact, there are usually several watchOS targets: one for the app, one for a notification handler, and possibly another for a complication.

Because watchOS apps are bundled with iOS apps, you don't use the same-name scheme as described for tvOS apps that have iOS counterparts. Instead, you use a naming convention for the bundle identifiers: for app `com.app.foo.FooApp`, the watch app would be `com.app.foo.FooApp.watchkitapp` and its extensions (notification handler and complication) would be `com.app.foo.FooApp.watchkitextension`.

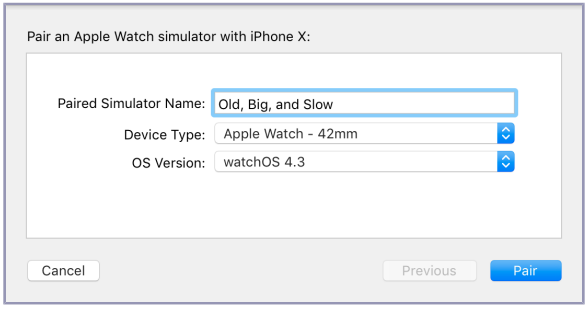


One thing that is unique and unusual about watchOS development is the use of multiple devices or simulators. This is a must because you install your iOS app to the iPhone, and then the watchOS app is installed to the watch. A given Apple Watch can only be paired with one iPhone. But going in the other direction, a given iPhone can be paired with multiple Apple Watches (good news for those of you fashionable enough to need different watches for different looks). However, only one watch can be *active* at a time, meaning it's the one that receives notifications or exchanges files via `WKWatchConnectivity`. With real devices, you make one of your paired watches active just by putting it on. In the simulator, it's another story, as seen in the figure.

When you go to run your watchOS app or an extension, the scheme selector shows a pairing of an iPhone model and a watch model. By default, these are arranged as a large and small phone of the same model, coupled with the large and small watch released around the same time. For example, one destination will be "iPhone 8 + Apple Watch Series 3 38mm", and another will be "iPhone 8 Plus + Apple Watch Series 3 42mm". These simuated devices are paired: only this simulated phone works with this simulated watch, and vice versa.

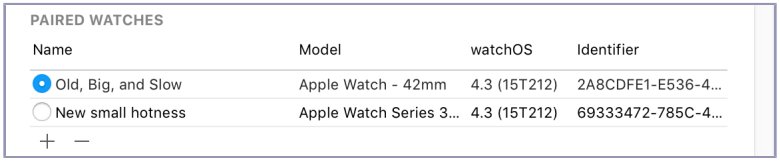
These parings are sensible enough as a default, but they're kind of limiting. What if you want to test on a new phone but an old watch (or vice versa)? Furthermore, Xcode 9 doesn't include any option for testing a simulated iPhone X with an Apple Watch. What are you supposed to do to test that combination?

As it turns out, the combinations of simulated phone and watch models shown in the scheme selector are entirely customizable. The settings are in Windows > Devices and Simulators (⇧⌘2), which was introduced back in *Packaging App Data for Tests, on page 167*. The Simulators tab is where you can customize your watch schemes, albeit indirectly. The left side shows every iPhone and iPad model with an installed simulator (keep in mind that you can download additional simulators and older iOS versions from Xcode's Settings window). If you select one, you'll see it has a list of paired watches, just like physical iPhones do. However, this list also has a plus (+) button, which lets you add simulators. So, if you wanted to combine an iPhone X with a first-generation Apple Watch, you'd just select iPhone X from the list of simulators, and then click the plus (+) button. This slides out a sheet (shown in the figure) that lets you select a watch model and optionally give it an easy-to-remember name:



Once a given phone model has any paired watch simulators, it shows up as a destination in the scheme selector, as seen in the following figure:

It's also possible to create multiple watch simulators for use with a given phone model. Once you use the plus (+) button to create a second watch simulator, the watch models in the list get radio buttons to determine which one is currently active, i.e., which one will be shown in the scheme selector.



On-Demand Resources for tvOS

So now that you've seen iOS, macOS, and watchOS, what do you make of tvOS? tvOS apps are kind of *weird* because Apple TV as a computing platform is *weird*. It's a box that's always plugged in—and thus has no energy consumption or battery use concerns—yet has the CPU and storage characteristics of a mid-tier iPhone. Moreover, unlike a Mac or an iPhone, which still has some functionality without a network connection, the Apple TV is assumed to always be connected to a network, and is nearly useless otherwise.

Being always connected means a tvOS app is in some ways akin to a browser app. In fact, tvOS apps were originally limited to 200 MB in size (though this restriction was eliminated in 2017), with the assumption that apps could download additional content once they're installed and running. This idea is still essential to the Apple TV experience: rather than simply running out of storage like a Mac or iOS device would, the Apple TV will purge unneeded resources to free up space… although this means apps will have to re-download those resources later if they're needed again.

The way this works is with a system of *on-demand resources*, which are well supported by Xcode.

Not Just for tvOS

While on-demand resources are primarily aimed at tvOS apps, they're also available on iOS. If you write an iOS app that uses a lot of large resources, but uses each one only for a limited time (like a level-based game, or episodic content), on-demand resources may make sense for you too. They'll allow your user to save time and money by not downloading unused assets, and make your app less of a multi-gigabyte target for deletion when your user runs out of storage.

Developing an On-Demand Strategy

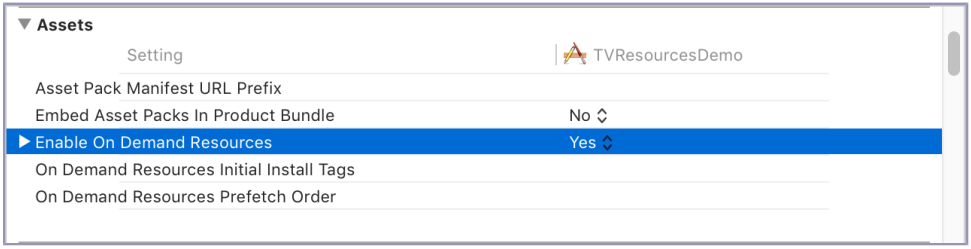
To use on-demand resources, you need to do some planning up front. Think about the following:

- *What kinds of resources can be loaded on-demand?* Pretty much anything other than executable code can be an on-demand resource. In general, you probably want to think about large files like images, textures, shaders, audio, video, and perhaps scripting-language code.

- *When will you need them?* An image used everywhere in your app might as well be in your app bundle (though it can still be made an on-demand resource to make the initial App Store download smaller). Instead, you need to strategize about things that may be needed in certain situations, long after the app has been run for the first time. For example, if you have a game that starts off as a romantic comedy but turns into a sci-fi war epic, you can put off downloading the giant robot graphics until after the player reaches the big twist.
- *Which resources go together?* Depending on your app, resources might be groupable by a specific part of the app (like different levels in a game), or by how it's used (maybe you have certain sounds that are only used in 1-player or 2-player modes, for example). Keep in mind that any asset can belong to multiple groups, and can be loaded when any group requires it.
- *Can the disruption of the download be minimized?* Downloading resources will take time, and rather than slamming the door on the user while resources download, you may want to think about how to get resources loaded just in time. For example, you can load a game's main resources in the background while the player is working through a tutorial mode, or load the level 3 data while they're fighting the level 2 boss.

Setting Up On-Demand Resources

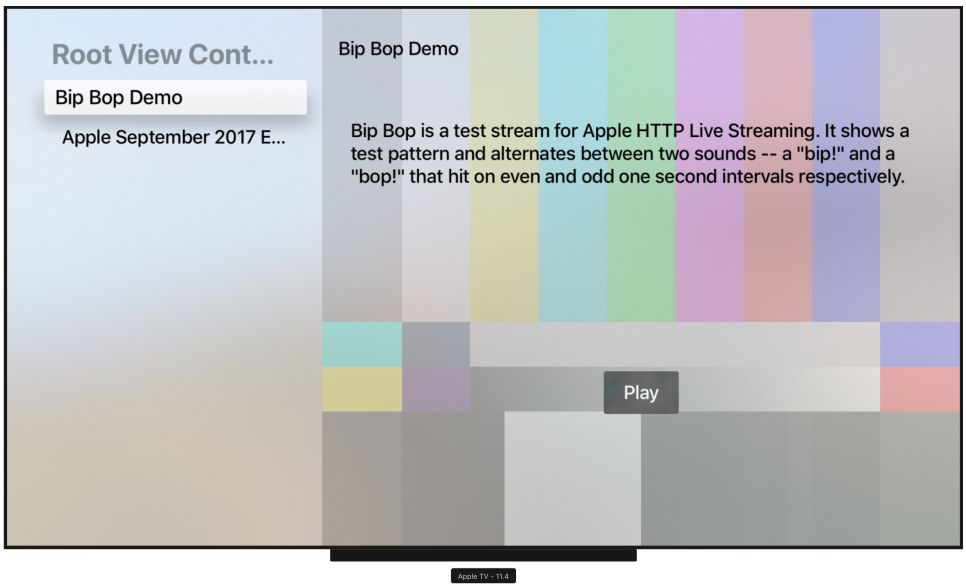
The first thing you need to do to use on-demand resources is to opt into using them. You do this by going to your target's build settings and looking in the "Assets" group for "Enable On Demand Resources", as seen in the figure. It's set to "Yes" by default for tvOS and iOS projects, but it's always worth a quick look to be sure:



Next, prepare your resources. You might think that to make resources available on demand, you'd have to put them on a website somewhere. But actually, you put your resources in the Xcode project itself, and then mark them as being on-demand. For images, this has the additional advantage of working

with the asset catalog system for supporting multiple Retina resolutions: provide 1x, 2x, and 3x versions of your images, and the right one will be sent to the user at runtime.

The download code has a sample project, TVResourcesDemo that reworks the example from *Embedded Scenes*, on page 50, which showed a list of videos by title and description, and let the user drill down into them. On Apple TV, the descriptions aren't enough to fill the screen and make it interesting, so this sample adds a background image for a little visual flair. These background images are the on-demand resources, and are collected in an asset catalog called Images.xcassets. The following figure shows the app running in the tvOS simulator:



The two videos available in the sample code are the same as before: the "bip bop" test for HTTP Live Streaming, and an Apple WWDC keynote. Imagine that these are the first of two groups of videos the app will make available: diagnostic videos (to test your connection), and event videos like keynotes. Those will be the two groups of resources: any time the user watches a diagnostic video, the app will load a test pattern background behind the description, and when they watch an event video, they'll see a WWDC audience image.

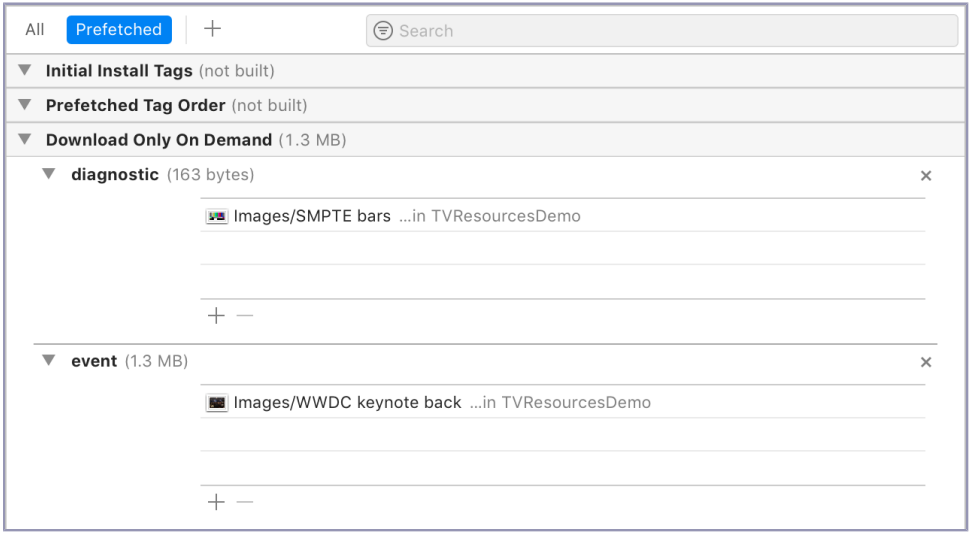
The way to "tag" an image (or any other file in your project) is to select the top-level project file in the Project Navigator, select the target, and select the *Resource Tags* tab. This is where you create resource tags and select which files (or members of asset catalogs) will get those tags.

The resource tag editor either shows all resources, or categorizes them by behavior. Switch to “Prefetched” to show the categories, which represent different behaviors for resources:

- *Initial Install Tags*: These resources are downloaded with the app itself, and count toward its size on the App Store. However, they can be purged locally if the system needs to free up filesystem space.
- *Prefetched Tag Order*: These resources start downloading after the app is installed. The order in which tags are arranged in this category determines the order in which they are downloaded.
- *Download Only on Demand*: These resources are only downloaded when explicitly requested by the app. This is the default category.

To create a new resource tag, click the plus (+) button at the top of this view, next to the “Prefetched” button. By default, this will create a new table titled “New” in the “Download Only on Demand” group. “New” is the name of the resource tag—which you should immediately change to something meaningful—and the table shows all the resources tagged with that name.

To add a file to a tag, click the plus (+) button at the bottom of that table and select a file from the sheet that appears. If you add an asset catalog, all images in the catalog are added to the tag, but you can selectively remove images you don’t want the tag to apply to. In the following figure, two tags have been created—diagnostic and event—each containing one image from Images.xcassets:



Once you add a resource (such as an image or a file) to a resource tag, its attributes inspector will show all of the tags that apply to it. For example, selecting the WWDC picture in Images.xcassets shows that it has the event tag:

Also, if you have moved any tags into the Initial Install Tags or Prefetch Order categories, these will be listed in the Assets group of your target's build settings.

Downloading On-Demand Resources

Once you've set up your on-demand resources, the last step is to actually use them. With an on-demand resource, you can't just immediately call it up with UIImage(named:) or by search for it in the app bundle, because *you can't assume it's even there.*

Instead, you need to work with the NSBundleResourceRequest class. You create an instance of this class with a set of tag names, and the request object coordinates your use of resources that use those tags. You can ask the request if the tagged resources are already downloaded, start downloading resources that aren't yet on the device, and mark the tagged resources as no longer being needed.

In its simple form, this is a two-step process: call conditionallyBeginAccessingResources() to see if the tagged resources have already been downloaded. This method takes a completion handler that receives a Bool parameter indicating whether the resources are already downloaded. If true, you can load the resources from the app bundle as you normally would.

If the completion handler gets a false parameter, you will need to call beginAccessingResources() to start downloading. This call takes another completion handler, which will be called when the download completes. Its parameter is an optional Error, where nil means there was no error and the resources downloaded successfully.

The sample code combines both of these steps into a convenience class called ResourceManager, with a single method called loadTag(). This method takes a single tag name and a completionHandler parameter, which will be called immediately if the resources are already downloaded, or once the download completes or fails with an error:

platforms/TVResourcesDemo/TVResourcesDemo/Resource Management/ResourceManager.swift

```
func loadTag(named name: String,
             completionHandler: @escaping (Error?) ->Void) {
  let tags: Set<String> = [name]
  let request = NSBundleResourceRequest(tags: tags)
  request.conditionallyBeginAccessingResources { success in
    if success {
      print ("already had \(name)")
      completionHandler(nil)
    } else {
      print ("beginning access for \(name)")
      request.beginAccessingResources() { errorOrNil in
        completionHandler(errorOrNil)
      }
    }
  }
}
```

Then, when the view controller that shows a clip's description needs the background image, it calls this method with the name of the image's resource tag, along with a completion handler to load the image and set it on a background UIImageView:

platforms/TVResourcesDemo/TVResourcesDemo/PlayerSceneVCs/PlayableItemViewController.swift

```
ResourceManager.shared.loadTag(named: backgroundImageTag) {
  [weak self] errorOrNil in
  if errorOrNil == nil, let strongSelf = self {
    DispatchQueue.main.async {
      strongSelf.backgroundImageView.image =
        UIImage(named: backgroundImageName)
    }
  }
}
```

This example does nothing to mitigate the user experience during the download; the user will get a black background until the resource is downloaded, and then the background image will just pop in. Depending on your needs, you may need to explicitly show the user some sort of "downloading…" UI while they wait. Fortunately, the NSBundleResourceRequest object has a progress property that you can use to update a progress bar or some other indication of how far the download has progressed.

Tags Can Overlap

Keep in mind that a given resource can appear in any number of tags; if an image was tagged as both diagnostic and event, then requesting either of those tags will download the image.

Layer Stack Images for tvOS

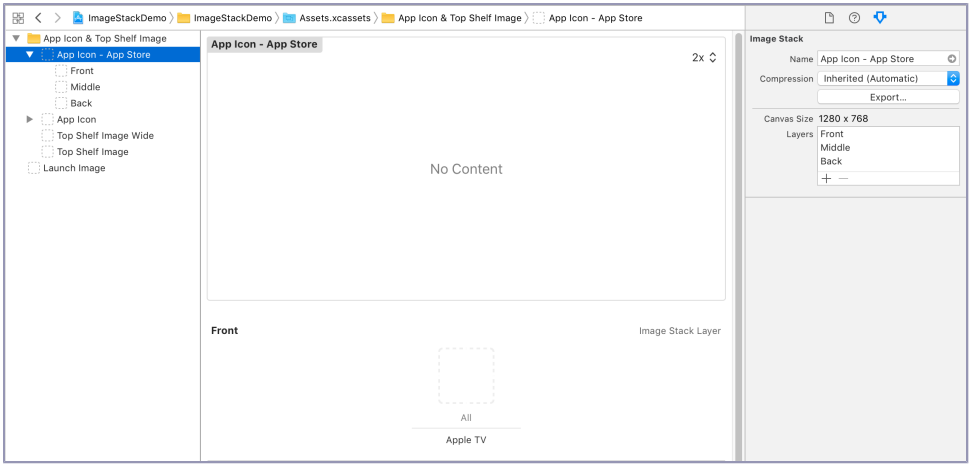
tvOS also has a unique look to its home screen, and this takes additional work on your part. When you start a new project, your Assets.xcassets file contains empty entries for not only an "App Icon", but also a separate App Store version of the icon, and two "Top Shelf" images.

The app icons are where you need to learn new tricks, since they're not just normal PNG files. tvOS app icons are *layer stacks*, a set of equally sized images that are stacked atop one another and then tweaked with a parallax effect to give an illusion of depth. There are three ways to create and edit layer stacks:

- *Parallax Previewer*—A standalone Mac app for importing .png and .psd files and exporting layer stack images as .lsr files. You can download it from Apple's developer site, https://developer.apple.com.
- *Parallax Exporter*—An Adobe Photoshop plug-in that lets you export the layers of a Photoshop image as an .lsr file. This is also available on Apple's developer site.
- *Xcode*

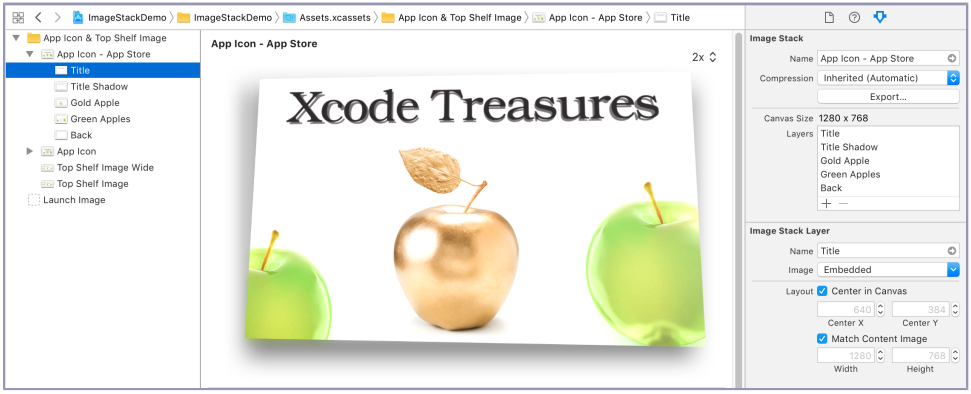
Since this is a book about Xcode, we'll look at option 3… but keep in mind author Bill Dudney's famous maxim: "*never* ship programmer art." Unless you're also a designer, hire a real one, and set them up with the Photoshop plug-in.

Layer stack icons for tvOS must use between 2 and 5 layers. If you expand the "App Icon" in the asset library, you'll see that it defaults to three layers, titled "Front", "Middle", and "Back", as shown in the figure. You can rename these in the Attributes Inspector, along with adding, removing, and reordering the layers:



App Store icons need to be 1280x768, and the icons used on the actual device need to be 400x240. For the smaller icons, you should also provide graphics at 2x resolution (i.e., 800x480), for use on 4K TVs.

To actually create the image stack in Xcode, export each layer of your stack as a separate PNG file at the required size. In Xcode's asset library, select the app icon (on the left) to expose image wells for all the layers. Drag the PNG file to the image well to set the image as the content for that layer. As you populate the layers, a preview at the top of the Editor area shows the resulting layered image. You can mouse over the preview to see how the parallax effect will be applied to the layers. In the following screenshot, the only overlap is between the title and a shadow layer below it, and mousing over the corners of the image creates the greatest separation between the layers:



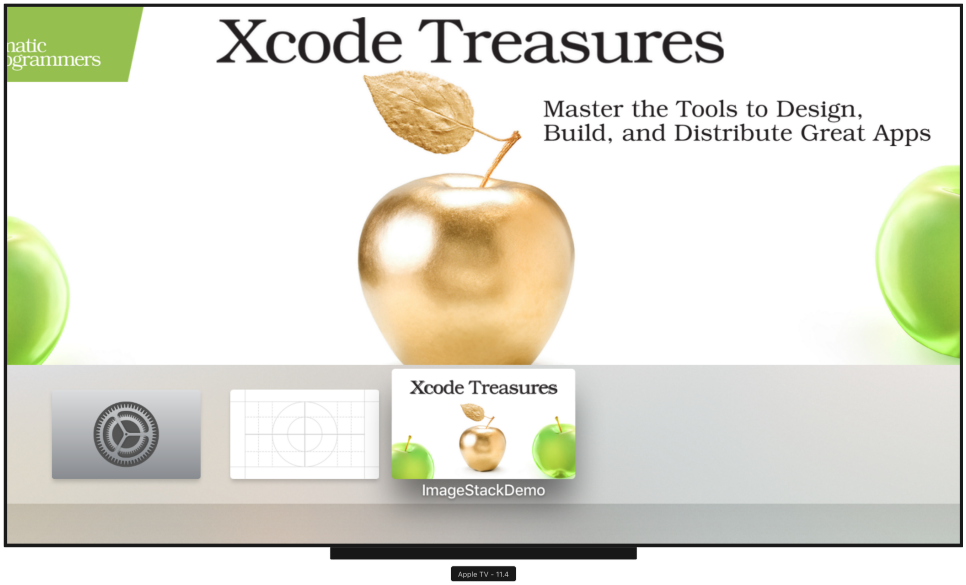
The Top Shelf images are much simpler, since they're just ordinary PNG files, although you'll still need 2x versions for users with 4K TVs. They'll only be used if your app is in the top row of apps *and* you're not using the TVTopShelfProvider to populate the top shelf with user-specific content. The figure on page 226 shows a Top Shelf image for the book that appears when its app icon is selected.

Mostly, the top shelf will be a task for your designer to make the best use of the space, so see the tvOS section of Apple's Human Interface Guidelines for how to use it.[1]

External Monitors on iOS

While this chapter has focused on the "other three" platforms so far—macOS, tvOS, and watchOS—there are a few features in Xcode that are iOS-specific, and yet aren't that well known. For example, iOS is the only platform for which

1. https://developer.apple.com/design/human-interface-guidelines/tvos/overview/themes/



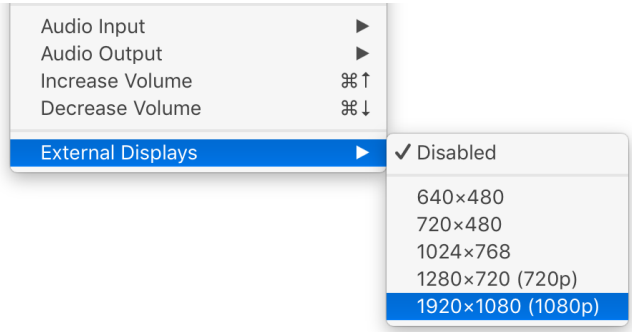
Xcode supports the idea of an external monitor: you can't have an Apple TV or an Apple Watch with two screens, and while Macs can have multiple monitors, developers are largely insulated from this by the windowing metaphor.

External monitor support for iOS supports some really interesting application types. For presentation apps like Keynote and PowerPoint, iOS lets you have a presenter view on the device's screen, and your displayed slides on the external monitor (which is often a projector or a big-screen TV). Another popular use is for party games where an emcee controls the game from their iPhone or iPad, and presents a game screen for players on the external monitor (you may have seen this in game show–style events at pop culture conventions).

With actual hardware, you connect an external monitor either with a Lightning adapter (which then connects via an HDMI or VGA cable), or via AirPlay from Control Center. At development time, Xcode provides external monitor support as an option in the Simulator app.

To simulate an external display being connected to your simulator, go to the "Hardware" menu and select "External Displays", then drill down in the sub-menu to choose the size of the simulated external display as shown in the figure on page 227.

In your code, connecting an external display will fire off a `Notification` named `UIScreenDidConnect` (in Objective-C, this is an `NSNotification` named `UIScreenDidConnect-Notification`). This is your signal to grab the notification's `object` as a `UIScreen` and



build a view hierarchy within it. This event can happen at any time once the app is running, so the book's sample code project adds an observer for this notification at startup, in the AppDelegate's application(_:didFinishLaunchingWithOptions:) method:

platforms/TwoScreenDemo/TwoScreenDemo/AppDelegate.swift

```
NotificationCenter.default.addObserver(forName: .UIScreenDidConnect,
                                       object: nil, queue: nil)
{ [weak self] notification in
  guard let newScreen = notification.object as? UIScreen,
    let strongSelf = self else { return }
  strongSelf.setUpPlayersScreen(screen: newScreen)
}
```

The simulated external monitor appears as an ordinary Mac window, with the name of its host device in the title bar, like "iPhone 8 - External Display". In the following figure, the sample code sets up a "Jeopardy!"-style game show event on the projector, which you could host at a developer conference as soon as you come up with some good questions:

Wrap-Up

Apple has four great platforms for app development, and Xcode supports all of them. As a result, there are features of Xcode that are only applicable on some or one of those platforms. In this chapter, you got your Mac on by writing a command-line application, which would blow the mind of classic Mac users of the 80s and 90s who didn't even *have* a command line.

Then, you saw how to create a project with app targets for multiple platforms. While this is useful for tvOS developers, it's essential for watchOS, where it's often set up for you. Also on watchOS, you can use the "Devices and Simulators" window to create arbitrary pairings of simulated iPhone and Apple Watch models.

Next, you got to see several tvOS-focused features, starting with setting up on-demand resources to make your tvOS (or iOS) a lighter initial download, by only downloading content as it's needed—and with Apple footing the hosting and bandwidth bill, to boot. Next, you saw how to create tvOS-specific image assets like Top Shelf images and image stacks for app icons.

Finally, in a return to focusing on iOS, you saw how the Simulator app supports external monitors.

In the final chapter of this book, you'll learn about Xcode's support for extending Xcode itself with your own functionality.

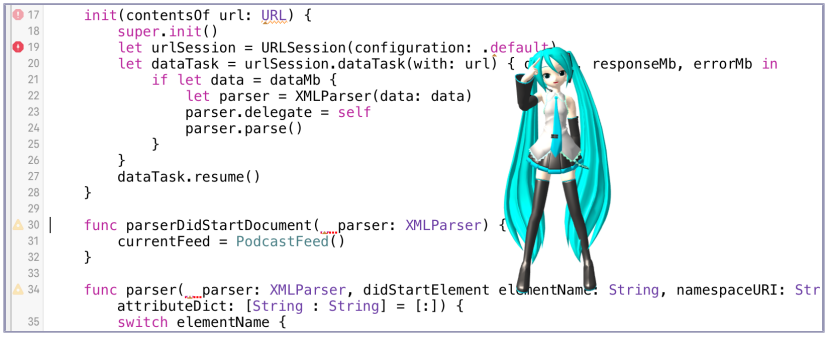
CHAPTER 12

Extending Xcode

Perhaps, by this point, you're itching to make Xcode better—you've seen what it can do, and now you want more! Sure, you can send your resume to Apple and work on the code behind the scenes; it seems they're always hiring.[1] In the meantime, while you're waiting for that callback interview, why not see how you can add your own functionality to Xcode right now.

To some degree, you already extended Xcode by providing your own IBDesignable views to storyboards, which you did in Chapter 3, *Storyboards: Behavior*, on page 43. You also added custom scripts to the build process in Chapter 5, *Building Projects*, on page 89. But what about the rest of Xcode?

The Xcode extensibility story is, unfortunately, a mixed bag. Prior to Xcode 8, there was a plug-in API that offered extensive customization throughout the IDE. However, it was completely undocumented and unsupported by Apple. Nevertheless, intrepid developers figured out how to make it do wonderful and crazy things. For example, the "Miku" plug-in[2] added the virtual pop star Hatsune Miku to the Xcode editor windows; her dancing sped up as you typed.



1. http://jobs.apple.com.
2. https://github.com/poboke/Miku

Sadly, Xcode 8 eliminated plug-in support, instantly disabling all of these clever hacks. In its place, Apple introduced a formal API for extending Xcode, with documentation and even a distribution strategy. That's the good news! The bad news is—so far—there's only one significant extension point.

In this chapter, you'll discover how to extend Xcode in the here and now, and you'll get a sense of how future Xcode extensions might work.

Editor Extensions

Xcode extensions use the *XcodeKit* framework, which contains protocols for extensions to implement, and data types sent to and from extensions. As of Xcode 9, the only extension point is the ability to add commands to the Editor menu while editing source code. The extension can access the contents of the file being edited, as well as the current text selection.

That's not a lot, but you can do interesting things with it. For example, some developers have used extensions to perform code formatting and other niceties. As a simple version of this, you can create an app extension to replace escape sequences for emoji with their actual emoji characters, like how Slack or Discord will replace :thumbsup: with the thumbs-up emoji.

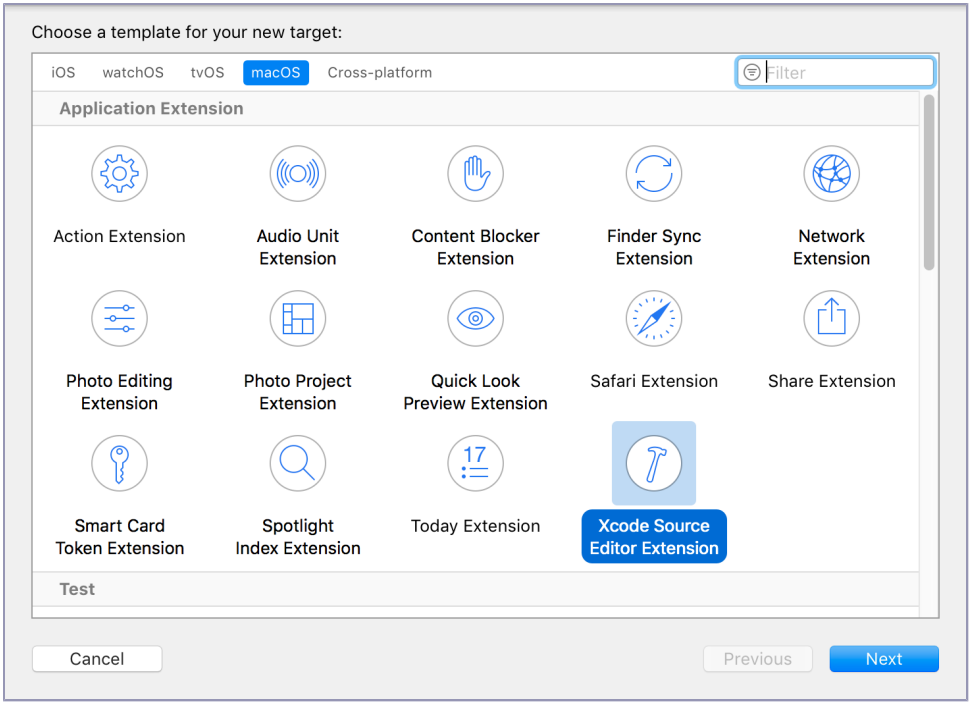
Xcode extensions use the existing App Extension system. You create a macOS app, and then implement the extension as an app extension target. Xcode will find the app extension and integrate it into its menus.

Creating and Configuring the Editor Extension

To create the app extension target, select the project from the File Navigator, and click the plus (+) button at the bottom of the projects and targets list. This brings up a sheet (seen in the figure on page 231) with the many types of targets you can add. The type needed for this is "Xcode Source Editor Extension":

This adds a new group with the extension's default files to the project. There are two source files, SourceEditorExtension and SourceEditorCommand, which implement the protocols XCSourceEditorExtension and XCSourceEditorCommand respectively. There's also an .entitlements file that sandboxes the app. This limits the extension's access to sensitive data and features like the filesystem or the network. If there's a good reason for the extension to get out of the sandbox, you'll need to add entitlement keys to this file.

The final file in the group is an Info.plist specific to the extension. It has a group of settings under NSExtension that define how the extension works, such as what the class names are (meaning this file must be updated if you change



the extension class names). One entry that you'll always want to change is XCSourceEditorCommandName, which provides a name for the menu item that will be added to Xcode for your extension:

▼ NSExtension	Dictionary	(2 items)
▼ NSExtensionAttributes	Dictionary	(2 items)
▼ XCSourceEditorCommandDefinitions	Array	(1 item)
▼ Item 0	Dictionary	(3 items)
XCSourceEditorCommandClassName	String	$(PRODUCT_MODULE_NAME).SourceEditorCommand
XCSourceEditorCommandIdentifier	String	$(PRODUCT_BUNDLE_IDENTIFIER).SourceEditorCommand
XCSourceEditorCommandName	String	Emojify! 😀
XCSourceEditorExtensionPrincipalClass	String	$(PRODUCT_MODULE_NAME).SourceEditorExtension
NSExtensionPointIdentifier	String	com.apple.dt.Xcode.extension.source-editor
Copyright (human-readable)	String	Copyright © 2017 Subsequently & Furthermore, Inc. All rights reserved.

Coding the Editor Extension

The two source files implement two protocols from XcodeKit. The SourceEditorExtension class is for lifecycle events, like overriding values from the Info.plist and getting a callback to indicate that the extension has loaded and launched.

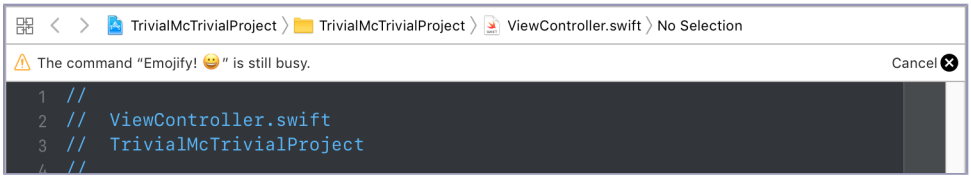
The simple extension you're building here doesn't need either of those, so instead, edit SourceEditorExtension. This has a single method, perform(with:completionHandler:), which is called when our menu item is invoked. It passes in a XCSourceEditorCommandInvocation, which is our access point to the text editor, and provides a completion handler that we call when we're done.

The invocation object has a XCSourceTextBuffer property named buffer. This is where you get access to the text buffer currently being edited. You can either get the completeBuffer as a single String, or iterate over its lines. You can also get the current selection as an array of XCSourceTextRanges, and get indentation information—for example, if the current file uses tabs or spaces, and how wide the indentation is.

For this extension, you just need to go through a list of replacement strings, and replace text in the buffer. First, you'll define the replacements as a private array inside the class (we've reproduced this code listing an image, since we can't be sure all the ebook readers out there support emoji):

```
private let replacementStrings: [String : String] = [
    ":thumbsup:" : "👍",
    ":thumbsdown:" : "👎",
    ":shipit:" : "🚢🇮🇹",
    ":beerme:" : "🍺"
]
```

One other thing that the invocation provides is a cancellationHandler. If the extension runs too long (currently about 3 seconds), a banner slides down from the top of the editor area, letting the user cancel the extension, as seen in the figure:



If they cancel it, this closure will be executed. As a simple implementation, you'll create a flag property named isCancelled to track this closure:

extending/EmojifierDemo/EmojifierExtension/SourceEditorCommand.swift

```
private var isCancelled: Bool = false
```

Now you're ready to code the perform(with:completionHandler:) method. It's a short listing, but it does a bunch of steps, which will be explained afterward:

extending/EmojifierDemo/EmojifierExtension/SourceEditorCommand.swift

```
Line 1 func perform(with invocation: XCSourceEditorCommandInvocation,
       -             completionHandler: @escaping (Error?) -> Void ) -> Void {
       -
       -     invocation.cancellationHandler = {[weak self] in
     5         NSSound.beep()
       -         self?.isCancelled = true
       -     }
       -
```

```
-     var buffer = invocation.buffer.completeBuffer
10    for (escapeString, emoji) in replacementStrings {
-         if isCancelled {
-             completionHandler(nil)
-             return
-         }
15        buffer = buffer.replacingOccurrences(of: escapeString,
-                                               with: emoji)
-     }
-     invocation.buffer.completeBuffer = buffer
-     completionHandler(nil)
20  }
```

- Start on lines 4–7 by setting up the cancellationHandler. If called, it simply beeps, and sets isCancelled to true. If this happens, you'll use the isCancelled value later.
- Line 9 copies the invocation's text buffer to a local variable named buffer, allowing you to mutate it.
- The for loop on lines 10–17 is where you really do the work, iterating over each of the possible substitutions. First, on lines 11–14, you check isCancelled each time through the loop, and if it has been set by the cancellation handler, you call the completion handler and do an early return. Keep in mind that the completion handler must *always* be called whether the extension succeeds, fails, or is cancelled.

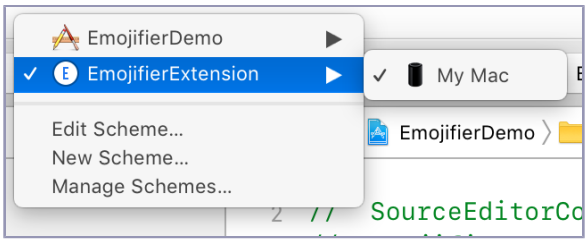
  On the other hand, assuming all is well, lines 15–16 do a string substitution of the escape string for its matching emoji.
- After the loop, you write the modified buffer object back to the invocation (line 18). And then you wrap up by calling the completion handler (line 19), passing in nil to indicate that no error has occurred.

That's it! You've extended Xcode's source editor. Now to take your emojifier out for a spin!

Running the Editor Extension

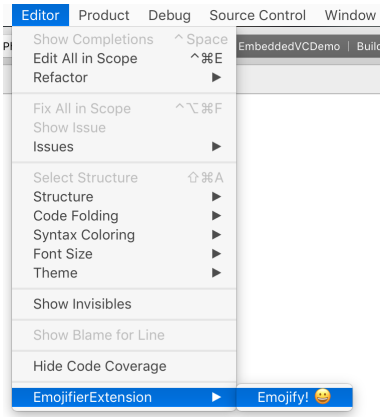
To run the editor extension, check the scheme selector and make sure the extension is selected and not the app, as seen in the figure on page 234. When you first created the extension target, Xcode asked if it should activate the extension's scheme, so if you clicked "Activate", you're all set. Otherwise, you need to change the scheme now.

When you click Run (⌘R), a sheet slides out asking which app should launch and host the extension. The recommended choice is Xcode, and of course this



is the only option that makes sense, so you should approve it. This launches a *second* instance of Xcode, separate from the one you're already working in, indicated by using a dark gray icon:

This second instance of Xcode works just like the original, but with one difference: the Editor menu now has a "EmojifierExtension" menu item. Inside is one submenu with the "Emojify!" command you created.



You can open a project, edit a file, and then use the menu item to convert all of the :emoji-name: escapes into their corresponding emoji. Furthermore, you can use Xcode's Preferences pane to add an Xcode-only keyboard shortcut for this menu item, making emoji goodness just a keystroke away.

Sure, it might be nicer if the extension can run on a timer or after every keystroke—so you can instantly replace the escape strings like Slack and Discord do—but it's still a nice start. And this kind of editor extension is particularly well suited to editor commands you might only run occasionally, like escape-encoding, syntax-cleaning, coding style enforcement, and so on.

Distributing the Extension

An extension isn't very interesting if you're the only one using it.

With Xcode extensions, there are two distribution options. The first is to put it on the Mac App Store, bundled inside an app. We touched on that already in *Automatic Code Signing for Distribution*, on page 179. However, in some ways, the MAS isn't ideal for an Xcode extension, because you have to create a sufficiently interesting parent app to pass App Review; maybe your extension is specific to a company or some other group and it isn't appropriate for world-wide distribution.

In that case, you can distribute the extension yourself by signing the app and sharing it via the web or some other file-sharing technique. This works as long as your users have "Allow apps downloaded from:" set to "App Store and identified developers" set in their Security & Privacy preferences.

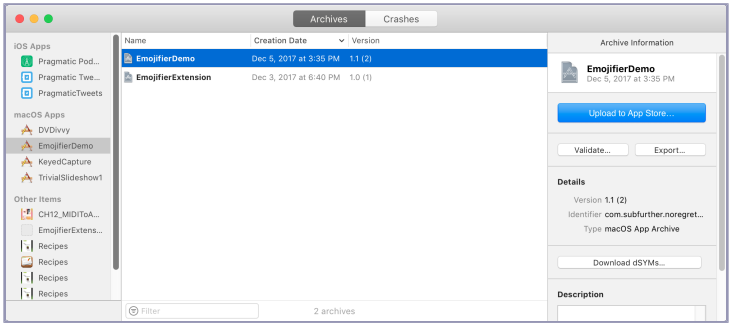
Getting a Developer ID

Of course, if your users are only going to run apps from "identified developers", you need to get identified. Xcode handles this for you pretty painlessly. If you haven't done this already for distributing other Mac apps, it's an automatic process the first time you need it. Start by archiving the app with Product > Archive, and make sure to change the scheme selector so you're building and archiving the whole app, not just the extension.

Versioning Multiple Targets

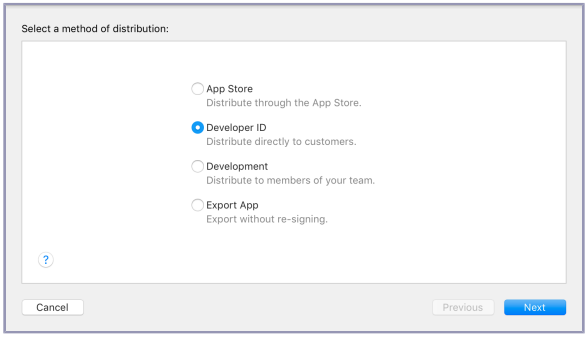
Keep in mind that both the app and the extension have their own Info.plists, which in turn means they have their own version and build numbers. You need to update both of these targets in the project editor, prior to archiving.

When the archive action is done, the Organizer window opens (as seen in the figure) and shows the newly archived app:

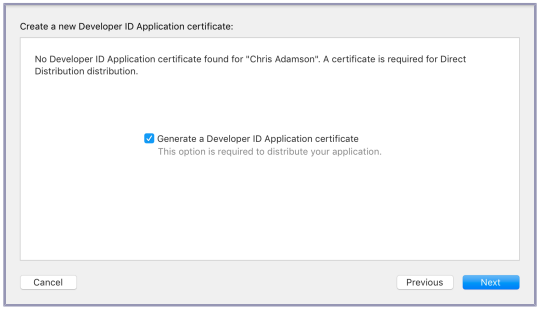


On the right, there are three buttons that act on the archive. The big, obvious one is for uploading to the Mac App Store, but notice that under this is one called “Export”. That’s what you want—you’ll export to a file, signing it with your developer credentials along the way.

After clicking “Export”, the first choice is for an export method, such as sending it to the App Store (jeez, they’re really pushing that, huh?), or just exporting the app file without applying any signing. The option you need here is “Developer ID”, which will sign it with credentials that specifically identify you as a trusted developer:

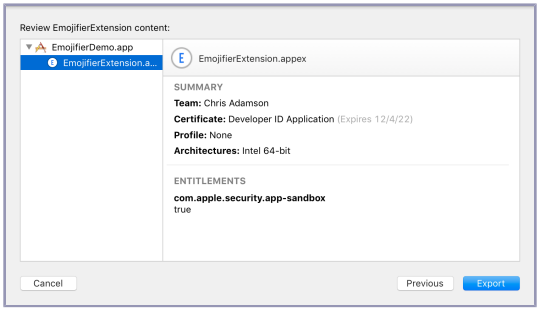


Next, the Organizer will say the app needs to be re-signed for direct distribution, and it offers a choice of automatically or manually managing signing. Unless you have some exotic signing needs—or you just like pain—go with automatic. The next step is where the Organizer will create a Developer ID certificate if there isn’t one already in the Keychain. Leave the checkmark selected and click “Next” to create the Developer ID certificate. It will be added to the Keychain, but since it can’t be recovered if lost, the Organizer adds an extra step allowing you to save a copy of the Developer ID certificate that you can store some place safe:



Prior to signing with the Developer ID certificate, the Organizer gives you a chance to review what will be signed. The left side of this sheet is a hierarchical list showing the parent app and the child extensions. Briefly review these to

reassure yourself that the extension is present and will get signed as part of the export. Then, go ahead and click “Export” and choose a destination:

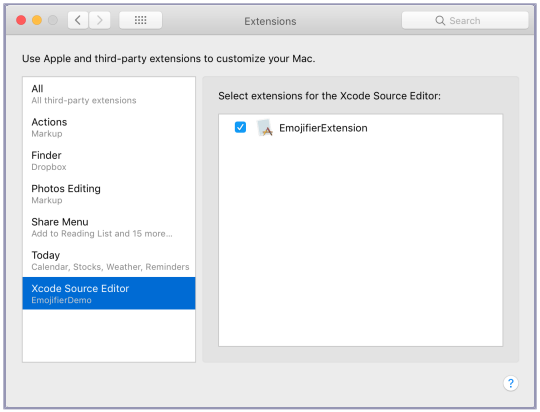


The result of the export is a folder containing the signed app, along with a Packaging.log file and two .plists describing the export options.

Running the Extension on Another Mac

The only file needed for distribution is the app itself. You can pass it around on USB drives if you’re sharing it face-to-face. But if you need to put it online, remember that Mac apps are bundles—folders with a known structure—so they don’t always play nice with web servers that don’t know to treat them as if they were files. A common way to deal with this is to simply zip the app, which you can do with the Finder menu item File > Compress “*Filename*”.

On the target machine, the app must be added to either the main /Applications folder, or a given user’s Applications folder. Once you’ve done this, System Preferences > Extensions will add a section for “Xcode Source Editor”, as seen in the figure. If you don’t see the new section, try running the app once. The editor extension will initially be disabled; you have to explicitly enable it with the check box in order for Xcode to notice it:



Once you do this, the extension will be available to Xcode. Open a project file, navigate to a source file, look to the bottom of the Editor menu, and you should see the “Emojify!” menu item.

Wrap-Up

Honestly, the prospect of Xcode extensions is a little more exciting than their current implementation, in that there’s only one extension point as of Xcode 9. And even then, you can only work with the source buffer’s contents and the current selection, not control how it’s rendered. To top it off, you can only get your code executed from a menu item invocation—you can’t set up code to run on a timer or on every keystroke.

But it’s a good sign that there is a proper framework for extending Xcode, supported by Apple with documentation and WWDC session videos. That means it’s more likely that it’ll continue to work, and even more hopeful that it will be more capable in the future. With any luck, future versions of Xcode may allow extension points to tie into storyboards, project files, the property list editor, and other parts of the IDE. I guess only time will tell.

Bibliography

[Cla17] Chris Adamson with Janie Clayton. *iOS 10 SDK Development*. The Pragmatic Bookshelf, Raleigh, NC, 2017.

[Gla16] Simon J. Gladman. *Core Image for Swift: Advanced Image Processing for iOS*. Reaction Diffusion Limited, London, United Kingdom, 1.3.1, 2016.

[KR98] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1998.

[Ste17] Daniel Steinberg. *A Swift Kickstart*. The Pragmatic Bookshelf, Raleigh, NC, 1.0, 2017.

[Swi10] Travis Swicegood. *Pragmatic Guide to Git*. The Pragmatic Bookshelf, Raleigh, NC, 2010.

Index

B

C

D

E

F

Q

R

S

T

X

Y

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2018 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

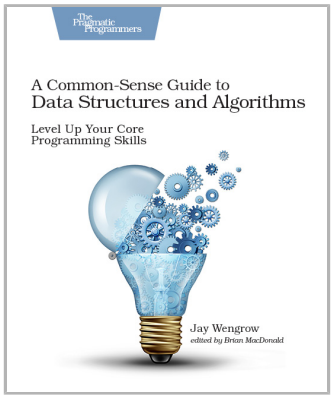
And thank you for your continued support,

Andy Hunt, Publisher

Level Up

From data structures to architecture and design, we have what you need.

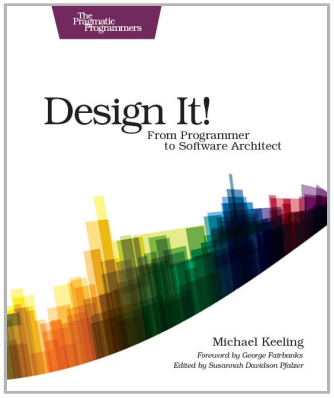
A Common-Sense Guide to Data Structures and Algorithms



If you last saw algorithms in a university course or at a job interview, you're missing out on what they can do for your code. Learn different sorting and searching techniques, and when to use each. Find out how to use recursion effectively. Discover structures for specialized applications, such as trees and graphs. Use Big O notation to decide which algorithms are best for your production environment. Beginners will learn how to use these techniques from the start, and experienced developers will rediscover approaches they may have forgotten.

Jay Wengrow
(220 pages) ISBN: 9781680502442. $45.95
https://pragprog.com/book/jwdsal

Design It!



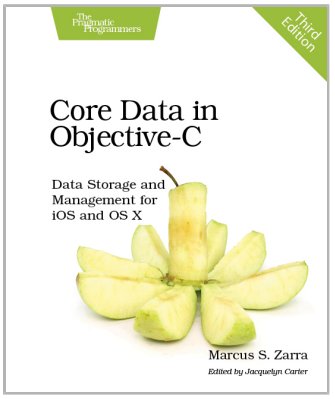
Don't engineer by coincidence—design it like you mean it! Grounded by fundamentals and filled with practical design methods, this is the perfect introduction to software architecture for programmers who are ready to grow their design skills. Ask the right stakeholders the right questions, explore design options, share your design decisions, and facilitate collaborative workshops that are fast, effective, and fun. Become a better programmer, leader, and designer. Use your new skills to lead your team in implementing software with the right capabilities—and develop awesome software!

Michael Keeling
(358 pages) ISBN: 9781680502091. $41.95
https://pragprog.com/book/mkdsa

Core Date

In Swift or Objective-C, we've got you covered.

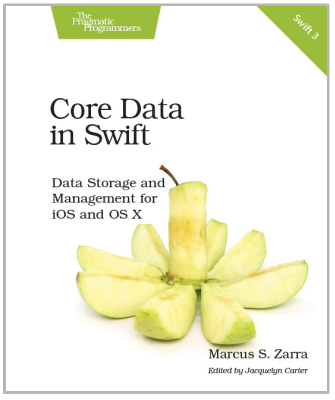
Core Data in Objective-C, Third Edition



Core Data is Apple's data storage framework: it's powerful, built-in, and can integrate with iCloud. Discover all of Core Data's powerful capabilities, learn fundamental principles including thread and memory management, and add Core Data to both your iOS and OS X projects. All examples in this edition are written in Objective-C and are based on OS X El Capitan and iOS 9.

Marcus S. Zarra
(238 pages) ISBN: 9781680501230. $38
https://pragprog.com/book/mzcd3

Core Data in Swift



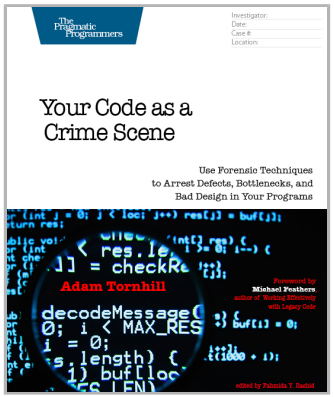
Core Data is intricate, powerful, and necessary. Discover the powerful capabilities integrated into Core Data, and how to use Core Data in your iOS and OS X projects. All examples are current for macOS Sierra, iOS 10, and the latest release of Core Data. All the code is written in Swift 3, including numerous examples of how best to integrate Core Data with Apple's newest programming language.

Marcus Zarra
(212 pages) ISBN: 9781680501704. $38
https://pragprog.com/book/mzswift

Pragmatic Programming

We'll show you how to be more pragmatic and effective, for new code and old.

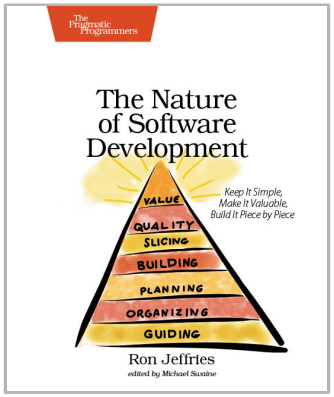
Your Code as a Crime Scene



Jack the Ripper and legacy codebases have more in common than you'd think. Inspired by forensic psychology methods, this book teaches you strategies to predict the future of your codebase, assess refactoring direction, and understand how your team influences the design. With its unique blend of forensic psychology and code analysis, this book arms you with the strategies you need, no matter what programming language you use.

Adam Tornhill
(218 pages) ISBN: 9781680500387. $36
https://pragprog.com/book/atcrime

The Nature of Software Development



You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning.

Ron Jeffries
(176 pages) ISBN: 9781941222379. $24
https://pragprog.com/book/rjnsd

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/caxcode
Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date
https://pragprog.com
Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/caxcode*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764