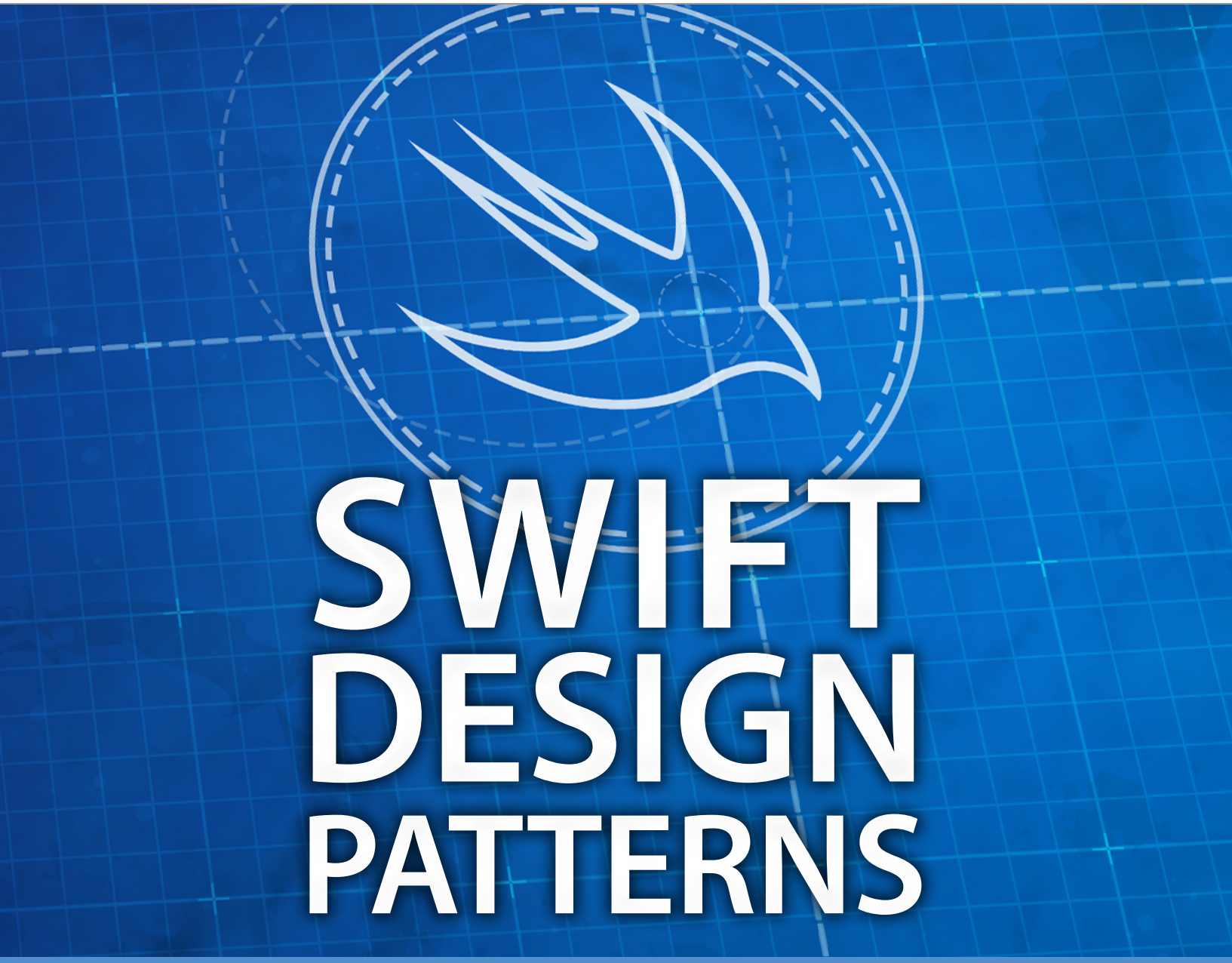
HACKING WITH SWIFT



SWIFT DESIGN PATTERNS

COMPLETE REFERENCE GUIDE

Learn smart, idiomatic ways to design great Swift apps

Paul Hudson

Swift Design Patterns

Paul Hudson

Version: 20181029

Contents

Preface

Welcome

If you ask ten people what it means to write "great code" you will probably get ten different answers. Is it code that runs fastest? Is it code that is easiest to read? Is it code that is easiest to *maintain*? Is it short code? Must great code be "elegant" or "clever"? Is there perhaps One True Way to solve common problems?

The answer is of course: "Yes." Great code might be all of those things, or even none of them. However, I think there is one term that can be used to describe great code accurately, and that is "thoughtful". Great code is *thoughtful* code: when you've had the time and space to work through your problem, decide on an approach, then write code to solve it effectively.

Maybe your goal was to have fast code, maybe your goal was to have concise code, maybe your goal was to have that ran on every platform that runs Swift – it doesn't really matter, because everyone's situation is different. However, when you're faced with a hard problem it's usually the case that writing code is the easiest part of your job.

Unless you're very junior, sitting down and typing conditions, loops, functions, and more is something that comes easily to you. The real problem lies before: working through your problem and trying to figure out your approach.

This book is not designed to teach you Swift, and it's not designed to teach you iOS, macOS, or any other specific platform. Instead, this book details a wide variety of time-tested, battle-proven, approaches to solving common computer science problems using Swift.

I don't mean "how do you sort an array?" or "what do I name things?", but architectural problems: how to structure projects so they are easier to maintain, how to design data types so they are reusable, how to take full advantage of Swift so your code is flexible and efficient, and more.

More importantly, this book also provides the *why* that is so often lacking in learning material. A major goal of this book is to teach you *why* Apple built Cocoa and Cocoa Touch in specific ways, and why following their approach will help you be a better code.

By the end of the book you'll be armed with new concepts, new techniques, and even a new vocabulary that will aid the way you approach future coding problems. Of course, concepts themselves don't ship apps, so I've provided dozens of comprehensive code examples you can apply to your own work – I strongly encourage you to try them out yourself and toy around with them a little, otherwise they are likely to be forgotten as quickly as they were learned!

What are design patterns?

Regardless of how you approach coding, everyone agrees that code ought to be reusable. That might mean creating a class hierarchy, it might mean using protocol-oriented programming, it might mean using generics, or any number of other techniques – they all make code useful in more than once place.

A design pattern is the ability to reuse *ideas*. It's the ability to say "last time we hit a problem similar to this, we solved it by…" and then outlining a solution you know to have worked previously. You never remember the exact code you wrote, but you *do* remember your structure, the advantages, the disadvantages, and any implications for the rest of your project.

A design pattern is the ability to say to say, "we can solve this using a template method," and have everyone else understand what you mean – and understand the same advantages, disadvantages, and implications that you had in your head. We all take this kind of shared vocabulary for granted when talking about things like classes, generics, and extensions – things that have entirely different meanings outside of programming – and it's just as useful when applied to ideas.

Most importantly, design patterns are techniques to help you write thoughtful code. The patterns presented in this book are the rock solid foundations of our industry: for decades they have been tested and refined in projects big and small around the world. When you have them in your toolkit you'll be able to consider draw on proven solutions as you face challenges, then apply whichever one makes most sense for your project.

The war on coupling

I've covered a range of design patterns in this book, but you're unlikely to use them all in a single app unless you enjoy torturing yourself. However, even though they solve different problems they do all share a common theme: they try to uncouple code.

Coupling is a measure of how connected one piece of code is with another – how dependent they are on each other. Code that is tightly coupled is code where everything relies on everything else, where information flows in every direction, and where taking out one piece of code is like removing the bottom card in a house of cards. Tightly coupled code is hard to reuse because often there's little to no separation of concerns, which means one file might end up with 100 methods that bare little to no relation to each other. Worse, tightly coupled code is hard to reason about, because often a developer needs to understand a huge part of the system in order to make a small change.

On the other hand, loosely coupled code strives to produce modules that are independent from each other, that can be moved, changed, or replaced easily. You can think of two objects as being loosely coupled when they communicate through strict public interfaces, and when changes in either object don't require you to change the other as well.

This clear use of public interfaces – a contract describing what methods and properties you expose – is central to achieving low coupling, because it draws a clear line between what you expose externally (your interface) and how you achieve that internally (your implementation). If everything goes to plan, you should be able to change your implementations freely without affecting any other part of your project, because as long as your interface stays the same – as long as you stick to the contract you made with other parts of your project – everything will carry on working.

You should already know that Swift is an aggressively value-oriented language, by which I mean it uses value types rather than reference types for nearly everything. Structs, bools, integers, floats, strings, arrays, dictionaries, sets, are all value implemented using value semantics in Swift, and that actually does wonders for decreasing coupling.

If you didn't already know, a value type is something that only ever has one unique owner. If you create a string and assign it as a property to two different objects, the first object can

change the string freely without it affecting the value held by the second object. In contrast, classes are reference types in Swift – not value types – which means that the same object can have multiple owners, and if any of them change the value then it changes everywhere.

Swift's extensive use of value types gives us two immediate and significant boosts towards lowering coupling. First, because structs only ever have one owner it means you aren't creating a mess of implicit dependencies in your code – you aren't enabling a situation where three different view controllers all point to the same property, and any of them can change it for everyone at any time. Second, structs do not support subclassing, which is a win for everyone.

Yes, you read that correctly: despite subclassing being a fundamental skill for Apple developers, it's broadly a bad idea because it produces the tightest possible coupling. A child class is utterly reliant on its parent class for its implementation, and one small change way up in a class hierarchy could ripple downwards to cause huge breakage.

Swift's ability to use protocol-oriented programming means we can – and should – favor composition over inheritance, and that's only one of the many ways we can reduce the need for subclassing.

Note: Sometimes I see people trying desperately to use structs when classes really are the best solution. This is particularly prominent when working with Objective-C code, which is pretty much all code that interacts with Apple's APIs. As you progress through this book you'll find that the platform design patterns often use classes with **@objc** attributes because that's the way Apple's frameworks are designed, whereas the Swift language design patterns are able to rely more heavily on natural, idiomatic Swift.

Avoiding spaghetti code

Spaghetti code is code that is tangled, confused, and messy – pretty much the antithesis of what we're aiming for in this book. However, there is another equally problematic issue developers hit when they try to apply design patterns a little too eagerly: they write *ravioli code*.

If spaghetti code is a messy jumble of code, ravioli code is dozens if not hundreds of discrete

code packages. That might sound good, but the truth is that an extremely fine level of granularity – the extent to which you break up your code into small chunks – makes code harder to understand and debug.

Like so many things, the optimal granularity for any given project depends on so many factors that you can't easily draw conclusions. However, one rule I hope you can commit to memory is this: abstraction is a means, not an end. That is, we don't strive for highly reusable code because we love building flowcharts and delivering tech talks, but instead because it allows us to deliver world-class products to our users faster and more reliably.

Patterns in practice

Design patterns are not an Apple invention – far from it! In fact, the canonical reference to design patterns is a book that was published back in 1994: "Design Patterns: Elements of Reusable Object-Oriented Software." This book formalized 23 software design patterns that were in use at the time, giving examples in C++ and Smalltalk had four authors. It had four authors, so it became known as the Gang of Four book.

Here's what Erich Gamma, one of the four authors, had to say about the usefulness of patterns: "I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior."

Although design patterns are not an Apple invention, there *are* many patterns that are very deeply embedded in Apple's platforms. The concept of *delegation*, for example, is baked deep into Apple APIs, to the point where you really can't get much done without knowing how it works.

These design patterns invisibly bind together Apple's APIs, and have done for decades. Interface Builder was first introduced in NeXTSTEP 30 years ago, alongside many of the "NS" classes we still know and love. And yet that code can and does live happily alongside other code written only a year ago, despite the fact that the intervening period saw the introduction of Mac OS X (now macOS), iPhoneOS (now iOS), and Swift.

This continuity is made possible by Apple's consistent application of its design patterns. No matter how the code has changed – and it has changed to be almost unrecognizable, even for Objective-C developers – the ideas and techniques have not. As a result, if you take the time to learn which patterns Apple applies and why they do so, you'll find coding for iOS, macOS, watchOS, and tvOS comes far more naturally.

Note: to avoid having to say "iOS, macOS, watchOS, and tvOS" repeatedly I'll just refer to "Apple platforms" or "Apple developers".

Objective-C and Swift

Swift was written from scratch to be "Objective-C without the C", so it's no surprise that Objective-C influenced the language greatly and continues to do so. In many ways this means that the history of Swift is the history of Objective-C: if you want to understand why Swift works the way it does and how to leverage that best, you can start to feel a bit like a code archaeologist.

Even though this book has "Swift" right in its title, I haven't shied away from Objective-C. I want you to try to write natural, idiomatic Swift, which among other things means using structs rather than classes, and using protocol-oriented programming rather than object-oriented programming. But this book is about helping you write great Swift apps, not helping you write great theoretical code, and that means playing by the rules enforced by all the Objective-C code that powers Apple's APIs.

Whenever possible, this book goes with a pure Swift approach that uses value types and protocols – two of the bedrocks of idiomatic Swift. However, when dealing with platform patterns that force us to work in a certain way, or when specific language restrictions allow no other options, classes and **@objc** are used. As I said, my goal is to help you write great Swift apps, so we need to be pragmatic.

How this book is structured

I've already touched on some of the patterns from the Gang of Four book – "program to an

interface, not an implementation", and "favor object composition over class inheritance" – but this book isn't just a Swift port of that book's 23 patterns.

Don't get me wrong: the 23 Gang of Four patterns are all conceptually interesting, and undoubtedly have a long history of success. However, some of them just aren't appropriate for Swift development – they don't sit naturally with Swift development, either because of the way Swift works or because of the way Apple has structured its Objective-C APIs.

So, this book is made up of five distinct sections:

- Section 1 introduces the Model View Controller (MVC) architecture, which is the fundamental architecture for all Apple development.
- Section 2 introduces the Model View ViewModel architecture, which is an increasingly popular alternative to MVC.
- Section 3 introduces design patterns that are common to Apple platforms, including delegation, notifications, associative storage, and more. These often sit very poorly alongside idiomatic Swift, but we are rather stuck with them.
- Section 4 introduces design patterns that are idiomatic to Swift, including extensions, protocol extensions, and keypaths. This is where pure Swift gets its chance to shine.
- Section 5 introduces classic design patterns that are directly useful to Apple developers today, including singletons, template methods, facades, and more.

So, two major architectures up front, followed by platform patterns, language patterns, and classic patterns. Where applicable, I've pointed out how patterns are the same as or similar to patterns from the Gang of Four book because it may help make your knowledge more transferrable, but it was not my goal to try to squeeze Smalltalk pegs into Swift holes.

Although this book is designed to be platform-agnostic, I was obviously keen to draw examples from real Apple code where possible. Where that happens I've used iOS as it's the platform most folks are familiar with, but I've tried to include some macOS examples too.

Frequent Flyer Club

You can buy Swift tutorials from anywhere, but I'm pleased, proud, and very grateful that you chose mine. I want to say thank you, and the best way I have of doing that is by giving you bonus content above and beyond what you paid for – you deserve it!

Every book contains a word that unlocks bonus content for Frequent Flyer Club members. The word for this book is **OMEGA**. Enter that word, along with words from any other Hacking with Swift books, here: **https://www.hackingwithswift.com/frequent-flyer**

Dedication

This book is dedicated to my friends Anthony and Eunie. Sometimes writing books is easy, but a lot of the time it’s hard. No matter what, these two have been there with their help, support, encouragement, and love.

Chapter 1

MVC

Start with Apple's preferred software architecture.

Model-View-Controller

The Model-View-Controller architecture (MVC) was invented in the 70s by Smalltalk developers. However, because Smalltalk heavily influenced Objective-C at NeXTSTEP, and NeXTSTEP eventually got folded into Apple, MVC became the de facto standard for Apple platform development.

In this chapter I want to introduce you to the concepts behind MVC and how they are applied in Apple development, discuss the major advantages and disadvantages of MVC, then walk you through example code that demonstrates how to clean up common MVC mistakes.

I realize that history and naming can sound dull, but please trust me on this: coming in the world of software architecture with fresh eyes can really confuse you. When acronyms like MVC, MVP, MVVM, MVVMC, and VIPER get tossed around extensively, the distinction between them can start to get lost. So, we're going to start with the basics so you can understand how we got here, then move on.

The fundamentals of MVC

Like I said already, MVC is old, predating both Swift and even Objective-C. That doesn't mean it's necessarily *bad*, in fact quite the opposite: the fact that an architecture invented before iOS, before Windows, and even before MS-DOS is still useful and popular today shows you how good it is. MVC is used extensively on Apple platforms, but it's also hugely popular on Windows, the web, and beyond, which makes extensive knowledge of MVC a transferable asset as your career develops.

You don't need to look far to figure out *why* MVC is so popular, because there are only two reasons.

First, it separates the concerns of your architecture so that each component does only one thing: stores data (Model), displays that data (V), or responds to user and manipulates the model appropriately (C).

Second, you can understand it in seconds. You just read a single sentence above that explains

the core of MVC, which makes it easy to explain, easy to learn, and easy to apply.

These two things combine powerfully: separation of concerns in your code is both explicit and easy to do, and teams can work together on code without treading on each others' toes. As a result, MVC has lasted about 40 years already, and I think it's likely to last another 40 or more.

What my description *doesn't* include is how the three M-V-C components ought to talk to each other. A view is nothing without data, for example, but how should the two communicate?

It won't surprise you to learn that I didn't leave off such information by accident – how these three components communicate can easily descend into a religious war, because MVC doesn't actually tell us that part. Instead it's been left open to interpretation and reinterpretation over the years, to the point where what many people consider MVC is not at all MVC.

For now, let's stick with the classic definition of MVC, as outlined by its author Trygve Reenskaug in 1979.

First, models. Here's how Reenskaug defined them: "Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects." This is the easiest component to understand, because it has a one-to-one mapping with real-world concepts like **User**, **Recipe**, and **Book**.

Second, views. Here's how they are defined by Reenskaug: "A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter. A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages." So, views are visual manifestations of the data behind them, and they both read from and write to their model directly.

Finally, controllers. From Reenskaug again: "A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output,

translates it into the appropriate messages and pass these messages on to one or more of the views." So, controllers provide the user the ability to control their app, and translate those instructions into messages for the views.

To make things just a teensy bit muddy, Reenskaug outlines a specialized form of controller called the *editor* that "permits the user to modify the information that is presented by the view."

(All quotes taken from Reenskaug, Trygve, December 1979, *Models-Views-Controllers*.)

So models are data, views are representations of that data, and controllers exist to arrange views, shuttle user input back and forth, and edit data. In other words:

- The user uses the controller.
- The controller manipulates the model as an editor.
- The model sends its data to the view.
- The view renders itself to the user.

There might be a few other steps in there – for example your model might apply data validation such as constraining ranges – but that's more or less it. If done well, all three components are neatly decoupled: neither the view and the model have any sort of coupling on the controller, so you can switch them around as needed.

Does that sound like the MVC you use in your application? Almost certainly not. But now that you understand the fundamentals of MVC we can look at how it's employed on Apple platforms.

How Apple developers uses MVC

As soon as you create a new application for iOS, macOS, or tvOS, you're immediately faced with view controllers. The existence of these things – *their very name* – tells you that Apple has been Thinking Different about more than just their marketing.

Think about this for a moment: how often do you create your views for your apps? The

chances are you do it fairly rarely, if ever – we rely on **UIView** on iOS/tvOS and **NSView** on macOS. How often do you make models that send their data straight to whatever view is representing them? Again, the chances are you do it fairly rarely if ever.

Instead, Apple developers tend to write applications a bit like this:

- Models are structs or classes holding a handful of properties.
- Views are almost exclusively taken straight from UIKit or AppKit (WatchKit is quite different).
- Controllers contain everything else.

And that "everything else" is huge – view controllers are commonly responsible for manipulating models and precisely configuring views to display them, handling network code, running animations, acting as data sources, responding to user input, loading and saving, and much more.

As a result, MVC has become retroactively named "massive view controller" because that's often the end result: view controllers that are thousand of lines long.

Now to be fair Apple hasn't done much to correct this problem. While regular view controllers are confused enough, its popular subclasses like **UITableViewController** couple both the view and controller tightly together so that the controller is effectively juggling responsibilities.

Even with its tendency to balloon out of control, MVC the way it's used by Apple really is the standard architecture for iOS apps. This means three things:

1. Apple's sample code, including its Xcode templates, all assume you're using MVC.
2. Most other developers will use MVC. If you join a team and work on their existing iOS app, it's probably based around MVC. If you open a book about Apple development, it's probably based around MVC. If you find some sample code online and want to use it, it probably also assumes you're using MVC.
3. If you don't use MVC, you risk fighting Apple's built-in APIs that are all designed around

MVC.

Sometimes it doesn't matter how appealing alternative architectures are, sometimes MVC is the best choice. This is particularly true for smaller apps, where the work required to set up an alternative is almost as much work as building the app.

Model-View-Presenter

As you've seen, the way we use MVC on Apple platforms is distinctively different from the way MVC was designed. This shouldn't come as a shock: MVC was codified at a time when graphical user interfaces were still young, and using a language that is quite alien to Swift.

A similar-but-different alternative is called Model-View-Presenter (MVP), and looks like this:

- The Model stores data that will be displayed to the user.
- The View stores passive user interface elements that forward user interaction to the presenter so it can act on it.
- The Presenter reads data from the model, formats it, then displays it in views.

Sound familiar? Although it's not an exact match, I think you'll agree that Apple's implementation of MVC is much more similar to *MVP*.

Advantages and Disadvantages

Although MVC is the default choice for most developers, that doesn't mean you should use it uncritically. To help you decide what's right for your project, I've tried to put together some advantages and disadvantages to give you a better idea of what you're working with.

Advantages

The biggest advantage of MVC is that it's a well-known pattern implemented in quite literally millions of programs. Although the definition of MVC will always vary slightly from person to person and platform to platform, the core tenets remain the same. This gives MVC an enviable position as being one of the few software architectures in the world that are recognizable by everyone – if you say "we should use MVC", the other developers in the room will understand the meaning and implications of that. The same can rarely be said for MVVM, MVVMC, VIPER, Elm, or other architectures.

Second, MVC separates responsibilities so clearly that even a beginner can be sure what role something has. Put simply, if it draws to the screen it's a view, if it stores data then it's a model, and if it doesn't draw to the screen or store data then it's a controller. This is a clear separation of concerns that makes it easy for everyone to get started, and lets us subdivide larger applications into digestible chunks – at least at first.

Third, whether or not you use classic MVC or Apple's modified MVP, it has a clear data flow. In Apple development the controller updates the model and responds to state changes, while also updating the view and responding to user events. The controller becomes the ultimate arbiter of truth, with both views and models being thin receptacles.

Finally, MVC is how UIKit, AppKit, and WatchKit are designed. Even though we see weird hydras like "view controller", that's where view-related events such as **viewDidLoad()** and **viewDidAppear()** get sent so that's where they ought be dealt with. View controllers are the inevitable hub of all events in Apple's world, which is why adding code there is so easy.

Disadvantages

The biggest disadvantage of MVC is that our view controllers frequently end up huge, confusing balls of mud. Yes, it's easy to add code to the view controller, but that doesn't mean it's the best place for it. It's not uncommon to find view controllers conforming to three, four, five, or even more different protocols, which means one view controller is forced to act as a data source for a table view, a collection view, a navigation controller, a WebKit navigation delegate, and more.

One of my favorite speakers is Soroush Khanlou, who had this to say about large controllers: "When you call something a Controller, it absolves you of the need to separate your concerns. Nothing is out of scope, since its purpose is to control things. Your code quickly devolves into a procedure, reaching deep into other objects to query their state and manipulate them from afar. Boundless, it begins absorbing responsibilities."

Second, the separation of concerns can sometimes be blurred. Should code to format data before presentation go into the model, the view, or the controller? A particular culprit is the **cellForRowAt** method of table views – whose job should it be to create and prepare cells for display?

Third, view controllers are notoriously hard to test. Even without your application logic in place, testing a view controller means trying to interact with UIKit controls and from experience I can promise you that's an absolute nightmare. Even though you might try your best to keep things separate, any controller code that encapsulates any *knowledge* – anything more than sending a simple value back in a method – will be harder to test when it touches the user interface.

Fourth, MVC often leads to small views and models. The point of MVC is that it divides responsibilities over three independent components, but in practice there's usually a tiny M, an even smaller V, and a huge C – the division is hugely slanted towards the controller, to the point that you'll often see folks doing all their layout in code as part of their view controller's **viewDidLoad()**.

Finally, MVC doesn't adequately answer questions like "where do I put networking?" or "how

do I control flow in my app?" This isn't an MVC problem – the same problem exists in MVVM and others.

Fixing MVC

There is a logical fallacy known as the No True Scotsman fallacy, which is when you make a sweeping generalization, have it disproved by counter-examples, then adjust the generalization so that it excludes the counter-examples.

Wikipedia has an example that I find particular pleasing because my own Scottish father has tried it on me:

- Person A: “No Scotsman puts sugar on his porridge.”
- Person B: “But my uncle Angus likes sugar with his porridge.”
- Person A: “Ah yes, but no *true* Scotsman puts sugar on his porridge.”

(In case you were curious, the correct way to make porridge is with salt. Apparently.)

A version of this same fallacy is often seen when people defend MVC. It goes something like this:

- Person A: “MVC architecture neatly separate models, views, and controllers.”
- Person B: “But all these apps put view code into the controller.”
- Person A: “Ah yes, but that’s not a *true* MVC architecture.”

Those apps might not represent the pure definition of MVC architecture, but they certainly represent how it’s used in practice.

In this chapter I want to look at four ways to improve bad MVC code. My goal is *not* to make you write MVC in the way Trygve Reenskaug defined it 40 years ago, because we’ve moved on since then. Instead, I want to use Apple’s implementation of MVC as a target: what can we do to identify problematic code, then refactor it so it’s cleaner, clearer, and more in tune with what Apple intended?

One view controller to rule them all

The C in MVC stands for “controller”, but that doesn’t mean you need to have precisely one

controller in your app, or even one controller per screen. You can have as many controllers as you like, and they can work together, work independently, or a mix of the two. Dave DeLong, a long-time Apple engineer with extensive experience across multiple frameworks, said this: "the principle behind fixing Massive View Controller is to unlearn a concept that's inadvertently drilled in to new developers' heads: 1 View Controller does not equal 1 screen of content."

I should clarify three things. First, that concept is not inadvertently drilled in to new developers' heads – it's *actively* drilled in there. Second, I've personally taught this understanding to thousands of students, so in some respects I'm part of the problem. However, third is that this understanding is actually useful when you're just getting started – it's a helpful way to explain the concept to users when they are just starting out, because they can look at the Mail app on their phone and see view controllers being pushed on and off a stack.

So, I don't think the old "1 view controller equals 1 screen of content" is a bad thing to get started with, but it *does* become bad as your application grows. Apple has provided us with specific, powerful tools such as view controller containment to help break screens up into more manageable chunks, but these get remarkably low usage – developers continue to be attracted to the Ball of Mud methodology.

This gives us our first target for bad MVC apps: can extremely large view controllers be split into smaller controllers using view controller containment? The code required is almost trivial – adding a child view controller takes only four steps:

1. Call **addChildViewController()** on your parent view controller, passing in your child.
2. Set the child's frame to whatever you need, if you're using frames.
3. Add the child's view to your main view, along with any Auto Layout constraints if you're using them.
4. Call **didMove(toParentViewController:)** on the child, passing in your main view controller.

In Swift code it looks like this:

```
addChildViewController(child)
child.view.frame = frame
view.addSubview(child.view)
child.didMove(toParentViewController: self)
```

When you're finished with it, the steps are conceptually similar but in reverse:

1. Call **willMove(toParentViewController:)**, passing in **nil**.
2. Remove the child view from its parent.
3. Call **removeFromParentViewController()** on the child.

In code, it's just three lines:

```
willMove(toParentViewController: nil)
view.removeFromSuperview()
removeFromParentViewController()
```

Just for convenience you might want to consider adding a small, private extension to **UIViewController** to do these tasks for you – they do need to be run in a precise order, which is easily done incorrectly.

Something like this ought to do it:

```
@nonobjc extension UIViewController {
    func add(_ child: UIViewController, frame: CGRect? = nil) {
        addChildViewController(child)

        if let frame = frame {
            child.view.frame = frame
        }

        view.addSubview(child.view)
        child.didMove(toParentViewController: self)
```

```
    }

    func remove() {
        willMove(toParentViewController: nil)
        view.removeFromSuperview()
        removeFromParentViewController()
    }
}
```

That's marked **@nonobjc** so it won't conflict with any of Apple's own code, now or in the future.

The next step is to divide work across view controllers, and the easiest place to start there is by splitting your controllers up by functionality.

Back when iOS 5 was introduced it came with a feature called Newsstand – a dedicated shelf where users could buy magazine apps. At the time I was running a team of seven at a media company, and we took it on ourselves to launch all 60 magazine brands onto Newsstand ready for iOS 5's arrival – we had less than two months to develop everything from scratch, which was guaranteed to be hectic.

I had done most of the coding to read magazines, others were working on making the Newsstand APIs work and adding a back-issue library, and one person was working on the storefront. When the first draft of the storefront was ready to test, we did the usual "test your app like you hate your app" fuzz testing that all smart developers do, and it was pretty awful – if you tapped a Buy button five times quickly then five purchases would begin, and the app would crash.

Obviously we didn't want users to cause such problems, so I asked the storefront developer to dim the screen as soon as purchasing started, and show an activity indicator on top. This happened in time for the second test run and made things a lot better, but only later did I notice how it was achieved: everywhere the dimming was required a subview was being created with a spinner on top, then removed when the purchase either finished or failed.

This meant two things. First, we had a lot of code duplication on our hands, which is never a good thing. Second, our store's view controller grew huge as it tried to handle all possible success and failure situations in a single place.

This is a situation that's crying out for view controller containment. To be fair to the developer in question, view controller containment was only introduced in iOS 5 so there was no way he would have any experience of it, but developers today have no excuse.

In this situation – any time where you need to overlay one view controller temporarily over another – view controller containment becomes the easiest solution. You can even wrap the whole thing in a closure, like this:

```
guard let vc =
storyboard?.instantiateViewController(withIdentifier: "Second")
else { return }

add(vc)

performSlowWork(with: data) {
    vc.remove()
}
```

In so many cases this literally means you can cut code from one view controller and paste it into another. The grand total amount of code hasn't changed, but your responsibilities are clearer and I guarantee the code is easier to maintain.

You can take this further: two or three child view controllers can work side by side in the same parent view controller, similar to how **UISplitViewController** works on the iPad. If the top half of your controller is only loosely linked to the bottom half, why put them all in one?

(In case you were wondering, we did launch all 60 onto Newsstand in time for the iOS 5 launch, despite Apple coming to visit us and saying "we're so glad you're supporting iPhone, because most others aren't" – iPhone support was apparently promised by an executive. On

launch day over half of all Newsstand apps were from our company, and we won a national award for our efforts.)

Delegation of responsibilities

Why must your view controller be responsible for network requests? Why must your view controller be responsible for loading and saving data? Why must it be the data source for your table view controller and your picker?

The answer of course is that there is no good reason – at least for anything beyond simple apps. Yet it's so common to see code like this:

```
class ViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, UIPickerViewDataSource,
UIPickerViewDelegate, UITextFieldDelegate,
WKNavigationDelegate, URLSessionDownloadDelegate {
```

What does that class *do*? Apple has already muddied the water enough by giving us "view controllers", but now this poor view controller is being asked to do almost everything by itself. This kind of code *works*, of course it does, but if you're making any given view controller conform to more than one or two protocols then you automatically lose the right to complain about any massive view controllers in that project.

The solution here is to create data source and delegate objects where it makes sense. Usually any protocol with "DataSource" in its name is a good target to be carved off, because you can create a dedicated object that reads your model and responds appropriately.

Yes, this makes your view controller smaller, and yes it helps you get towards the goal of each object having a single responsibility, but the biggest win here is testability. Once you have a dedicated object that consumes model data, adjusts it according to your application logic, then sends the data back as a data source, you can write tests that exclude UIKit – you can create some mock data, hand it to your data source class for processing, then write tests to make sure it's worked correctly.

Let's look at a practical example: you want to embed a **WKWebView** that enables access to only a handful of websites that have been deemed safe for kids. In a naïve implementation you would add **WKNavigationDelegate** to your view controller, give it a **childFriendlySites** array as a property, then write a delegate method something like this:

```
func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
    if let host = navigationAction.request.url?.host {
        if childFriendlySites.contains(where: host.contains) {
            decisionHandler(.allow)
            return
        }
    }

    decisionHandler(.cancel)
}
```

(If you haven't used **contains(where:)** before, you should really read my book Pro Swift.)

To reiterate, that approach is perfectly fine when you're building a small app, because either you're just learning and need momentum, or because you're building a prototype and just want to see what works.

However, for any larger apps – particularly those suffering from massive view controllers – you should split this kind of code into its own type:

1. Create a new Swift class called **ChildFriendlyWebDelegate**. This needs to inherit from **NSObject** so it can work with WebKit, and conform to **WKNavigationDelegate**.
2. Add an import for WebKit to the file.
3. Place your **childFriendlySites** property and navigation delegate code in there.
4. Create an instance of **ChildFriendlyWebDelegate** in your view controller, and make it the navigation delegate of your web view.

Here's a simple implementation of just that:

```
import Foundation
import WebKit

class ChildFriendlyWebDelegate: NSObject, WKNavigationDelegate
{
   var childFriendlySites = ["apple.com", "google.com"]

   func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
      if let host = navigationAction.request.url?.host {
         if childFriendlySites.contains(where: host.contains) {
            decisionHandler(.allow)
            return
         }
      }

      decisionHandler(.cancel)
   }
}
```

That solves the same problem, while neatly carving off a discrete chunk from our view controller. But you can – and should – go a step further, like this:

```
func isAllowed(url: URL?) -> Bool {
   guard let host = url?.host else { return false }

   if childFriendlySites.contains(where: host.contains) {
      return true
   }
```

```
    return false
}

func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
    if isAllowed(url: navigationAction.request.url) {
        decisionHandler(.allow)
    } else {
        decisionHandler(.cancel)
    }
}
```

That separates your business logic ("is this website allowed?") from WebKit, which means you can now write tests without trying to mock up a **WKWebView**. I said it previously but it's worth repeating: any controller code that encapsulates any *knowledge* – anything more than sending a simple value back in a method – will be harder to test when it touches the user interface. In this refactored code, all the knowledge is stored in the **isAllowed()** method, so it's easy to test.

This change has introduced another, more subtle but no less important improvement to our app: if you want a child's guardian to enter their passcode to unlock the full web, you can now enable that just by setting **webView.navigationDelegate** to **nil** so that all sites are allowed.

The end result is a simpler view controller, more testable code, and more flexible functionality – why *wouldn't* you carve off functionality like this?

Coding your user interface

A second group who have lost all privileges to complain about massive view controllers are those people who insist on creating their views in code and doing so inside **viewDidLoad()**. I have no problem with people creating UI in code, and in fact it's the smart thing to do for a

great many projects, but just cast your eyes over this monstrosity:

```
backgroundColor = UIColor(white: 0.9, alpha: 1)

let stackView = UIStackView()
stackView.translatesAutoresizingMaskIntoConstraints = false
stackView.spacing = 10
view.addSubview(stackView)

stackView.topAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.topAnchor).isActive = true
stackView.leadingAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.leadingAnchor).isActive = true
stackView.trailingAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.trailingAnchor).isActive = true
stackView.axis = .vertical

let notice = UILabel()
notice.numberOfLines = 0
notice.text = "Your child has attempted to share the following
photo from the camera:"
stackView.addArrangedSubview(notice)

let imageView = UIImageView(image: shareImage)
stackView.addArrangedSubview(imageView)

let prompt = UILabel()
prompt.numberOfLines = 0
prompt.text = "What do you want to do?"
stackView.addArrangedSubview(prompt)

for option in ["Always Allow", "Allow Once", "Deny", "Manage
Settings"] {
```

```
    let button = UIButton(type: .system)
    button.setTitle(option, for: .normal)
    stackView.addArrangedSubview(button)
}
```

That's not even a complex user interface, but it's the kind of thing you'll see in **viewDidLoad()** even though that's a *terrible* place to put it.

All the code above – literally all of it – is *view* code, and needs to be treated as such. It is not controller code, and even with Apple's muddled definition it is not *view controller* code either. It's view code, and belongs in a subclass of **UIView**.

This change is trivial to make: you copy all that code, paste it into a new subclass of **UIView** called **SharePromptView**, then change the class of your view controller's view to your new subclass.

The final **SharePromptView** class should look something like this:

```
class SharePromptView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        createSubviews()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        createSubviews()
    }

    func createSubviews() {
        // all the layout code from above
    }
}
```

All **UIView** subclasses must implement **init(coder:)**, but as you're creating your UI in code you will also need to add **init(frame:)**. The **createSubviews()** method is there to support both.

Thanks to that custom **UIView** subclass you can now take a huge amount of code out of your view controller:

```
class ViewController: UIViewController {
    var shareView = SharePromptView()

    override func loadView() {
        view = shareView
    }
}
```

Having a dedicated **shareView** property allows you to access any properties you declare inside **SharePromptView** without having to keep casting **view**.

Avoiding the app delegate

Of all the places that get mightily abused, the **AppDelegate** class is king, queen, chief, and emperor. When you think to yourself, "I need to create this data and share it in all sorts of places," your **AppDelegate** class does not jump up and down shouting "pick me, pick me!"

In fact it's quite the opposite: this class is created to handle launching your app and responding to specific system events, and even if you handle only a few of those you're already talking about 100-200 lines of code.

This class is a particularly bad dumping ground for developers who create their user interface in code. This isn't restricted to developers who write *all* their user interfaces in code – just setting up a tab bar controller can force you to do this.

Here's a simple rule you can try following: if you're putting something into **AppDelegate** that isn't a simple implementation of the **UIApplicationDelegate** protocol, it should almost

certainly be its own controller.

Summary

Massive view controller syndrome is real, but it doesn't *need* to be. I've just outlined four common ways you can make your view controllers smaller, more testable, and closer to the goal of single responsibility:

- View controller containment lets us create simpler view controllers that get composed into one larger one.
- Creating dedicated classes to act as data sources and delegates helps isolate functionality and increase testability.
- Coding your user interface is perfectly fine, but do it in a **UIView** subclass.
- Don't dump code in the app delegate – it's almost never the right place for it.

Chapter 2

MVVM

Explore an increasingly popular alternative to MVC.

Model-View-ViewModel

There are several alternatives to the MVC architecture, but the only one with any widespread adoption is called the Model-View-ViewModel architecture (MVVM). This was invented by Microsoft back in 2005 as a new way to build apps using its Windows Presentation Foundation (WPF) framework, but its use has spread a lot further since then.

In this chapter I want to introduce you to the concepts behind MVVM and how they are applied in Apple development, discuss the major advantages and disadvantages of MVVM, then walk you through example code that demonstrates how to put MVVM into practice in your own apps.

The fundamentals of MVVM

I want to make it clear immediately that MVVM is not that different from MVC. Microsoft describes it as a "variation" and "refinement" of MVC, so I hope that by going into some detail on how MVC and MVP works you'll be able to understand MVVM relatively easily.

In MVC we split things up by roles: if something stores data it's a model, if something renders data it's a view, and everything else is a controller – an easy approach to learn, but hard to put into practice without ending up with a big ball of mud.

We need to put Apple's view controllers into one of those three buckets, so most people say "it's a controller" and use it as a dumping ground. But it doesn't need to be that way, and in 2004 Martin Fowler suggested an alternative: introduce a new of object called a *presentation model* that stores the state of your app independently of your user interface.

Warning: Even though they sound similar, the Presentation Model architecture is not the same as Model-View-Presenter architecture.

Using presentation models requires you to write classes that are able to represent your view fully but *without* any sort of UIKit or AppKit attachment. It should be able to read your model data and all the transformations that are needed in order to prepare that data for presentation – without actually adding **import UIKit**. This immediately means you ought to be able to write

tests for everything your presentation model does, because your presentation model can be fed any data you like and respond with something you can check.

In the presentation model Apple's view controllers are part of the *view* system rather than the *controller* system – their only job is to respond to all the standard events Apple posts there, such as **viewDidLoad()**. Some developers prefer an approach to presentation model where view controllers respond to events rather than the presentation model, and in practice I think this approach works more easily on Apple platforms.

Even though the presentation model delivers an immediate improvement in testability, it introduces a significant complexity: if all your presentation models store the current application state, and all your views display application state, how do you synchronize the two?

This is where MVVM comes in: it was designed to take the Presentation Model architecture and attach a system of data-binding to remove the need to write boilerplate code connecting your views and your presentation models. Microsoft says the same: "in fact, pretty much the only difference is the explicit use of the data-binding capabilities of WPF and Silverlight."

To be clear: MVVM is the Presentation Model with the addition of some functionality enabled by the Windows Presentation Foundation framework. It doesn't rely on WPF (far from it!) but WPF makes MVVM natural because it has support for data binding – the ability to connect a text field in your view to a string in your model so that changing either one updates the other.

This means that MVVM benefits from the same testability as the Presentation Model, without the same boilerplate problems of having to synchronize the various parts of your app. John Gossman, the Microsoft developer who invented MVVM, said "the ViewModel, though it sounds View-ish is really more Model-ish, and that means you can test it without awkward UI automation and interaction. If you've ever tried to unit test UI code, you know how hard that can be."

How Apple developers use MVVM

Whether you call it "presentation model" or a "view model", you've seen how separating

presentation *data* from views enables a level of testability that was difficult if not before, and MVVM was designed to deliver that using data bindings present in WPF.

I've said WPF several times so far because it directly affects how Apple developers use MVVM. It shouldn't come as a surprise that a technology called Windows Presentation Foundation is not available on Apple's platforms, which means the bindings that make MVVM work are a completely alien concept to most of us.

I say "most of us" because bindings *do* exist on macOS: you can tell text fields to retrieve from values model data, and any changes will automatically be synchronized. These bindings have never been ported to iOS, and indeed aren't used even by many macOS developers.

macOS bindings suffer from three major problems:

1. They rely heavily on the Objective-C runtime. In order to use them you must use classes, then mark the properties of those classes with both **@objc** and **@dynamic**
2. They are stringly typed: you type strings into Interface Builder that match properties inside your code, so it's easy to make mistakes and hard to figure out why.
3. They are completely opaque: values are adjusted automatically and without you really know why or how, which makes debugging extremely hard.

As a result, we're faced with the curious situation of iOS developers crying out for bindings so they can get MVVM to work, and macOS developers running away from bindings screaming.

All this matters because iOS/tvOS developers still need to have bindings otherwise MVVM becomes painful, and macOS developers would love to have the power of MVVM bindings just without the black box complexity and un-Swifty nature of macOS's native bindings.

So, we're faced with three choices:

1. Write your own binding code that keeps your view model in sync with your views.
2. Use a library such as Bond, which lets you say how values should bound to views.
3. Implement something large like RxSwift, which includes binding as part of its framework.

Of the three, only really the first two are viable options for most of us – RxSwift is such an incredibly different way of working that using it just to get bindings is a bad move.

Once you decide how you'll implement bindings, the rest is more or less straightforward. Your view controller becomes responsible for presenting layouts and binding the components in those layouts to fields inside your view model, and all the business logic you previous had there gets moved into your view model. How much logic you put in your view model and how much in your model depends on you, but that's the same dilemma you can face in MVC.

MVVM is only a little harder to explain than MVC, but it is much hard to adopt thanks to the lack of bindings on Apple platforms. John Gossman describes MVVM as "overkill" for simple user interfaces – this goes doubly so on Apple platforms.

I don't think you'll find anyone, even MVVM advocates, who thinks that MVVM is perfect, but that's OK. In many ways MVVM represents an improvement over MVC, and we shouldn't let the perfect become the enemy of the good.

Advantages and Disadvantages

MVVM makes a great choice for many projects, and its usefulness only increases as your project grows. I've already walked you through the major advantages and disadvantages of MVC, and it's only fair to do the same for MVVM.

Advantages

By far away the biggest advantage of MVVM is its testability. This goes two ways: you can now write much simpler unit tests for most of your app because you can call directly into the view model, and you can also provide your (extremely small) view controllers with mock view models using stubbed data to check that they all work correctly without having to feed identical live data.

Second, it frees up the view controller to do what it was meant to do: focus on view lifecycle events. Yes, you might just have transferred 80% of your code somewhere else, and that "somewhere else" might end up being just as much of a dumping ground as your view controller was, but it's an improvement.

Third, similar to MVC I think it's fair to say that MVVM separates responsibilities reasonably well enough that beginners can understand it: models store data, views store visual representations of data, and everything else goes into the view model. At least with MVVM putting "everything else" into the view model feels dubious, so developers might be inspired to split it off.

Finally, MVVM has a clear data flow, as long as you don't make the bindings too hard. You connect your view model directly to your views using two-way binding, and your view model manipulates your model. The view controller does next to nothing, as nature intended.

Disadvantages

In the same way that MVVM has one major advantage, it also has one major disadvantage: unless you enjoy the monotony of writing boilerplate code to synchronize your view model

and views, you need bindings. These enable changes in your data to reflect on screen, and changes on screen to be updated in your data. Bindings aren't hard to write as you'll see in the next chapter, but there's a good chance you'll switch to a framework once you understand the concept, at which point your complexity increases.

Second, although MVVM does help alleviate the problem of small views – largely because view controllers are lumped into the "view" category – it doesn't do much for small models. It's a matter for debate how much code goes into models, but in both MVVM and MVC apps you'll usually find the answer is "not much."

Third, MVVM introduces a lot of complexity for small projects. None of Apple's templates use MVVM, so you need to introduce bindings yourself every time – a bit of a waste of effort if you're just working on a prototype. Even if you don't introduce bindings, you still need to shuttle data to and from your view model yourself, which is unpleasant.

Finally, although MVVM frees up our free view controllers so they are less muddled, developers still use view models as dumping grounds for all sorts of unrelated code. You can expect to find networking code, input/output, validation, and more. This is a similar problem to MVC, but at least with MVC you can argue that your networking code can go in a dedicated controller – that just seems *weird* with view models.

Bindings in Practice

Now that you should be pretty clear on when MVVM makes sense (and, as importantly, when it doesn't), I want to look at what it takes to make it work in a real project. Ultimately there's a very high chance you'll end up relying on a framework like Bond because it does so much on your behalf – even though we'll be walking through a lot of code now, it's only a fraction of what Bond does.

That doesn't mean this chapter is useless – in fact it's quite the opposite. Making the jump from MVC to Bond can be quite a shock (with the jump to RxSwift being even more so!). If you're able to step through bindings at a simpler level first – using code you wrote directly into Xcode – it will massively reduce the shock, and hopefully make MVVM much easier to understand.

The point of MVVM's bindings is to stop you having to copy data from your view to your view model, and from your view model back to your view. This known as *two-way binding* because the data flows in both directions.

In this chapter I'm going to demonstrate two different ways of creating two-way bindings: the harder, Swifty way, and the easier, un-Swifty way. The idiomatic Swift approach is significantly better than the alternative, un-Swifty approach, so I hope you'll bear with it!

Creating an Observable type

Create a new iOS app using the Single View App template, then give it the following trivial struct:

```
struct User {
    var name: String
}
```

We're only testing here, so you can put that in ViewController.swift if you like – just make sure it's *outside* the **ViewController** class to avoid problems.

Our job, in this most simple of examples, is to make a text field that synchronizes its text with the **name** value in that struct. It's one view controller, one model object, and one property, so this really is the most simple example we can construct. We need a view to work with, so open Main.storyboard and give it a single text field. Switch to the assistant editor, then create an outlet for that text field called **username**.

Swift structs do not have any general-purpose way of observing changes to their values. You can add a **didSet** property observer, of course, but that's a highly *specific* way of observing changes – you need to add that for every value you intend to watch.

Instead, we're going to create an **Observable** class that is designed to monitor one value. It's a class here because it allows us to mutate it freely as you'll see, but once your binding skills are honed you'll probably want to switch to a struct for the extra immutability it provides – or just move to Bond.

Our **Observable** type needs to be generic, because even though it's just going to handle strings in a text field here you could easily extend it to other types in the future.

Let's start with the basics – add this below your **User** struct:

```
class Observable<ObservedType> {
    private var _value: ObservedType?

    init(_ value: ObservedType) {
        _value = value
    }
}
```

That creates our generic type, giving it one property to store that value, and an initializer so that we can create it easily. Note that the stored value is private: we don't want other folks touch this by accident.

A private value *matters* because of two-way bindings: if the text field changes it needs to update the model, and if the model changes it needs to update the text field. This can easily

lead to an endless loop, because whenever one change it notifies the other, which then notifies the first that it's changing, which notifies the second, and so on.

The easiest way to avoid this is to create a **valueChanged** property in the **Observable** class, which gets called whenever the value changed. This closure will point to some code in our view that we'll write later, which is where we'll place the new value into the text field.

Add this property to **Observable**:

```
var valueChanged: ((ObservedType?) -> ())?
```

That will send the current value to whatever is watching it.

With that in place we can now create a **value** property in **Observable** where the stored value can be manipulated safely. This is different to the **_value** property, which is private – this one is designed to be adjusted from anywhere, and will both change **_value** and send its new value out to the observer using its **valueChanged** closure.

This is easy enough thanks to property observers – add this to the **Observable** class now:

```
public var value: ObservedType? {
    get {
        return _value
    }

    set {
        _value = newValue
        valueChanged?(_value)
    }
}
```

The other thing our class cares about is when the other side of its binding has changed – when the text field has its text changed by the user. Remember, this should automatically update the value we're storing, but it *mustn't* do so using **value** otherwise the property observer will be

triggered and you'll be in a loop.

Fortunately, this method belongs to **Observable** so we can just modify its **_value** in place – that won't trigger the **didSet** property observer.

Add this method to **Observable**:

```
func bindingChanged(to newValue: ObservedType) {
    _value = newValue
    print("Value is now \(newValue)")
}
```

I added a call to **print()** to make things a little easier to follow – you'll see it's usefulness soon enough!

Tip: Another common way of breaking the change loop between observer and observable is to notify the other only when the new value is different to your existing one.

Now that we have a type dedicated to observing changes, we can modify the **User** struct to use it. Previously it was defined like this:

```
struct User {
    var name: String
}
```

Now we need to use an **Observable<String>** instead, so please change it to this:

```
struct User {
    var name: Observable<String>
}
```

That's our **Observable** type complete, so we can now move on to the text field. We need to find a way to monitor changes in the text field so that they report back to our model, and also monitor changes in the model so that they update the text field. This is complicated somewhat

by the fact that you can't attach closures to **UITextField** (or indeed any **UIControl**), so we're going to implement a workaround.

Now in the cases of text fields this is straightforward. They have a dedicated **editingChanged** event that is triggered by a user's touch, which means it shouldn't ever get itself into a binding loop even without our private property in **Observable**. That doesn't mean the private setter approach is wrong, it just means that when you extend it later you won't hit problems immediately.

We need to connect the text field's **editingChanged** event to a closure that will update the model value with the new text. As we are unable to do that directly, the workaround is to subclass **UITextField**, have it call a dedicated **valueChanged()** method when the **editingChanged** event happens, which can then call the closure.

Start by adding this new class underneath **Observable**:

```
class BoundTextField: UITextField {
    var changedClosure: (() -> ())?
}
```

That **changedClosure** is what we'll call when the **editingChanged** event happens. Because we can't call it directly from the text field, we need a little *thunk* method – a method that is able to bridge the world of Objective-C and Swift.

Add this to **BoundTextField**:

```
@objc func valueChanged() {
    changedClosure?()
}
```

That's a pure Objective-C method, which means we can attach it as a selector for our button, but all it does is pass the call directly to the **changedClosure()** call.

We need to add one more method, which is the one that does all the hard work: **bind()**. This

will:

1. Accept any **Observable** value that stores a string underneath, because that's what our text fields work with.
2. Add the text field as its own handler for the the **editingChanged** event, pointing it at the **valueChanged()** method we just wrote.
3. Set the text field's **changedClosure** to some code that calls the **bindingChanged()** method on its observed object.
4. Set the observable's **valueChanged** closure to some code that updates the text in the text field.

Add this method to **BoundTextField** now:

```
func bind(to observable: Observable<String>) {
    addTarget(self, action:
#selector(BoundTextField.valueChanged), for: .editingChanged)

    changedClosure = { [weak self] in
        observable.bindingChanged(to: self?.text ?? "")
    }

    observable.valueChanged = { [weak self] newValue in
        self?.text = newValue
    }
}
```

That completes the two types we need to make simple binding work. In a more advanced implementation you could attach your closure directly using an Objective-C runtime method called **objc_setAssociatedObject()**, but if you're considering doing that you might want to look at trying Bond first.

Now that we have something that can observe and something that can be observed, we need to put the two together in our view controller. Earlier we created a text field called **username**, but

it was a plain text field – a **UITextField** rather than our new **BoundTextField**. So, we need to make two changes:

1. Open your storyboard, select the text field, then use the identity inspector to change its class to "BoundTextField".
2. Back in ViewController.swift, change **var username: UITextField** to be **var username: BoundTextField**.

The final step is the easy one, and if you were using any sort of binding library it would pretty much be the *only* one: we need to create some data and bind it to our text field.

First, give the **ViewController** class this property so that it has some data to work with:

```
var user = User(name: Observable("Paul Hudson"))
```

Now add this call to **bind()** in the **viewDidLoad()** method:

```
username.bind(to: user.name)
```

That's it – that's all the code we need in our view controller to make the whole bindings system work. This should give you a glimpse of what view controllers are like when you're using MVVM: they present their views (a text field, in our case), bind those views to data, then do little more than respond to lifecycle events.

Go ahead and run the app now. You should be able to type into the text field to see your changes printed in Xcode's log – that's a result of the **print()** call we added inside the **bindingChanged()** of **Observable**.

If you want to check the bindings work in the other direction, try adding this to **viewDidLoad()**:

```
DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    self.user.name.value = "Bilbo Baggins"
}
```

That updates the model username to be “Bilbo Baggins”, which will in turn cause the new value to appear in the text field.

The alternative: key-value observing

There is an alternative way to monitor changes, but it’s so very ugly and so very un-Swifty that I was in two minds about including it here. However, sometimes seeing what *not* to do is as useful as seeing *what* to do, so I want to walk you through the alternative at least partly.

Objective-C has a system of observing values called key-value observing (KVO). It’s covered in detail in the Keypaths chapter of this book, but I want to touch on it briefly because you could use it to at least partly solve the problem of bindings.

Objective-C’s KVO is available to us as Swift developers, but only in highly specific conditions are satisfied:

- You use a class.
- Your class inherits from **NSObject**.
- All your observable types are visible to Objective-C. (No structs or enums.)
- All your observable types are marked **@objc**.
- All your observable types are marked **dynamic**.

Then and only then can you use KVO. It would transform our **User** struct into this:

```
class User: NSObject {
    @objc dynamic var name: String

    init(name: String) {
        self.name = name
    }
}
```

You can now observe the **name** property directly by adding this code to **viewDidLoad()**:

```
let observer = user.observe(\User.name, options: .new) { [weak self] user, change in
    self?.username.text = change.newValue ?? ""
}
```

That **observer** value needs to be stored somewhere, otherwise the observation stops. For example, you might add a property like this:

```
var observers = [NSKeyValueObservation]()
```

You can then call **observers.append(observer)** to store the observer away safely.

However, this only solves one half of the problem: if our data changes, the view will get updated, but how do we make the view update our data? Well, UIKit has no good solution here: it wasn't designed to be KVO-compliant even though it can often be used as such. In this case, there isn't even a hacky solution because the **text** property of **UITextField** isn't even vaguely KVO-compliant, which means you still need to go back and create a custom subclass that catches changes the old-fashioned way.

I hope at this point you can see why KVO is a pretty awful approach to bindings. It might be conceptually interesting to explore, but beyond that you're effectively writing Objective-C in Swift!

Where does the View Model come in?

In this simple example we've only looked at the MV part of MVVM, effectively pretending the View Model component doesn't exist. The view model represents a *super set* of what you've seen here: rather than managing only a single text field with a string, the view model is designed to wrap the entire model with many more properties.

In a real app your view model would include formatting of values (for example converting a **Date** into a string ready for display), archiving and unarchiving of data, and perhaps even

networking code. It's not uncommon to add validation of data from the view, making sure this falls in line with what you expect or allow – I'd personally put this in the model, but I've seen it done both ways.

The goal is that your view model should wrap your model entirely – the view layer should have no concept of the model existing. When building your view model, remember one of the big wins is to make it testable without involvement from UIKit or AppKit: your view controller should take your **UITextField** (or **UISwitch**, **UIButton**, etc) and send its *value* into the view model rather than the object itself.

Summary

If you read through this chapter and though it sounded hard, you're right: it *is* hard. If it were easy, MVVM would probably already be the default architecture for Apple platforms. The need for bindings really is a disadvantage of MVVM either because you craft them yourself (non-trivial for anything serious) or import a third-party dependency to do it for you.

You could if you wanted attempt MVVM without bindings, but then by definition it's not MVVM. Instead, you're probably closer to the Presentation Model – that's not necessarily bad, but I think it's important to be aware of the distinction.

Chapter 3
Platform Patterns

Design patterns we inherited from Objective-C APIs.

Delegation

If you were to ask an Apple developer what one key design pattern is most closely associated with their platforms, I think the answer would almost certainly be delegation. Although it does exist on other platforms, it's baked so deep into Apple platforms that it's hard to make any meaningful apps without using it.

A delegate is one object that is asked to respond to events that happen to another object, or to guide its behavior. The example most people meet first is with **UITableView**/**NSTableView**: when you assign its **delegate** property the table view will tell your custom object when events happen, such as the user selecting a cell.

Think about it: when did you last try to subclass a table view? Chances are it was along time ago if ever, because it just isn't needed – using delegation your custom object and the table view work together as one, even though the table view needs to have no knowledge of what your object is or how it works.

Table views are ubiquitous user interface elements on Apple platforms, and yet I could show you ten apps that customize them so much they look almost hand-crafted. This shows the true power of delegation: Apple's table view code is a generalized class designed to handle all manner of table-related presentation, whereas your delegate object is unique to the coding problem you're facing.

At its core, delegation makes it easy to customize behavior of objects while also minimizing coupling. However, it's often misused to create monstrous controllers that attempt to act as delegates to multiple things – they might all require delegates, but must they all use the same object for their delegate?

If you're faced with a massive view controller, try this simple search in your code:

```
someObject.delegate = self
```

Every match that finds is a chance for you to take advantage of the delegation pattern and refactor your code. That **delegate** property must point to something, but it can – and should –

point to a custom object specifically designed to act as a delegate.

Building a delegate

There are a few rules you should follow when working with delegation:

1. Name your delegate property **delegate** if there is only one of them, or put **delegate** at the end if there is more than one. For example, **WKWebView** has **uiDelegate** and **navigationDelegate** properties that can point to two different objects.
2. Decide on the correct level of granularity before you write any code – which are the corrects points to notify your delegate? Remember, there is a small cost for each check you make to see whether you have a delegate and whether it cares about the event in question.
3. Follow Apple's standard approach to avoid verb conjugation. This means many of your method names will start with **will**, **did**, and **should** – with the latter checking for a boolean return value.
4. Although you can require a specific data type for your delegate, it lowers your coupling to specify a protocol instead. So, before you create your data type you should start with a protocol first.
5. Where possible, allow your delegate to implement only the methods it wishes, either by marking methods optional or by providing sensible default implementations. If you have no required methods, your object should work well enough without a delegate.

That last point is particularly important in Swift, because we don't have the ability to create protocols with optional methods without bringing in Objective-C and classes. This is covered more in the Protocols chapter; we'll be working with required methods here.

To demonstrate delegation, we're going to design part of a calendar control that will report various events to its delegate. You can run all this code in a Swift playground, so go ahead and create one now.

First, we need to create a **CalendarDelegate** protocol. Remember, delegates are there to respond to events from and guide the behavior of something else, so this involves thinking about how different kinds of users might want to use our control.

Obviously we're only going to implement a tiny part of the calendar here, but even so you might think to define something like this:

```
protocol CalendarDelegate {
    func willDisplay(year: Int)
    func didSelect(date: Date)
    func shouldChangeYear() -> Bool
}
```

That lets calendar delegates update their UI when a year has been shown, take some action when the user selected a date, and control whether the year should be changeable – it's an example of **will**, **did**, and **should**.

However, that protocol isn't practical for one major reason: what happens if users have multiple calendars on the screen? Those protocol methods don't say which calendar sent the change, which makes it hard to take appropriate action.

As a result, Apple delegates have a very specific naming convention that often reads a bit like you're stuck in the movie Being John Malkovich:

```
func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath)
```

So, "table view table view table view, did select row at index path index path" – I vividly remember seeing the Objective-C version of that when I first learned to make iPhoneOS apps, and found it utterly baffling. However, really it just answers two questions: what row was selected, and which table was it selected on.

Using Apple's delegate naming convention isn't too hard. For example, **willDisplay(year: Int)** becomes this:

```
func calendar(_ calendar: Calendar, willDisplay year: Int)
```

And **didSelect(date: Date)** becomes this:

```
func calendar(_ calendar: Calendar, didSelect date: Date)
```

The only different part is **shouldChangeYear()**, because that accepts no parameters other than the sender and so needs to be named as follows:

```
func calendarShouldChangeYear(_ calendar: Calendar) -> Bool
```

Protocol naming is covered further in the Protocols chapter.

The finished product is this:

```
protocol CalendarDelegate: class {
    func calendar(_ calendar: Calendar, willDisplay year: Int)
    func calendar(_ calendar: Calendar, didSelect date: Date)
    func calendarShouldChangeYear(_ calendar: Calendar) -> Bool
}
```

Note: Using the **class** keyword for the protocol restricts it to be used only with classes. This in turn allows us to store instances of the delegate using the **weak** keyword to avoid retain cycles.

The next step is to create a **Calendar** data type that calls its **delegate** property at appropriate times. This is the kind of thing that users might want to subclass, although to avoid tight coupling we'll obviously do our best to limit the need for that.

Again, we're not going to design a full calendar class here because that would rather miss the point, so instead we're going to limit it to a few things:

1. A **delegate** property. This needs to be a weak optional so we can clear it to avoid retain cycles.
2. A **selectedDate** property to track whichever date the user currently has selected.
3. A **currentYear** property to track which year is currently showing.
4. A **changeDate()** method that will be called when the user selects a date.

5. A **changeYear()** method that will be called when the user navigates forwards or backwards a year.

Both those methods would be called by some sort of visual layout that is outside the remit of this book. Here's the skeleton of a **Calendar** class:

```
class Calendar {
    weak var delegate: CalendarDelegate?
    var selectedDate: Date = Date()
    var currentYear: Int = 2018

    func changeDate(to date: Date) {
        selectedDate = date
    }

    func changeYear(to year: Int) {
        currentYear = year
    }
}
```

The next step is to insert our three delegate methods: **didSelect**, **willDisplay**, and **calendarShouldChangeYear**. The first one is easy enough, because it should be called as soon as **selectedDate** changes. So, add this to the end of **changeDate()**:

```
delegate?.calendar(self, didSelect: date)
```

Our **delegate** property is optional, which is why we need to use **delegate?.calendar()** rather than just **delegate.calendar** – if there is no delegate set this method call will just be ignored. Notice that the calendar passes itself as the first parameter, so the delegate knows which calendar triggered the event.

The next two methods require a little more thinking. First, the **currentYear** property should only be changed and the **willDisplay** call should only be sent if the year is actually changing.

However, our delegate might prefer to disallow the year changing by returning **false** from **calendarShouldChangeYear()**. So, we need to wrap that functionality inside a check with the delegate.

There's a small catch, though: **calendarShouldChangeYear()** normally returns a boolean, but because we're calling it on an optional we're actually going to get back an optional boolean – it could be true, false, or nil. This is rarely a data type you want to see, so to avoid problems we're going to use nil coalescing to provide a sensible default value: if there is no delegate, assume that changing years is allowed.

Once that check passes, the rest is easy enough: call **willDisplay** on the delegate so that it prepares to update its date, then change the **currentYear** property.

Here's the revised method:

```
func changeYear(to year: Int) {
    if delegate?.calendarShouldChangeYear(self) ?? true {
        delegate?.calendar(self, willDisplay: year)
        currentYear = year
    }
}
```

Now that we have a protocol and **Calendar** class in place, we can try it out by creating a **Reminders** class. In most apps this would probably start as a **UIViewController** or similar, but if you're facing a massive view controller this would be a great candidate to carve off into its own separate object. Note: if you'd prefer to use a struct here I've added instructions for that at the end of the chapter.

Here's a basic **Reminders** class:

```
class Reminders: CalendarDelegate {
    var title = "Year: 2018"
    var calendar = Calendar()
```

```
    init() {
        calendar.delegate = self
    }
}
```

Xcode will complain immediately because we've said that conforms to **CalendarDelegate** but haven't implemented its three required methods.

Add these now:

```
func calendarShouldChangeYear(_ calendar: Calendar) -> Bool {
    return true
}

func calendar(_ calendar: Calendar, willDisplay year: Int) {
    title = "Year: \(year)"
}

func calendar(_ calendar: Calendar, didSelect date: Date) {
    print("You selected \(date)")
}
```

That doesn't do much (by design), but it does make our **Reminders** class fully conform to **CalendarDelegate**.

Data sources

Data sources are a specialized form of delegation and have a lot in common with regular delegates, although they are subtly different. As you've seen, delegates are there to respond to events from and guide the behavior of something else, but data sources are there to provide data rather than control.

The benefits of separate data sources are identical to those of delegates: we get loose coupling

through the use of a protocol, which allows one generalized object and one specialized object to work together in tandem. However, it's important to resist the temptation to merge your delegate and data source protocols: the two serve different purposes and there is no benefit to coupling them.

Data sources can be thought of as the "M" in MVC, but really they act as a thin layer on top of your model to provide any shaping you need. Previously we developed a trivial **Reminders** class so that you could see a delegate example from scratch, and while we could extend the class to support a calendar data source it would limit our options.

Our model - however you're storing your data – is fixed and inert, like it ought to be. However, because the data source acts as a layer between the model and our **Reminders** class it gives us a chance to shape the data: should we display only important reminders? Should we display reminders in a chunky card-based layout or show them in slimline rows?

That might seem trivial, but I've lost track of how many times I've seen code like this:

```
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
    if userOption == "whatever" {
        // 30 lines of layout code
    } else {
        // 30 different lines of layout code
    }
}
```

That might have started off sensibly, but now it's just ignoring the power of delegation. I've already said that there's no reason to put your data source code right inside your view controller, but if you have two or more completely different ways of responding to data source or delegate methods then you *definitely* should.

In this case the code is trying to lay out a **UITableView** according to one set of rules if a certain user option is set, and a different set of rules otherwise – and the developer who wrote

it has probably also written a blog post about why MVC leads to massive view controllers.

In this case, a much better solution is to design two different data source objects, each of which are responsible for one type of layout. Your app can then select whichever one it needs at runtime, but now you have the added benefit that those same data sources can be used in different controllers.

To demonstrate data sources in action, we're going to extend the **Reminders** example from earlier to have a **CalendarDataSource** protocol. There are a number of methods you might want from a calendar data source, but in this simple example we're going to add just two: listing events on or near a specific date, and adding an event to a date.

As with delegate protocols, it's important to pass the name of the object as its first parameter so that you follow Apple's naming conventions.

Add this protocol next to **CalendarDelegate**:

```
protocol CalendarDataSource {
    func calendar(_ calendar: Calendar, eventsFor date: Date) ->
[String]
    func calendar(_ calendar: Calendar, add event: String, to
date: Date)
}
```

Now that we have a protocol, we can update **Calendar** to use it. First, give it this property:

```
var dataSource: CalendarDataSource?
```

Even though the data source is optional, it isn't really optional in practice – a calendar without a data source is as much use as a table view without a data source.

Inside the **changeDate()** method we need to retrieve all the events that are stored on or around the selected date, then print them out. It's optional, but that just means we need to implement sensible default behavior if it's missing.

Put this at the end of **changeDate()**:

```
if let items = dataSource?.calendar(self, eventsFor: date) {
    print("Today's events are…")
    items.forEach { print($0) }
} else {
    print("You have no events today.")
}
```

As for adding an event, you might craft some neat animation or you might update your UI in interesting ways, but in this example we're just going to forward the message onto the data source.

Add this method to **Calendar** now:

```
func add(event: String) {
    dataSource?.calendar(self, add: event, to: selectedDate)
}
```

The final step is to make **Reminders** conform to that protocol by adding **CalendarDataSource** to its class definition, then giving it these two new methods:

```
func calendar(_ calendar: Calendar, eventsFor date: Date) ->
[String] {
    return ["Organize sock drawer", "Take over the world"]
}

func calendar(_ calendar: Calendar, add event: String, to date:
Date) {
    print("You're going to \(event) on \(date).")
}
```

Now that it conforms to the protocol, you can add this line to its initializer so that it gets called

by the calendar whenever data is needed:

```
calendar.dataSource = self
```

Separating your delegates

We've now implemented two protocols, one general-purpose **Calendar** class, and one app-specific **Reminders** class that embeds the calendar and acts as both its data source and delegate. This is *extremely* common: you subclass **UIViewController**, make it the data source and delegate of a table view, add thirty or so lines of script, then go and make coffee – it's iOS 101.

While this sort of architecture works well enough for small apps and prototypes, it becomes nightmarish when your code expands. Suddenly you're the delegate for three things and the data source for four, and all the code lives together in one increasingly confused jumble alongside the rest of your business logic.

This ought not to be a surprise to you, and in fact I'd wager that most developers already know this is a problematic way to write code. However, the solution usually seems equally problematic: if we carve out the data source and delegate code from **Reminders** and put them into separate data types, how then can they communicate changes back to the **Reminders** class?

Mostly the answer is "they shouldn't", because a uni-directional data flow is always easier to think about. So, for the data source the code is nice and easy – start creating a dedicated data source class that conforms to the **CalendarDataSource** protocol:

```
class RemindersCalendarDataSource: CalendarDataSource {
    func calendar(_ calendar: Calendar, eventsFor date: Date) ->
[String] {
        return ["Organize sock drawer", "Take over the world"]
    }
```

```
    func calendar(_ calendar: Calendar, add event: String, to
date: Date) {
        print("You're going to \(event) on \(date).")
    }
}
```

You can now modify the initializer in **Reminders** to instantiate that data source and use it for the calendar, like this:

```
calendar.dataSource = RemindersCalendarDataSource()
```

Note: Now that we've moved the data source from **Reminders** you should also remove its **CalendarDataSource** conformance otherwise Xcode will throw up errors.

So far we've had easy wins: we've made our main **Reminders** class simpler and we've made the data source code reusable in other classes if needed too. Even better, we've made the data flow extremely clear: **Calendar** asks **RemindersCalendarDataSource** for its data, and that queries and shapes whatever model data you have underneath.

The problems inevitably lie with delegates, because often they need to take action that involves your main controller code. For example, the user tapping a table view cell row might display a new view controller – if you were to try to split that off into a separate class you might find you end up with a messy network of calls between that class and **Reminders**.

In our example I added this method to simulate such a hurdle:

```
func calendar(_ calendar: Calendar, willDisplay year: Int) {
    title = "Year: \(year)"
}
```

That modifies the **title** property of **Reminders** to reflect whatever year was chosen. If we were to spin that off into its own **RemindersCalendarDelegate** class, we would need to have that class store a reference to its **Reminders** object so that it can communicate changes – it can turn

into a false economy very quickly.

One excellent solution is to use protocol extensions to solve this problem, because it allows you to write shareable code without adding clutter or coupling. However, protocol extensions are invisible to Objective-C code – I've specifically kept **@objc** and its friends away from this chapter, but you might not have such a luxury in your own code.

Another alternative is to use regular extensions. This doesn't give the benefit of reusability and neither does it reduce coupling, but it does at least help you organize your code a little more cleanly.

Just so you can see it in action, let's walk through splitting off the data source into its own data type. Start by just taking all three delegate methods into their own class that conforms to the **CalendarDelegate** protocol, like this:

```
class RemindersCalendarDelegate: CalendarDelegate {
    func calendarShouldChangeYear(_ calendar: Calendar) -> Bool
{
        return true
    }

    func calendar(_ calendar: Calendar, willDisplay year: Int) {
        title = "Year: \(year)"
    }

    func calendar(_ calendar: Calendar, didSelect date: Date) {
        print("You selected \(date)")
    }
}
```

You can then update the **init()** method of **Reminders** so that it creates a new delegate object rather than using **self**:

```
calendar.delegate = RemindersCalendarDelegate()
```

On the surface that looks like a big improvement: we've taken **Reminder** down to only a handful of lines of code:

```
class Reminders {
    var title = "Year: 2018"
    var calendar = Calendar()

    init() {
        calendar.dataSource = RemindersCalendarDataSource()
        calendar.delegate = RemindersCalendarDelegate()
    }
}
```

However, our code doesn't compile, which is a bit of a deal breaker: the **RemindersCalendarDelegate** class tries to adjust a **title** property that belongs to **Reminders**.

This is usually where people turn to hacks, so you might start writing code like this:

```
var parentController: Reminders?

func calendar(_ calendar: Calendar, willDisplay year: Int) {
    parentController?.title = "Year: \(year)"
}
```

Sure it makes the code compile, but that doesn't mean it's time for tea and biscuits. Not only does that introduce tight coupling – we're relying on a specific class now – but it also breaks object-oriented encapsulation because we're now directly adjusting the **title** property of our parent controller.

If you were serious about taking this approach – and there *can* be benefits as long as there isn't a crazy amount of crosstalk – then you should use a protocol instead, like this:

```
protocol ReminderPresenting {
```

```
    func yearChanged(to year: Int)
}
```

Note: you *could* have used **setTitle(str: String)**, but that exposes implementation detail unnecessary – it changes the title now, but it might end up doing more in the future so there's no need to be so specific.

The next step is to make the **Reminders** class conform to **ReminderPresenting**, like this:

```
class Reminders: ReminderPresenting {
```

We can now implement the **yearChanged** method inside **Reminders** so that it updates its **title** property, like this:

```
func yearChanged(to year: Int) {
    title = "Year: \(year)"
}
```

Using a protocol rather than a concrete class allows us to have more flexibility in the future, and it's just a matter of changing the **parentController** definition to this:

```
var parentController: ReminderPresenting?
```

Now we can call **yearChanged()** and have it do the right thing regardless of what data type implements the protocol:

```
func calendar(_ calendar: Calendar, willDisplay year: Int) {
    parentController?.yearChanged(to: year)
}
```

That completes the code. It wasn't clear cut – it involved adding another protocol as well as a custom data type to act as delegate – but it still meant that the **Reminders** class ends up being relatively simple.

However, it was also a fairly trivial example: in your own code it's likely your delegate will need to retrieve information from your data source, which means having even more directions of communication. Moving the data source to its own object was a clear win, but moving the delegate takes more thinking.

Before I finish, there's one last thing to know: I've used **class** here for all my data types even though in many cases **struct** is a more natural choice. This is because protocols behave differently when working with structs and classes, and that particular discussion is best reserved for the Protocols chapter later on in the book. In the meantime, if you want to use structs for **Reminders**, **RemindersCalendarDataSource**, and **RemindersCalendarDelegate** it's easy enough:

- Add the **mutating** keyword before **func yearChanged** in both the protocol and **Reminders**.
- Add **mutating** before the **willDisplay** method in both the protocol and **RemindersCalendarDelegate**.

SOLID code

As you've seen, the **Calendar** class had no knowledge of the **Reminders** class – one is designed to be reusable in a variety of forms, and the other is a specific implementation to solve an app-specific problem. The two of them work together using the **CalendarDelegate** and **CalendarDataSource** protocols, which is what ensures loose coupling: as long that protocol remains fixed, either of the two classes could be changed without breaking the code.

One of the most popular design principles in computer science - Apple development or otherwise – is "SOLID". It stands for five principles:

- The Single responsibility principle, where one class should be responsible for only one thing.
- The Open/closed principle, which states that software should be open for extension but closed for modification.
- The Liskov substitution principle, which allows a subclass to be used where one of its

parent classes would be.
- The Interface segregation principle, which states that having multiple small interfaces is better than having one large one.
- And the Dependency inversion principle, which states that it's better to depend on abstractions than concrete things.

Even though delegation is only one of many design patterns open to Apple developers, it manages to do a good job of making your code SOLID:

- Rather than combine lots of functionality into one class we adopted the single responsibility principle by having dedicated data source and delegate classes.
- Rather than forcing users to subclass **Calendar** to control its behavior, we used delegation to adopt the open/closed principle so that users can extend its behavior without modifying its code.
- Rather than lump calendar data source and delegate into one protocol, we used the interface segregation principle to make two smaller, independent protocols.
- Rather than depend on specific classes, we used the dependency inversion principle to depend on abstractions - our protocols.

So, it's not quite *SOLID* code, but it is certainly *SOID* code!

Next time you find yourself adding **XYZDataSource** or **XYZDelegate** to a view controller, ask yourself: does it really need to be there? If you strive to make your view controllers the data source or delegate for as little as possible, then when you *do* add those protocols it becomes a meaningful exception.

Summary

Apple platforms use delegation in so many places and so many ways that you simply can't avoid it. Fortunately – unlike some other platform patterns – delegation works just as well in Swift as it did in Objective-C, and it's still the best way for two objects to communicate without tight coupling.

Here are my suggested guidelines:

- It isn't *required* that you use protocols with delegation, but it is certainly a good idea to reduce your coupling.
- Always follow Apple's naming conventions for delegates: use a **delegate** property if there is only one, or put "delegate" at the end if there are more than one.
- Apple also has naming conventions for delegate methods, so you should pass in the object that triggered the event as the first parameter.
- Your types should function even without a delegate set, but you're OK to make them throw errors without a data source if you want.
- If your delegate doesn't need to implement a method, either use **@objc** and mark it optional or provide a default implementation. The latter is nearly always preferable.
- Trying to make one type act as data source or delegate to multiple things is problematic in anything larger than a toy app – try to separate them if possible.

Selectors

Selectors give us a generalized way to refer to functions, usually for the purpose of calling them – like function pointers do in some other languages. However, they are more than a little murky in Swift because of its Objective-C history, and when you encounter them it's always a result of finding yourself in the uncanny valley between Swift and Objective-C.

Even though it's not something I relish saying, to be blunt if you want to fully understand selectors in Swift and the drawbacks they have, you need to understand how they worked in Objective-C. As a result, parts of this chapter read a bit like a history lesson, but I'm afraid that's unavoidable – selectors really do feel uncomfortable in Swift thanks to their unique heritage.

First, though, let's look at what problem they solve. Selectors are commonly found in Swift code such as this:

```
navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .add, target: self,
action: #selector(addSong))

let tap = UITapGestureRecognizer(target: self, action:
#selector(userDoubleTapped))

let timer = Timer.scheduledTimer(timeInterval: 10, target:
self, selector: #selector(chooseNewSong), userInfo: nil,
repeats: true)

performSelector(inBackground: #selector(checkWikipedia), with:
nil)

NotificationCenter.default.addObserver(self, selector:
#selector(userLeavingApp),
name: .UIApplicationWillResignActive, object: nil)
```

```
let lookup = UIMenuItem(title: "Applause", action:
#selector(applaudGreatMusic))

undoManager?.registerUndo(withTarget: self, selector:
#selector(undoPlaying), object: nil)
```

In each of those examples the selector refers to a method without actually calling it – "run **checkWikipedia()** in the background," or "run **chooseNewSong()** every 10 seconds." This is what makes them object-oriented function pointers: they let us refer to a function for use at some point in the future.

If you try to imagine how you might implement the code examples above without selectors, you'll realize the usefulness of them: subclassing buttons just to add some functionality would cause all sorts of complexity (never mind coupling!), using delegation would mean one method being responsible for all your button presses, and trying to squeeze in Swift closures will wreak havoc with the Objective-C code.

Selectors are *not* enough to make a full method call, because they are missing two pieces of information: where to call the method, and what parameters to send. In the examples above, the parameters get filled in automatically by the caller (e.g. **Timer** or **NotificationCenter**), but the "where" part is either explicit or implicit.

When it's explicit Apple calls the "where" a *target*, as in these examples:

```
navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .add, target: self,
action: #selector(addSong))

let tap = UITapGestureRecognizer(target: self, action:
#selector(userDoubleTapped))

let timer = Timer.scheduledTimer(timeInterval: 10, target:
```

```
self, selector: #selector(chooseNewSong), userInfo: nil,
repeats: true)
```

In each of those, the target is **self** and the action to perform on that target is a selector. The combination means the final method call becomes something like **self.chooseNewSong()**, and it's so commonly used on Apple platforms that it has a name: the target/action pattern.

Although this pattern is extremely common on Apple platforms, it doesn't fit perfectly with Swift. You see, each of those examples above register a callback with the Objective-C underbelly that sits just below the surface in all our apps: **Timer**, **NotificationCenter**, and **UIBarButtonItem** are all written in Objective-C, and so for them to call Swift methods we must mark those methods with the **@objc** keyword.

The **@objc** keyword is fundamental to what makes the target/action pattern work. Objective-C allowed any message to be sent to any object – you could take the **addObject** method of an array and call it on a string, on a view controller, or a timer, even though none of those things support that method. Every object was free to decide how it should respond to unknown messages – on macOS you would normally get nothing more than an error printed in your Xcode log, but on iOS a complete app crash was more common. Objective-C even allowed you to send messages to **nil** – to empty memory – in which case they would be silently ignored.

Now, you might have noticed that I switched from saying "method" to "message", and this is where some folks get confused. Objective-C didn't make method calls, and instead relied on a principle of message sending. This looks similar at first glance: you say "do thing" and the **doThing** method gets called.

However, it's hugely different behind the scenes, and the explanation starts with this phrase: "every object was free to decide how it should respond to unknown messages." If you send a message to an object that doesn't understand that message, it can actually forward the message on to something else that can understand it – and that could forward it even further.

What this means is that when you send a message to an object, all manner of things could

happen based on runtime behavior, and the final method that gets run might not be the one you were trying to call.

In Swift this sort of behavior is frowned upon, partly because it can be error prone, but also because it's slow – looking up functions at run time allows all sorts of flexibility, but also stops the compiler from performing aggressive optimization. Runtime function look up, known as *late binding*, was the standard in Objective-C, and Swift can do that too but prefers not to for performance reasons.

Now let's get back to that **@objc** attribute: this marks a method, property, or class as being exposed to Objective-C. By default Swift optimizes all its code so that it makes method calls rather than sending messages, which is both faster and more optimizable. But because the target/action pattern requires connecting to Objective-C components, we need to ask Swift to generate an Objective-C thunk method – a method that maps from the Objective-C way of calling methods to the Swift way of calling methods – and that's done using the **@objc** attribute.

Putting all that together, when we use the target/action pattern we are, at least temporarily, reverting back to sending messages rather than calling methods. Swift will try to type check those methods for us where it can – that's precisely what **#selector** does – but it's not always possible, and in fact Swift is perfectly capable of taking part in Objective-C's message forwarding behavior.

Calling selectors

Let's start with the basics of selectors: using them to call methods. All classes that inherit from **NSObject** – which is pretty much everything in iOS, macOS, watchOS, and tvOS – have a variety of methods that begin with **perform**. Some of them are named clumsily as you'll see, but they all work to allow us to run selectors.

And before you ask, yes: the **NSObject** part really is required, which is part of the reason selectors and the target/action pattern isn't terribly Swift. Using any selector – even just using **#selector(someMethod)** – requires you to use a class rather than a struct, and the **perform**

family of methods are only available if you inherit your class from **NSObject**.

At its most basic, calling a selector is simple enough. Try it out by adding this code to a playground:

```
class BookStore: NSObject {
    func open() {
        perform(#selector(turnOnLights))
        perform(#selector(openDoors))
    }

    @objc func turnOnLights() {
        print("The lights are on")
    }

    @objc func openDoors() {
        print("The doors are open")
    }
}
```

To take that class for a test run, add this code below:

```
let store = BookStore()
store.open()
```

That will instantiate a new instance of **BookStore**, then call its **open()** method – which in turn calls **turnOnLights()** and **openDoors()**.

#selector is a Swift compiler directive: it tells Swift that it should check to make sure both our methods exist while compiling our code. If you had written **#selector(turnOnLihgts)** – with a typo on "lights" – your code simply wouldn't build.

However, like I said earlier selectors do not include parameter information, which is problematic because you start to hit places where Objective-C's relaxed "send anything

anywhere" approach does not play well with Swift.

To demonstrate this, let's modify the **turnOnLights()** method so that it accepts a parameter describing what setting the lights should be on:

```
@objc func turnOnLights(intensity: Int) {
    print("The lights are on at intensity \(intensity)")
}
```

As a pure method call in Swift that clearly must be called using an **intensity** parameter, but remember that selectors don't care about parameters. As a result, **perform(#selector(turnOnLights))** is still valid code - it will still build, run, and work, at least for a very loose definition of "work".

Swift expects a parameter to be passed in, and it will get one even though one isn't being sent. However, *what* it gets is pretty much random, because you're effectively reading junk. In the case of an integer as seen above you'll get entirely random numbers delivered to your method, booleans will always get delivered as true, doubles will always get delivered as zero, and if you had tried using a string instead you'd get a crash – hurrah for consistency.

However, the problem gets worse before it gets better. When you want to perform a selector and pass parameters, you use one of two alternatives:

```
perform(_:with:)
perform(_:with:with:)
```

The first of those is for when you want to pass one parameter, and the second is when you want to pass two. If you want to pass three, you can't – no, really, you can't, which is why it's not uncommon to see Objective-C code that packages up variables in a dictionary so they can be sent over a **perform()** call.

Worse, all those parameters are of type **Any!** - i.e., anything at all, or perhaps nothing. The eagle-eyed among you might wonder how **Any** is possible given that Swift structs cannot be represented in Objective-C (and yet are included in **Any**), but remember the method that will

ultimately be called must be marked with **@objc** so if you use a struct in its signature Xcode will refuse to compile.

However, the worst news is still yet to come: because selectors go through the Objective-C bridge, the parameters you pass into **perform()** might not end up being what you think. To try this out, modify your **perform()** call to this:

```
perform(#selector(turnOnLights), with: 10)
```

You might think that would run **turnOnLights(intensity: 10)**, but you'd be wrong. In my test it actually ran **turnOnLights(intensity: -5764607523034234717)**.

What's happening here is that Swift is silently trying to be helpful, but ultimately failing. Objective-C was fairly poor about working with primitive data types such as integers and booleans – the clue is in its name, it likes *objects*. As a result, Objective-C had a number of object wrappers for primitives such as **NSNumber** and its parent class **NSValue**, so if you wanted to call **perform()** using a primitive data type in Objective-C you would first need to wrap your primitive in an object.

Swift knows this, so when it sees **perform()** it automatically wraps our number inside an instance of **NSNumber**. However, that *doesn't* automatically get unwrapped when **turnOnLights()** is called, so Swift gets confused and tries to treat the incoming **NSNumber** as an integer.

So, if you want to use a method from the **perform()** family with parameters, make sure you send something can or already does inherit from **NSObject** otherwise you might hit trouble.

Resolving selectors at runtime

At this point you might be thinking that selectors sound like an utter nightmare and are best avoided at all costs. However, they really are baked deep into Apple platforms – if they were so easily avoided I wouldn't have covered them here!

I've already given several code examples of using **#selector** in regular Swift code:

UIBarButtonItem, **UITapGestureRecognizer**, **UIMenuItem** and more. But even if you choose to use Interface Builder to create your UIs rather than craft them in code, you're *still using selectors*. How else would the OS know how to connect actions to methods in your code? (In case you were curious, the **@IBAction** attribute automatically implies **@objc**.)

If you'd like to see this for yourself, try creating a dummy iOS app then add a button to its storyboard. Use the assistant editor to Ctrl-drag from that button into your view controller class and create an action called **buttonClicked()**. Now right-click Main.storyboard in the project navigator and choose Open As > Source Code, and you'll see the raw XML behind your storyboard. Here's what I got:

```
<connections>
    <action selector="buttonClicked:" destination="BYZ-38-t0r"
eventType="touchUpInside" id="AeH-Hn-uld"/>
</connections>
```

Your **destination** and **id** values will be different from mine, but the **selector** (boom!) and **eventType** won't be. If you look a few lines up you should see a **viewController** tag that has the same ID as your action's destination – that's the "target" part of target/action.

When you first meet them, outlets and actions are a truly bizarre feature of Xcode – you literally draw lines from your user interface into your code, which seems deeply inefficient. However, there is a reason behind this approach: when your storyboard gets loaded the operating system converts method names like "buttonClicked:" into selectors that can run on your class. But as I've said previously, selectors don't contain information on where to run code or what parameters to run, so using them in Interface Builder allows a pretty amazing level of flexibility should you need it.

Storyboards are designed to work at the view controller level on iOS, macOS, and tvOS, but if you use XIBs instead then you can actually attach the same XIB to any number of different view controllers. At runtime the operating system will check that whatever class you have attached supports all the selectors required to make the XIB work, but as long it does everything is OK. This is loose coupling in the extreme: the view has no idea what kind of

controller will be attached.

Creating selectors by hand

You've seen how Interface Builder is able to convert method names like "buttonClicked:" to calls to the **buttonClicked()** method, but there are two things I haven't explained yet. First, why the method is called "buttonClicked:" with that colon on the end, and second how you can do something similar in your own code.

First, the colon. Although selectors don't include parameters, they *do* at least need to be aware of their existence. This is because we're able to have multiple methods with the same name that differ only by the parameters they accept, like this:

```
@objc func buyBook(name: String)
@objc func buyBook(name: String, author: String)
@objc func buyBook(name: String, author: String, price:
Decimal)
```

Because those three all use the same function name, **#selector(buyBook)** is no longer good enough to produce an unambiguous function call. As a result, we need to be more specific:

```
perform(#selector(buyBook(name:)))
```

Objective-C has a convention of combining its first parameter with its method name that's pretty much unique in the programming world. To see this in action, try adding these two methods to the **BookStore** class:

```
func buyBook(name: String) { }
func buyBook(name: NSString) { }
```

They differ only through the type of parameter they accept: one takes a Swift string and one takes an Objective-C string. That code will build just fine, but now try changing them to this:

```
@objc func buyBook(name: String) { }
```

```
@objc func buyBook(name: NSString) { }
```

This time Xcode will issue an error: “Method 'buyBook(name:)' with Objective-C selector 'buyBookWithName:' conflicts with previous declaration with the same Objective-C selector.”

That one error message reveals three things:

1. When we use **@objc** with a selector Swift silently converts its name to match Objective-C conventions: **buyBookWithName** puts its first parameter right into the method name, like Objective-C developers expect.
2. While Swift is able to distinguish two methods based solely on their parameter data types, Objective-C is not.
3. The colon at the end is there once again, because this is a method that accepts one parameter.

If you have more than one parameter then Objective-C requires you to list them similar to Swift, but in the case of just one it gets baked right into the method name – hence the trailing colon.

Armed with this knowledge I want to show you how you can achieve a similar result to Interface Builder – how you can convert plain strings into method calls. It should go without saying that this bypasses a significant amount of Swift’s type safety, so you should tread carefully.

All the work here is done using the function **NSSelectorFromString()**. It’s the same one that Interface Builder uses, meaning that it’s called once for every connection you make in your storyboards. You call this function with the string form of a selector you want to find, and it will send it back to you.

Remember, selectors don’t have any concept of the thing you are calling them *on*, so the same selector could be applied to any number of different objects or classes. If the selector is totally unique – i.e., if it doesn’t exist anywhere because you made a typo – this function will give one back to you anyway. It really is as unsafe as they come!

When you pass in the selector name there are three important things you must keep in mind if you want to preserve what little sanity this approach affords you.

First, those colons *matter*: Interface Builder's XML used "buttonClicked:" because its actions accept one parameter, so if you forget to include the colon you can expect different behavior or perhaps even a crash.

Second, as you saw Swift automatically converts **@objc** methods into methods that use the Objective-C naming convention: **buyBook(name:)** became **buyBookWithName:**. Because these selectors are being issued from the Objective-C subsystem you must use the Objective-C name of your method.

Third, when you use **#selector** to specify a selector Swift can check the selector exists and is marked as **@objc**. However, when you use **NSSelectorFromString()** Swift has no idea what methods will be called, so it's down to you make sure they are marked with **@objc**.

Let's take this for a test run now. Add this new class:

```
class Customer: NSObject {
    var name = "Taylor Swift"
}
```

That inherits from **NSObject** so it can be used with selectors. Next, add this method to the **BookStore** class:

```
@objc func greet(_ customer: Customer) {
    print("Hello, \(customer.name)!")
}
```

We can now greet a customer when the store opens by using **NSSelectorFromString()**. Add this to the **open()** method:

```
let customer = Customer()
let sel = NSSelectorFromString("greet:")
```

```
perform(sel, with: customer)
```

You can build and run that to see “Hello, Taylor Swift!” printed out. Notice the colon at the end of “greet:” signifying that it accepts one parameter.

Now try changing the **greet()** method to this:

```
@objc func greet(customer: Customer) {
    print("Hello, \(customer.name)!")
}
```

All that does is take out an underscore, making **customer** a named parameter. This time your code will crash, because Objective-C can’t see a **greet:** method to run – Swift has now renamed the Objective-C version of the method to match that language’s conventions. As a result, it must be called like this:

```
let sel = NSSelectorFromString("greetWithCustomer:")
```

One last thing before we move on: if you ever find yourself with a selector that you’re not 100% sure about, use the **responds(to:)** method to check that it’s safe to call. You can call this on any object that can receive selectors, passing in the selector you want to query as its only parameter – if it’s safe to call you’ll get back true, otherwise false.

Delaying selectors

We’ve been using the **perform()** method to run selectors immediately, but you can use **perform(_:with:afterDelay:)** to run a method after a delay has passed. For example, the manager of our book store might want to start piping smooth jazz through the speakers 10 seconds after the store opens:

```
perform(#selector(startMusic), with: nil, afterDelay: 10)
```

That’s equivalent to use this method from GCD:

```
DispatchQueue.main.asyncAfter(deadline: .now() + 10, execute:
startMusic)
```

Given the need to mark things as **@objc** you might think the few extra characters necessary to use GCD are worth it. However, the two methods are subtly different, as represented by this common use case:

```
perform(#selector(startMusic), with: nil, afterDelay: 0)
```

That instructs the system to run **startMusic()** after a zero-second delay, which sounds like it's going to be equivalent to a regular **perform()** call without a delay. However, it *isn't* the same: a zero-second delay causes your method to be called at the start of the next application run loop. Run loops are the infinite loops that sit at the heart of all Apple platform apps, and in simplified form look like this:

```
while true {
    while let event = getNextEvent() {
        run(event)
    }
}
```

There are two loops there: an outer one that continues forever so that your app doesn't just die by surprise, and an inner one that reads and processes all available events sequentially. Those events are things like the mouse moving or key presses on macOS, or finger presses and location changes on iOS. One run through the run loop is the process of calling **getNextEvent()** until nothing remains, at which point the whole thing repeats. When that happens, the inner loop finishes and the outer loop goes around again, looking for more events.

This is where **afterDelay: 0** becomes useful: it automatically schedules the method to be called during the next run loop. This means your application will have finished all outstanding events before your method gets called.

afterDelay: 0 is most commonly used alongside user interface manipulation. For example, a

common problem is running code in response to a user action like a button click: if you run it immediately the button will stay pressed while your code completes, whereas if you run it after a zero-second delay the button will be able to redraw itself fully before your code runs.

Using **afterDelay: 0** is *not* the same as writing either of these two:

```
DispatchQueue.main.asyncAfter(deadline: .now(), execute:
startMusic)
DispatchQueue.main.async(execute: startMusic)
```

The difference is in the precise timing of the selector's execution. When you use **DispatchQueue.main** you're posting work to the main dispatch queue as opposed to the main thread, which is a subtly different thing. Apple's documentation for the main queue says, "this queue works with the application's run loop (if one is present) to interleave the execution of queued tasks with the execution of other event sources attached to the run loop" – it's that interleaving that causes the subtle difference between **perform()** and GCD. When you use GCD to create a zero-second delay you are in fact scheduling the work for the current run loop.

Zero-second delays are one of those things that just isn't needed when you're working in theory, but in practice there are a thousand corner cases that will make you glad it's in your toolkit.

Other ways to perform selectors

There are two other useful ways to work with selectors, both of which are common on Apple platforms.

The first is the rather clumsily named **performSelector(inBackground:with:)** method. I say it's clumsily named because the **inBackground** label is rather nonsensically attached to the selector name, but it's a name we're stuck with so there's no point quibbling.

This method automatically creates a new thread and runs your named method there, which makes for trivial multithreading:

```
performSelector(inBackground: #selector(startMeeting), with:
team)
```

Once you're in the background, another useful method is **performSelector(onMainThread:with:waitUntilDone:)**, which pushes work back to the main thread – the one on which the app was originally launched. This is hidden in Swift using the **@UIApplicationMain** and **@NSApplicationMain** attributes, but that's the same as creating main.swift and making it call the **UIApplicationMain()** function. Being able to push work back to the main thread quickly is very helpful, because most user interface work must happen on the main thread.

Both of these methods are general frowned upon; GCD is preferable in almost all cases. If you see background and main thread selector code it's often a sign that someone has ported Objective-C code to Swift or still feels more comfortable relying on Objective-C code – it's the kind of code you're more likely to read than write, all being well.

Forwarding selectors: the composite pattern

Earlier I said that Swift is perfectly capable of taking part in Objective-C's message forwarding behavior, and I want to demonstrate how it's done and why you might want to use it.

Suppose a customer walks into our book store, doesn't see nearly enough books about Swift, and decides they want to complain to someone. Should they go to the person working at the checkout, should they find an employee wandering around the book shelves, should they head straight to the manager's office, or should they perhaps fire off a passive-aggressive tweet?

The Gang of Four book includes the *composite* pattern, where a collection of related objects can be grouped into one object for consistency and simplicity. This is sometimes used to achieve multiple inheritance-like behavior, but more commonly it's just there so that you have a single entry point that can trigger a variety of different behaviors depending on the current state.

To try this out on our **BookStore** class, create this new class below it:

```
class Manager {
    @objc func handleComplaint() {
        print("Here's a refund and a free latte.")
    }
}
```

Now add this property to the **BookStore** class:

```
let manager = Manager()
```

The next step is to tell the system where the book store should forward any messages it can't process. In Swift apps this would nearly always be the result of an Interface Builder connection, because hopefully you don't have much need for **NSSelectorFromString()**. This is done using the **forwardingTarget(for:)** method – add this to the **BookStore** class now:

```
override func forwardingTarget(for aSelector: Selector!) ->
Any? {
    return manager
}
```

Finally, you just need to send a message that the book store can't handle, like putting this inside its **open()** method:

```
perform(NSSelectorFromString("handleComplaint"))
```

That will send the message **handleComplaint** to the book store, which can't handle it and so forwards it on to the manager to deal with. To the end user it looks like the book store handled the message, but internally we can write code to find the best person to handle a complaint and make sure it gets handled correctly – the composite pattern has helped unify multiple objects in one easier object.

Sending actions

Later in the book I introduce you to the responder chain, known in the Gang of Four as the Chain of Responsibility pattern. This lets multiple objects work together to figure out how a problem should be solved, and it's commonly used in Apple development.

Although that's covered later, there is one special case that relates to selectors: sending actions with a **nil** target. If you specify a **nil** target for any action the system will automatically use the responder chain starting with the current first responder. If it can handle the message then it does, otherwise it goes to the next item in the responder chain, then the next, then the next, and so on, all the way up to your app delegate. If no one can handle the message then it silently does nothing.

You can also send actions using the **sendAction()** method that belongs to both **UIApplication** and **NSApplication()**, and again specifying **nil** as the target will ensure it gets delivered to the first responder. The **sendAction()** method has the added benefit that it returns a boolean set to true if the message was handled somewhere or false otherwise, in which case you can implement some fallback behavior.

In macOS this is commonly used to trigger menu item actions: when the user selects "Paste" from the menu bar that action gets sent to the first responder – whatever is the active text field in the active window – to handle.

In iOS this is commonly used to respond to application messages appropriately. Lots of things in iOS get delivered to your app delegate or other singletons, such as files being sent using AirDrop or push messages. This can be confusing for new developers: some files arrived, so how can you notify users cleanly?

One option is to dig through your view hierarchy, which looks a little like this: find the window's root view controller, typecast it to a **UITabBarController**, find its first view controller, typecast that to a **UINavigationController**, find its top view controller, typecast *that* to your custom view controller type, then call a method. Tiresome, right?

An alternative is to use **sendAction()** with a **nil** target. As long as your view controllers are

able to act as first responders, they will be checked to see if they can run your action.

First, add this to any view controllers that should be able to participate in action responses:

```
override var canBecomeFirstResponder: Bool {
    return true
}
```

Now go ahead and call **sendAction()** straight from your app delegate and it will get routed to the top most view controller:

```
UIApplication.shared.sendAction(#selector(filesArrived), to:
nil, from: self, for: nil)
```

You can call that from anywhere because **UIApplication** is a singleton. You'll notice I used **#selector(filesArrived)** rather than **NSSelectorFromString()** because it's always a good idea to have some fallback method in place in case no one else handles the message – adding something like this ought to do it:

```
@objc func filesArrived() {
    // take fallback action
}
```

If the user has anything interactive selected – for example if they are currently editing text in a text field – that text field will be sent your message first, but as it won't be able to handle it the message will go to the view controller instead

Summary

Selectors are unhappy creatures in Swift. They don't feel Swifty, and they come with a vast array of features that are downright unsafe. However, they are a fundamental feature of Apple programming: they are used across the board in Interface Builder as well as in the target/action pattern – I don't think you can write any useful app for Apple platforms without using one of

those two things.

In this chapter I've also tried to explain *why* selectors work the way they do. As ungainly as they are in Swift, I hope you can at least see the power and flexibility they offer in careful hands.

More broadly, selectors enable us to construct what the Gang of Four call the *command* pattern: being able to create objects that represent actions that can be executed later. Thanks to Swift's first-class functions I'm dubious this is a sensible pattern for general use. As pervasive as selectors are, I would recommend Swift developers restrict their use to the target/action pattern unless faced with a particularly troubling challenge.

Here are my suggested guidelines:

- Although selectors are an inevitable feature of any AppKit or UIKit app, that doesn't mean you need to add them in your own types – closures are preferable.
- Sending actions to nil is a smart and simple way to rely on the responder chain to execute an action. This is covered in more detail in the Responder Chain chapter.
- Performing selectors with a zero-second delay is one of those things you'll think you never need, but will almost certainly come in useful thanks to various UI quirks – it's a good skill to have in your toolbox.
- If you can think of a legitimate reason to use **NSSelectorFromString()** in new Swift code, I'd love to hear it.

Notifications

The Gang of Four book details the *observer* pattern: a publish/subscribe system that lets any number of objects be notified when a specific event happens. Publish/subscribe systems are used so often that many people shorten the term just to "pubsub", but on Apple platforms we call them *notifications* and they are used extensively.

Some years ago I was traveling to Zürich with a sales team, preparing to attend a meeting with the Swiss bank UBS. On the plane one of the sales team mentioned to me that UBS loved our product, but what would really seal the deal would be the ability for them to embed it inside their own apps – to use our app as a framework alongside their own code.

Our app was not at all designed to work as a framework, but in theory it wouldn't have been too hard to convert. However, there was a hiccup: Swiss banks take security extremely seriously, which meant that all our code to track analytics and store logs had to go, which would have been a major downgrade of our app for most others.

I already covered the delegation pattern, which allows one object to be notified of changes to another object. The observer pattern is similar to that, except *multiple* objects can be notified when an event happens, and the object issuing the event is effectively broadcasting the event to any interested parties. You can send a message to any number of observers without knowing if they exist or what type they are, and there may even potentially be no observers.

Notifications are a great example of loose coupling: the broadcaster has no idea of what objects are listening, and the receivers have no idea who sent the notification – it's effectively anonymous communication within your app. However, if only *one* thing ought to be receiving notifications, delegation is always a better choice because it simplifies your program flow and works in a type-safe manner.

(In case you were curious: my evening in Zürich was spent in a hotel room rewriting some core elements of the product so that it dispatched messages using notifications that could then be picked up independently by our logging and analytics services and also theirs. UBS licensed the finished product, and ended up being my favorite client.)

Designing notifications

Notifications on Apple platforms are handled using the **NotificationCenter** system, which delivers us a default center that can both send and receive notifications. You've probably already used it for things like **UIKeyboardWillHide** or **UIApplicationWillResignActive**, using code like this:

```
NotificationCenter.default.addObserver(self, selector:
#selector(screenshotTaken),
name: .UIApplicationUserDidTakeScreenshot, object: nil)
```

That will run a **screenshotTaken()** method whenever a screenshot is taken – Snapchat uses this approach to notify users when someone screenshots their snap. The only requirement for your code is that the method in question be marked **@objc**.

When it comes to delivering your own notifications, a little more thought is required: you need to plan how you intend to implement notifications so that you achieve a useful level of granularity. You see, posting notifications isn't free: you create them, send them to the notification center, which then distributes them all observers – and if there *aren't* any observers you're just wasting CPU time.

You should also give some thought to your notification names. They are *stringly typed*, meaning that they use strings as constants, so to avoid typos you should aim to make them reusable. Apple's own notifications use the same string text for their value as for their name, with the addition of "Notification". For example **UIApplicationUserDidTakeScreenshot** maps to "UIApplicationUserDidTakeScreenshotNotification".

One last thing before we get into a code example: you should aim to follow Apple's standard naming conventions, which avoids verb conjugation. This means using "will" and "did" in your notification names – "DidTakeScreenshot" rather than "TookScreenshot" – but also adding two or three letters to the beginning to ensure your notification names are unique. Apple uses "UI" and "NS" a lot, but you'll probably use your initials or a short form of your company name. Remember, a notification name like "LogFileUpdated" could be observed by

your own code, but by accident also by a third-party library you're using.

Let's start by looking at a simple example in a playground: in an app with many screens, we need a way to inform every screen simultaneously when user settings have changed. While you could attempt to use delegation here, it would be messy and realistically you would just end up recreating **NotificationCenter**. Instead, it's better to have each relevant view controller in your app register for settings change notifications, then post a notification when something changes. Apple takes the same approach when things like Dynamic Type change – the **UIContentSizeCategoryDidChange** notification is posted to anyone listening.

First, add this extension to **Notification.Name** to your playground:

```
extension Notification.Name {
    static let HWSSettingsDidChange =
Notification.Name("HWSSettingsDidChangeNotification")
}
```

Second, have each of the relevant controllers add an observer for that notification. If your controllers all inherit from the same class, e.g. **UIViewController** or **NSViewController**, you might find it easier to create an intermediate subclass to do this automatically. If not, you could create an extension on **UIViewController** to house your observer code – whatever works best for you.

Here's a trivial example controller so we have something to try out:

```
class Controller {
    init() {
        NotificationCenter.default.addObserver(self, selector:
#selector(reloadSettings), name: .HWSSettingsDidChange, object:
nil)
    }

    @objc func reloadSettings() {
```

```
        print("Reloading settings!")
    }
}
```

As you can see, that adds an observer for the **HWSSettingsDidChange** notification we made, pointing it at the **reloadSettings()** method.

Tip: You don't need to remove observers you add; the system removes them automatically.

Finally, add some code to instantiate a controller and post the notification:

```
let controller = Controller()
NotificationCenter.default.post(name: .HWSSettingsDidChange,
object: nil)
```

You should see "Reloading settings!" printed out as the controller responds. We'll come back to this playground later, so please keep it around.

Synchronous notifications

All notifications are delivered synchronously which can be problematic. You see, the very nature of notifications is that they are anonymous: we don't know where it's coming from or where it's going to. As a result, it's possible our notification might get delivered to 50 different objects, and if they do non-trivial work then your app is likely to freeze while that work completes.

On the flip side, the fact that notifications are synchronous can be turned into an advantage. For example, we could extend UIKit components so they have both a light and dark theme, then post an app-wide notification to have them change as one.

To try this out, create a Single View App iOS template and open ViewController.swift for editing.

First, we need a custom notification name that we can send and receive, so add this outside the

ViewController class:

```
extension Notification.Name {
    static let HWSThemeDidChange =
Notification.Name("HWSThemeDidChangeNotification")
}
```

Second, we need something we can theme. There are lots of customizable settings in iOS, but for this example we're going to change the background color of a **UIView**. Add this below the previous extension:

```
extension UIView {
    @objc func makeThemeable() {
        NotificationCenter.default.addObserver(self, selector:
#selector(enableDarkTheme), name: .HWSThemeDidChange, object:
nil)
    }

    @objc func enableDarkTheme() {
        backgroundColor = UIColor(white: 0.1, alpha: 1)
    }
}
```

That adds a **makeThemeable()** method to all views, and when that's called the view will be able to change to a dark theme on demand.

Third, we need a simple test harness:

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let testView = UIView(frame: CGRect(x: 0, y: 0, width:
```

```
256, height: 256))
        testView.backgroundColor = UIColor(white: 0.9, alpha: 1)
        testView.center = view.center
        view.addSubview(testView)
        testView.makeThemeable()
    }
}
```

That creates a new themeable view in the center of the main view so we have something visible to work with.

Finally, we need to post the **HWSThemeDidChange** notification at some point. Add this **viewDidAppear()** method to the **ViewController** class:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
        NotificationCenter.default.post(name: .HWSThemeDidChange,
object: nil)
    }
}
```

That waits two seconds before telling all themeable controls to change – if you run it back you should find it works well enough.

Now, the fact that notifications are *synchronous* matters here: when you call **post()** it will not return until all observers have been notified and have completed their work. In this instance it means that **post()** will not return until all themeable views have switched to dark mode.

We can actually use this to our advantage: because **post()** is synchronous we can place it inside an animation block to have all views animate their change to dark mode. They don't know they are being animated, and indeed don't care they are being animated, because our code is all

loosely coupled.

Change the **post()** call to this:

```
UIView.animate(withDuration: 1) {
   NotificationCenter.default.post(name: .HWSThemeDidChange,
object: nil)
}
```

Keep in mind that not all **UIView** properties can be animated, but this approach certainly makes it easy to adjust many unrelated things at once.

Asynchronous notifications

As you've seen, posting notifications synchronously can be both an advantage and a disadvantage. However, there is an alternative because you can have your notifications delivered asynchronously instead. This approach brings with it two useful benefits: you can instruct the system to deliver notifications at three different urgencies, plus you can ask it to coalesce similar messages to avoid repetition.

Let's start with the basics. Asynchronous notification delivery is powered by the **NotificationQueue** class, which is similar to **NotificationCenter**. It delivers notifications as a first in first out queue (FIFO), meaning that notifications that are queued first get delivered before notifications that are queued later – as long as they have the same priorities.

There are three priorities available to us, known as *posting styles*:

- **.whenIdle** waits until the run loop is empty before posting the notification. This could be some time in executing, so you should use this for "nice to have" notifications.
- **.asap** will run during the next run loop, like using **afterDelay: 0** when performing selectors.
- **.now** will post the notification immediately, like a regular **post()** call.

Head back to the playground code we wrote earlier. It ended with this line to trigger a

synchronous notification:

```
NotificationCenter.default.post(name: .HWSSettingsDidChange,
object: nil)
```

Just so you can see how things work, please change it to this:

```
print("Before")
NotificationCenter.default.post(name: .HWSSettingsDidChange,
object: nil)
print("After")
```

That won't change the way the notification is being sent, but it will print "Before" and "After" so you can see "Reloading settings!" is sandwiched in the middle – synchronous, like I said.

Making that notification asynchronous takes three steps:

1. Create a **Notification** object encapsulating the notification you want to send.
2. Enqueue that with **NotificationQueue**, telling it when it should be delivered and how coalescing should work.
3. Tell the playground it should continue running forever, otherwise it will terminate before the asynchronous work has time to happen.

The second of those steps is the only one that's complicated, because as well as a notification it wants three other parameters: the posting style, the coalescing mask, and a list of modes. We'll come to those soon enough, but for now let's just get it working – replace the existing **post()** call with this:

```
let notification = Notification(name: .HWSSettingsDidChange)
NotificationQueue.default.enqueue(notification,
postingStyle: .whenIdle, coalesceMask: .none, forModes: nil)
PlaygroundPage.current.needsIndefiniteExecution = true
```

Using **PlaygroundPage** requires importing the PlaygroundSupport framework like this:

```
import PlaygroundSupport
```

What you'll see now is "Before", then "After", then "Reloading settings!" – our notification is no longer synchronous.

I've already explained what the posting style options mean, and almost everyone ever will want to use **nil** with the **forModes** parameter because it defaults to the standard run loop that makes the most sense.

However, the coalescing part is both new and interesting, and it's where some of the real power of asynchronous notifications come in. *Coalescing* is the ability to combine things together into one, which in this context means that if you queue up Notification A then queue another Notification A before the first one has been posted, the second will be ignored.

Coalescing is what makes posting style so useful. If a user is making changes in your app, you might post a notification using **.whenIdle** to say that their changes should be synced to the cloud during a quiet period. They carry on making changes, so you post more notifications with **.whenIdle** – it doesn't matter if the original notification hasn't been delivered yet, because you can ask the system to coalesce them.

Finally the user does something big, like inviting a friend to collaborate with them – that's something that can't wait for a quiet period, because it would be confusing if the friend couldn't start collaborating immediately. So, you queue your cloud sync notification with the **.now** posting style. That causes your existing notifications to be coalesced (i.e. discarded in place of the new one), and all your user's changes to be pushed to the cloud immediately. This coalescing behavior is why enqueueing with the **.now** posting style is different from and more powerful than a simple **post()** call to **NotificationCenter**.

Warning: It should go without saying that if you're attaching data to your notifications that is unique and important then notification coalescing might not be for you.

Attaching data

All the notifications we've used so far have not included any data in the notification – it's like saying something has changed, but not saying what its new value is. Fortunately, notifications are capable of sending a **UserInfo** dictionary with your notification, providing any custom data you want to include.

UserInfo dictionaries are Apple's implementation of the associative storage pattern, covered not by chance in the very next chapter. However, it's worth examining briefly how they are used in the context of notifications.

There are two ways a method can be called from a notification. The first is the way we've used so far: a method that accepts no parameters. However, the second way is to make it accept precisely one parameter, which is the **Notification** object that triggered the message.

To try it out, change the **reloadSettings()** method to this:

```
@objc func reloadSettings(note: Notification) {
    print("Reloading settings!")
}
```

The **Notification** class includes the name of the notification that was received, as well as an optional **userInfo** property that contains a dictionary of data. For the purpose of testing, add this to **reloadSettings()** to print out that dictionary if it exists:

```
if let userInfo = note.userInfo {
    print(userInfo)
}
```

Now all that remains is to send some information when posting your notification. Previously we used this code to post a simple notification:

```
NotificationCenter.default.post(name: .HWSSettingsDidChange,
object: nil)
```

That will still work, but let's try changing it to include a **UserInfo** dictionary containing some test data:

```
NotificationCenter.default.post(name: .HWSSettingsDidChange,
object: nil, userInfo: ["theme": "dark"])
```

Now you should see the method printing out the data that was sent.

UserInfo dictionaries are hugely used and abused in Apple development as you'll see in the next chapter, but if you want to send notifications there isn't any other choice.

Filtering by sender

When using **post()** so far we've always specified **nil** for the **object** parameter, and I couldn't really finish this chapter without elaborating on what that does.

Earlier I said that "the broadcaster has no idea of what objects are listening, and the receivers have no idea who sent the notification – it's effectively anonymous communication within your app," and that's true. However, this **object** parameter lets you change that: you can opt to listen for notifications that come only with a specific object attached, often the sender.

Right now here's the code we use to make instances of the **Controller** class listen for the settings change notification:

```
NotificationCenter.default.addObserver(self, selector:
#selector(reloadSettings), name: .HWSSettingsDidChange, object:
nil)
```

If we wanted to, we could say "only listen for times when this notification is sent relating to us, the current instance of **Controller**," by writing this:

```
NotificationCenter.default.addObserver(self, selector:
#selector(reloadSettings), name: .HWSSettingsDidChange, object:
self)
```

The meaning of **object** is really whatever you want. Like I said it's often used as the sender of the notification, which makes it mean "only show messages that come from here." You could also use it as a target for times when you want to restrict notifications, which makes it mean "only show messages directed at me," or "only show messages directed at my category."

You could even use it to pass an entirely arbitrary object, which is often a better idea than using the **UserInfo** dictionary. Using this approach, specify **nil** when adding your observers so they don't apply an object filter, then post your notifications using your custom object - you'll be able to read it inside your notification method using **note.object**. This is more type safe than using **UserInfo**, so for our settings changing example you could post the latest user settings data as a struct.

Summary

Notifications offer a lot of power with low coupling, so it's no surprise they feature heavily in the Apple ecosystem. When adding them to your own code, here are my suggested guidelines:

- Plan carefully how granular your notifications should be.
- Name your notifications with a unique prefix so they don't accidentally clash with any libraries you're using.
- Following Apple's own naming conventions, using **will** and **did** appropriately.
- Remember that notifications are synchronous by default, although an asynchronous alternative exists.
- If you end up with lots of notifications, consider adding an object filter. What it means is down to you, but it will help simplify things.
- Delegation is often the better solution if you're notifying only one thing: it is simpler to follow in code, and completely type safe.
- Don't imagine for a moment that it's pleasant writing tests that rely on **NotificationCenter**, because it isn't.

Associative Storage

The associative storage pattern is Apple's way of attaching arbitrary extra data to fixed objects. This often takes the form of a **UserInfo** dictionary: a property on an object called **userInfo** that stores a dictionary of attached information.

Associative storage is also used when the selection of data that might be provided will vary, for example the list of values that get sent to your app inside the **didFinishLaunchingWithOptions** method. Sometimes it will contain hardly anything, but it might contain a 3D Touch quick action, a URL, information about transferred files, a notification message, and more. It would be wasteful to transfer all possible options inside one giant class, so a **UserInfo** dictionary is used instead.

These **UserInfo** dictionaries vary in what they store: **[AnyHashable : Any]** (see **Notification**) and **[String : Any]** (see **NSError**) are common, but you might also find **[String : NSSecureCoding]** for dictionaries that will be saved to disk (see **UIApplicationShortcutItem**). They are frequently optional, which is a smart way of allowing attached data without incurring extra cost for times it isn't used.

Accessing values in dictionaries in this way isn't type safe, so you need to typecast pretty much everything you get out. To make things a little easier, Apple sometimes adds a very thin layer of Swift over the more raw Objective-C underbelly. For example, the dictionary provided with **didFinishLaunchingWithOptions** is a flat **NSDictionary** in Objective-C but a slightly less awful **[UIApplication.LaunchOptionsKey: Any]** in Swift – at least the *keys* are type safe. Similarly, reading file attributes with **FileManager** returns **[FileAttributeKey : Any]** so you can't accidentally use invalid key names.

You've used dictionaries more than enough already, but it's worth adding one code example to demonstrate optional dictionaries combined with Apple's Swift-friendly key names. Here, for example, is how you check for the presence of a 3D Touch quick action in your app's launch options:

```
if let shortcutItem = launchOptions?[.shortcutItem] as?
```

```
UIApplicationShortcutItem {
    if shortcutItem.type == "com.yourcompany.app.action" {
        // run the appropriate action
    }
}
```

Acting as instance variables

Although Swift's extensions let you add methods and computed properties to existing data types, they don't let you add *stored* properties. This stops you from arbitrarily changing the layout of existing data types, but can also lead to situations where you need to create a subclass in order to add new storage.

Associative storage, where available, can serve as an excellent middle ground: you can read and write custom data in the storage as if it were stored properties, without needing to modify the structure of the data type itself.

A good example of this comes from the Kitura server-side Swift framework. They have a **RouterRequest** class that signifies a user requesting a web resource, and that class has a **userInfo** property to store arbitrary data. Rather than have developers dig around in **userInfo** they created an extension to **RouterRequest** that provides a computed property with getters and setters that work with **userInfo** on their behalf.

To demonstrate this functionality, we can write some playground that adds virtual stored properties to an existing data type. Create a new playground and give it this content:

```
struct User {
    var name: String
    var age: Int
    var userInfo: [String: Any]

    init(name: String, age: Int) {
        self.name = name
```

```
        self.age = age
        self.userInfo = [:]
    }
}
```

Notice the **userInfo** dictionary – that's key to this whole operation.

If a type was created by someone else and you wanted to add new stored properties to it, the only option would usually be to subclass it if you could. However, when they add a **userInfo** dictionary another option is possible: we can write an extension that adds a computed property to dip into that dictionary.

Add this extension to your playground:

```
extension User {
    private var favoriteIceCreamKey: String { return
"@FavoriteIceCream@" }

    var favoriteIceCream: String? {
        get {
            return userInfo[favoriteIceCreamKey] as? String
        }

        set {
            userInfo[favoriteIceCreamKey] = newValue
        }
    }
}
```

That defines a custom **favoriteIceCreamKey** property that we can use inside both the setter and getter, giving it a string purposefully designed to be weird so that it won't clash with any other values being placed in the **userInfo** dictionary.

It also defines a **favoriteIceCream** computed property. Internally we can see that this stores its data in the **userInfo** dictionary, but all users externally see is a regular **favoriteIceCream** property. You could even use nil coalescing to provide an empty string in the getting if you wanted to eliminate optionality.

String constants as keys

I said that Apple sometimes adds a very thin layer of Swift over the more raw Objective-C underbelly, which is true, but sadly there are many times when this isn't the case and you get code like this:

```
public let UIImagePickerControllerMediaType: String // an
NSString (UTI, i.e. kUTTypeImage)
public let UIImagePickerControllerOriginalImage: String // a
UIImage
public let UIImagePickerControllerEditedImage: String // a
UIImage
public let UIImagePickerControllerCropRect: String // an
NSValue (CGRect)
public let UIImagePickerControllerMediaURL: String // an NSURL
@available(iOS 11.0, *)
public let UIImagePickerControllerPHAsset: String // a PHAsset
@available(iOS 11.0, *)
public let UIImagePickerControllerImageURL: String // an NSURL
```

That's taken directly from Apple's own code inUIKit, and it's how you read the user info dictionary that gets sent back from a **UIImagePickerController**.

This kind of code pollutes your namespace, makes code harder to read, and even interferes with code completion because the cases are so similar. The need to comment what each key returns is a reminder of why associative storage is so unsafe.

Sooner or later you will undoubtedly have to deal with this kind of associative storage in your

own, but that doesn't mean you *should* – this is an anti-pattern, and best avoided.

Summary

Associative storage is another Apple platform feature that worked fine in the Wild West world of Objective-C, but makes for an uncomfortable fit in the stricter, safer world of Swift. That being said, it's in here with other platform design patterns because it's unavoidable for Apple developers: so many different APIs rely on this pattern that you can't help but come across it eventually, and the only thing you can do is make the best of the situation.

Here are my suggested guidelines:

- If you're exposing an Objective-C **NSDictionary** to Swift add a typedef for your key type so that it's more specific than just **NSString**. For example, Apple uses **typedef NSString *UIApplication.LaunchOptionsKey** for app launch options – it won't affect Objective-C developers but helps in Swift.
- Add a **userInfo** property to data types that need flexible storage. Make it optional if it won't always be used.
- Use extensions to add computed property to data types, allowing access to stored data without forcing users to deal with dictionary access. Make sure your access key is unique.

Archiving

Archiving – known as serialization in other languages – is the process of converting an object to a state that can be stored on disk or otherwise transferred, as well as the process of *unarchiving* data back to objects. Done properly, archiving preserves all relationships between objects, meaning that each object archives its properties, and each of those properties archives their own properties, and so on, creating a tree.

Archiving is used extensively on Apple platforms: each bundle's Info.plist is an archived dictionary, every storyboard or XIB uses archiving, **UserDefaults** uses archiving, and more. Archiving is another platform technology that sits uneasily in Swift, however Swift 4 introduced the **Codable** protocol in an attempt to smooth over the relationship.

There are two approaches to archiving in Swift, and you may find you need both – in fact you may find you need both even in the same project. In order to use archiving effectively, it's important to at least *understand* both even if you don't quite remember all the syntax.

The first approach is using **NSCoding**, which is the Objective-C archiving protocol. It forces you to write boilerplate code that is error-prone and quite tedious, and it forces you to uses classes that inherit from **NSObject**. On the flip side it offers widespread compatibility with Apple's Objective-C API and any of your own code that also uses Objective-C, which is not to be sniffed at.

The second approach is using **Codable**, which is Swift's archiving protocol. This avoids all boilerplate code unless you have specific needs, works with structs as well as classes, and even serializes to JSON. However, it doesn't support most of Apple's Objective-C API so you need to add workarounds. Used to its fullest, **Codable** is almost invisible – the Vapor project uses it to parse input forms and queries, sending data over requests, and return objects as JSON.

Implementing NSCoding

Let's put together an example of **NSCoding** in a playground, so you can see what it takes if you haven't already used it.

First we need to create a class that inherits from **NSObject** so that it's able to work in the Objective-C world. Add this to your playground:

```
class Settings: NSObject {
    var username: String
    var age: Int
    var lastLogin: Date
    var friends: [String]
    var darkMode: Bool

    init(username: String, age: Int, lastLogin: Date, friends:
[String], darkMode: Bool) {
        self.username = username
        self.age = age
        self.lastLogin = lastLogin
        self.friends = friends
        self.darkMode = darkMode
    }
}
```

I've given the class properties with a variety of data types so you can see them all in action.

The first step to supporting **NSCoding** is the easiest: we need to add **NSCoding** as a conformance in the class definition, like this:

```
class Settings: NSObject, NSCoding {
```

The **NSCoding** protocol requires us to add two methods to the class: one to load data from an archive and one to write data to an archive.

So, the second step is to create a new initializer that accepts an **NSCoder** as its only parameter. This object is Apple's way of loading and saving from archives, and it has a variety of methods to work with different kinds of data. For example, loading an integer uses the

decodeInteger(forKey:) method, and loading a boolean uses **decodeBool(forKey:)**.

Where things get less pleasant is decoding any kind of object. In our class we have a string, an array of strings, and a date, all of which are objects in the eyes of **NSCoding**, so all of which must use the **decodeObject(forKey:)** method. This returns **Any?** so you need to force cast it to whatever type you need, like this:

```
self.username = aDecoder.decodeObject(forKey: "username") as!
String
```

That combines both string typing and force casting in one line of code – I *did* warn you that **NSCoding** sits uneasily with Swift.

Add this initializer now:

```
required init?(coder aDecoder: NSCoder) {
    self.username = aDecoder.decodeObject(forKey: "username")
as! String
    self.age = aDecoder.decodeInteger(forKey: "age")
    self.lastLogin = aDecoder.decodeObject(forKey: "lastLogin")
as! Date
    self.friends = aDecoder.decodeObject(forKey: "friends") as!
[String]
    self.darkMode = aDecoder.decodeBool(forKey: "darkMode")
}
```

The final step is to tell **NSCoding** how to write your data, which is pretty much the inverse of the initializer above: you write an **encode()** method then encode each of your properties using a string key.

This time the syntax is better because there's a single **encode(_:forKey:)** method with multiple overrides for each kind of data you can archive. Add this method to the **Settings** class:

```
func encode(with aCoder: NSCoder) {
    aCoder.encode(username, forKey: "username")
    aCoder.encode(age, forKey: "age")
    aCoder.encode(lastLogin, forKey: "lastLogin")
    aCoder.encode(friends, forKey: "friends")
    aCoder.encode(darkMode, forKey: "darkMode")
}
```

As you might expect, the key names in both methods must match up, so if you have to use **NSCoding** in production it's better to use shared constants rather than typing strings.

Now that we have a class that conforms fully to **NSCoding** we can archive it to a **Data** object using **NSKeyedArchiver**, like this:

```
let settings = Settings(username: "tswift", age: 26, lastLogin:
Date(), friends: ["Ed Sheeran"], darkMode: true)
let data = NSKeyedArchiver.archivedData(withRootObject:
settings)
```

Notice the **withRootObject** parameter name – archiving preserves the relationships between objects, which in this case means it will archive our **friends** array and all items inside it. As long as all properties support archiving, and all properties of those properties support archiving, you can go as many levels deep as you need.

That **data** object contains all the information required to reconstruct the **Settings** object: all the property names, as well as their data types. It's stored as binary data for performance, but you can take a sneaky peek if you're curious – try adding this line directly after **let data =**:

```
let str = String(decoding: data, as: UTF8.self)
```

Click the preview button in the playground results pane, and you should see a jumble of binary characters. However in there you'll see some strings like "bplist" (binary property list), "age", "username", "friends", and other properties from our class, **NSDate** and **NSArray** (the

Objective-C versions of our **Date** and **[String]** properties), and **Settings** (our class name) – it's all there.

Converting a **Data** object back to our **Settings** class is done using **NSKeyedUnarchiver**. This will unarchive a root object and all its properties, but you need to typecast it to whatever class you were using. Here's some sample code:

```
if let loadedSettings = NSKeyedUnarchiver.unarchiveObject(with:
data) as? Settings {
    print(loadedSettings.username)
}
```

That's all there is to **NSCoding** with a custom data type.

The power of **NSCoding** – and why it's still used extensively today – is that so many of Apple's own data types natively conform to **NSCoding**. If you design your user interfaces using Interface Builder, what you create gets saved to XML and unarchived at runtime using **NSCoding**. This is possible because all **UIView** and **NSView** subclasses conform to **NSCoding**, so it's just a matter of recreating them direct from their relevant XML.

Many other common data types are also compatible with **NSCoding**: **UIColor** and **NSColor**, **CGRect** and **CGPoint**, and even **Data** can be stored in there as a binary blob. Behind the scenes, the restoration API of **UIViewController** and **NSViewController** both use **NSCoding**, so both of those classes conform to **NSCoding** out of the box.

This means you can save practically anything using **NSCoding** without having to write any extra code yourself. It might be stringly typed and need force casting, but it sure is flexible!

Basic Codable

The **Codable** protocol is Swift's answer to **NSCoding**. It is nowhere near as flexible, but it does have numerous advantages of its own:

1. It can convert structs and enums as well as classes.

2. It can output JSON or XML depending on your needs.
3. You don't need to write custom archiving and unarchiving methods unless you need something specific.
4. If you *do* want to write custom archiving and unarchiving methods they are type safe and don't use strings.

We're going to convert the **Settings** class so that it uses **Codable** so its benefits are clear.

Start by changing it from a class to a struct, then replace **NSObject, NSCoding** with **Codable**. The **Codable** protocol is actually the combination of **Encodable** and **Decodable** protocols – you can conform to only one of them if you prefer.

Finally, delete both initializers and the **encode(with:)** methods – they will automatically be created for us.

The finished struct should look like this:

```
struct Settings: Codable {
    var username: String
    var age: Int
    var lastLogin: Date
    var friends: [String]
    var darkMode: Bool
}
```

Yes, that's as simple as it can possibly get – we just list the properties it has, and let Swift figure out the rest.

When it comes to encoding that struct, you can use **JSONEncoder** to get JSON or **PropertyListEncoder** to get XML that's designed to be read by Objective-C code. In a pure Swift environment, it won't surprise you to learn that JSON is significantly easier to work with, not least because it can be consumed natively by web services.

Replace the existing **let data =** line with this

```
let encoder = JSONEncoder()
let data = try encoder.encode(settings)
```

That creates an encoder and has it convert our **settings** object to JSON. Add this below the **let str =** line so you can see the JSON contents more easily:

```
print(str)
```

This time you'll see regular JSON, more or less like you would get in any web service. Notice, though, that you don't get the same saved type information with the JSON – it's effectively anonymous data. That's not necessarily a bad thing, because it means **Codable** is capable of importing any kind of data as long as it fits your data structures.

Decoding **Codable** archives is done using **JSONDecoder** or **PropertyListDecoder**, like this:

```
let decoder = JSONDecoder()
let loadedSettings = try decoder.decode(Settings.self, from:
data)
print(loadedSettings)
```

Customizing your archived data

The JSON we made contains a small wrinkle: if you look at the **lastLogin** property you'll notice it a has value something like 537803410.59028234 – hardly a meaningful date.

What you're seeing here is Apple's internal way of storing dates: that number represents how many seconds have passed since January 1st 2001. While this number might be of interest to Apple, it's almost certainly not what you want when working with pretty much every other platform ever conceived.

Fortunately, you can customize the way **JSONEncoder()** stores dates by setting its **dateEncodingStrategy** property. **.deferredToDate** is the default value, where "Date" refers to the **Date** type, but the two options most folks are going to want are **.iso8601** for web standard

dates and **.secondsSince1970** to get a Unix timestamp.

Try adding this line directly under **let encoder =**:

```
encoder.dateEncodingStrategy = .iso8601
```

And add this directly under **let decoder =** line:

```
decoder.dateDecodingStrategy = .iso8601
```

Now the JSON will use dates like 2018-12-22T17:30:00Z – a date and time in the UTC (Zulu) timezone. Whenever you change the date encoding strategy you should adjust your *decoding* strategy to match otherwise you'll hit problems.

There are two further date options that I want to cover briefly, if only for the sake of completeness. First, rather than using one of the built-in date styles you can create your own **DateFormatter** instance to format them using any of Apple's built-in formats. For example:

```
let formatter = DateFormatter()
formatter.dateStyle = .full
formatter.timeStyle = .full
encoder.dateEncodingStrategy = .formatted(formatter)
```

That will use the longest form of both date and time, so you'll see output like "Wednesday, June 20th, 2018 at 2:54:59 PM Greenwich Mean Time". For times you need absolute control, you can also provide a custom closure to write the date in any format you like. You'll be given the date to encoder along with an **Encoder** object – a generalized type that might be a **JSONEncoder**, a **PropertyListEncoder**, or something else.

To encode your own dates in a closure you need to ask the encoder for a single value container – an encoding slot that stores precisely one item, which will be your date. You can then call **encode()** on that container to store a string of your choosing.

This closure will get called once for every date that needs to be converted, so avoid doing

expensive work.

Here's an example you can try:

```
encoder.dateEncodingStrategy = .custom { date, encoder in
    var container = encoder.singleValueContainer()
    try container.encode("The date is \(date)")
}
```

Moving on from dates, another useful property to set is **outputFormatting**, which can be set to **.prettyPrinted** to make the output easier to read, **.sortedKeys** to sort each JSON key lexicographically, or both. For example:

```
encoder.outputFormatting = [.prettyPrinted, .sortedKeys]
```

If you look again at the playground preview for **str** you'll see it's split across multiple lines, making it easier to read.

Custom Codable

In our example so far the Swift compiler is doing most of the work, and that works well enough for times when your JSON directly matches your Swift data types.

However, in practice that's not often the case, not least because of naming – Swift properties mostly use camelCaseNaming whereas JSON often uses snake_case_naming, and **Codable** isn't able to match one up to the other.

There is a more advanced forms of **Codable** that lets you write methods similar to **NSCoding** where you choose exactly how everything should be loaded and saved. However, it also has an intermediate step that is often good enough: you can add an enum to your data type that provides the conversion between Swift property names and their JSON names.

The enum needs to be named precisely: it should called **CodingKeys**, have the raw type **String**, and conform to the **CodingKey** protocol. To be clear, the enum itself is called

CodingKeys and the protocol is **CodingKey** – don't worry if you mix them up, because Xcode will correct you.

Inside the enum you must provide cases for all properties that don't have a default value. In its simplest form you would write this:

```
enum CodingKeys: String, CodingKey {
    case username
    case age
    case lastLogin
    case friends
    case darkMode
}
```

Any property where the JSON name doesn't match the Swift property name should be assigned a string value, otherwise the property name is used for both. In the example above we don't specify any string values so it's no different from the default, but both **lastLogin** and **darkMode** should be changed to match JSON naming conventions.

Add this enum to your **Settings** struct now:

```
enum CodingKeys: String, CodingKey {
    case username
    case age
    case lastLogin = "last_login"
    case friends
    case darkMode = "dark_mode"
}
```

You don't need to make any changes to the rest of your code – all we've done is specified a custom set of coding keys rather than have Swift synthesize them for us. All being well the JSON should reload and you'll see both "last_login" and "dark_mode".

That's the intermediate solution, but if you want complete control over **Codable** archiving you

need to implement one or both of **init(from:)** and **encode(to:)** – and if you implement **init(from:)** you'll also need to write your own standard initializer because you lose the synthesized memberwise initializer of structs.

Note: In the Initialization chapter I detail a way to keep both the memberwise initializer alongside your own custom initializers.

Both of these methods are useful for doing custom processing. For example, one property might be calculated from others rather than stored in the archive. To demonstrate this we're going to write a custom **init(from:)** initializer that ensures everyone has a friend called Tom – a throwback to the blissful days before Facebook.

Writing a **Codable** initializer is similar to **NSCoding** except it's type safe and doesn't use strings. Instead, Swift is able to read the **CodingKeys** enum we provided to know exactly what data type each property has, along with its string name.

So, first you tell the decoder that you want to read a container using the coding keys we specified:

```
let container = try decoder.container(keyedBy: CodingKeys.self)
```

Now you can call **decode()** on that, telling it what kind of data to expect and what key it should read:

```
username = try container.decode(String.self, forKey: .username)
```

Notice that these calls are all throwing methods – decoding will fail immediately if any value was not present. If you'd rather read optionals and *not* fail on missing values, use this instead:

```
username = container.decodeIfPresent(String.self,
forKey: .username) ?? "Some default value"
```

As this is a regular Swift initializer you must ensure all properties are set before the method finishes.

Add this initializer to **Settings** now:

```
init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy:
CodingKeys.self)
    username = try container.decode(String.self,
forKey: .username)
    age = try container.decode(Int.self, forKey: .age)
    lastLogin = try container.decode(Date.self,
forKey: .lastLogin)
    friends = try container.decode([String].self,
forKey: .friends)
    darkMode = try container.decode(Bool.self,
forKey: .darkMode)

    if !friends.contains("Tom") {
        friends.append("Tom")
    }
}
```

Now that we have a custom initializer, Swift will no longer provide its default memberwise initializer, so you'll need to add this too:

```
init(username: String, age: Int, lastLogin: Date, friends:
[String], darkMode: Bool) {
    self.username = username
    self.age = age
    self.lastLogin = lastLogin
    self.friends = friends
    self.darkMode = darkMode
}
```

As for custom encoding, just call **encode(_:forKey:)** for each property you want to include – if

there are some you want to skip you can do so, just make sure your initializer is updated to reflect that.

Add this **encode()** method to the **Settings** struct:

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(username, forKey: .username)
    try container.encode(age, forKey: .age)
    try container.encode(lastLogin, forKey: .lastLogin)
    try container.encode(friends, forKey: .friends)
    try container.encode(darkMode, forKey: .darkMode)
}
```

Codable and NSCoding compatibility

Although there is a gap between **NSCoding** and **Codable** you can bridge it with a small amount of work: encode your **NSCoding** object to **Data** first, then encode *that* using **Codable**.

To try this out, we're going to add a **favoriteColor** property to the **Settings** struct. This will store a **UIColor**, which would normally work with **NSCoding**.

This takes a few small changes to set up:

1. Add **var favoriteColor: UIColor** to the list of properties.
2. Add **case favoriteColor = "favorite_color"** to the **CodingKeys** enum.
3. Modify the memberwise initializer to accept and store **favoriteColor** as a **UIColor**.
4. Modify the **settings** test value we create so that it species a color – start with **UIColor.green**.

All that just sets up the **favoriteColor** property. The actual work of storing that needs to come in the **encode(to:)** method. We literally need to combine the approaches from **NSCoding** and **Codable**: use **NSKeyedArchiver** to generate a **Data**, then encode that using

the **.favoriteColor** key.

Add this to the end of **encode(to:)**:

```
let colorData = NSKeyedArchiver.archivedData(withRootObject:
favoriteColor)
try container.encode(colorData, forKey: .favoriteColor)
```

When it comes to decoding the situation is fractionally trickier: all the **Codable** decodes use **try** so they can throw an error if something went wrong, but that isn't possible with **unarchiveObject()**. Fortunately that should only be a problem in a fairly precise scenario – the **Codable** decoder must have found some data, but for some reason it's the wrong *kind* of data, i.e. we encoded an array rather than a color. If this (hopefully unlikely!) situation does occur we're going to provide a default value of **UIColor.black**.

Add this to the end of the **init(from:)** initializer:

```
let colorData = try container.decode(Data.self,
forKey: .favoriteColor)
favoriteColor = NSKeyedUnarchiver.unarchiveObject(with:
colorData) as? UIColor ?? UIColor.black
```

All being well, the playground log should show the **favoriteColor** property being stored and loaded correctly.

Before you move on, take a moment to examine the JSON that includes the color. Even though **UIColor** is a relatively trivial component compared to something like **UITextView**, it still takes quite a few characters to store.

Working with hierarchical data

Everything we've used so far has been flat data: a 1-to-1 mapping between JSON and Swift types. In practice, though, you're likely to find that your JSON is organized differently from your Swift types, so you need to give a little thought as to how you parse them.

First, take a look at this example JSON data:

```
let jsonString = """
{
   "name": "Taylor Swift",
   "address": {
      "street": "555 Taylor Swift Avenue",
      "city": "Nashville",
      "state": "Tennessee"
   }
}
"""

let jsonData = Data(jsonString.utf8)
```

That contains a person's name and address, but the address is actually a dictionary containing child values – there's a hierarchy here.

If you want to *keep* the hierarchy then there's nothing special you need to do: define a second struct containing the inner values, make it **Codable**, then add it as a property to the outer struct.

In this case, we'd define two structs: **User** contains properties for **name** and **address**, with the **address** property being a second struct that contains **street**, **city**, and **state**.

Here's that in Swift:

```
struct User: Codable {
   var name: String
   var address: Address
}

struct Address: Codable {
```

```
    var street: String
    var city: String
    var state: String
}
```

If the **Address** were something that only relates to users, I'd be tempted to make it a nested struct to make the relationship clear:

```
struct User: Codable {
    struct Address: Codable {
        var street: String
        var city: String
        var state: String
    }

    var name: String
    var address: Address
}
```

Regardless of which approach you choose, decoding is the same:

```
let decoder = JSONDecoder()
let user = try decoder.decode(User.self, from: jsonData)
print(user.address.city)
```

That wasn't hard to do, and there are definitely advantages to make your Swift layouts match your JSON layouts.

The alternative is to make your Swift hierarchy different from JSON. This might mean flatter – i.e. removing the **address** struct entirely and just using **user.city** – or maybe you need to restructure things entirely.

We already used **CodingKeys** as a way of controlling the mapping between Swift property

names and their JSON equivalents. When you're working with JSON in a hierarchy, you need to create more of these enums – one for each level you want to work with.

In this case what we want is a **User** struct like this:

```
struct User: Codable {
    var name: String
    var street: String
    var city: String
    var state: String
}
```

To avoid confusion, replace your existing **User** struct with that.

In order to make our flat layout work we need two sets of coding keys: one that governs **name**, and one that governs **street**, **city**, and **state**. However, thanks to the type-safe nature of **Codable** we actually need a *linking* case that connects the first set of coding keys to the second. In this, that's going to be **address**: a case that doesn't map to one of the properties of our type, but does let us access the address details.

Add these two sets of coding keys to the **User** type:

```
enum CodingKeys: String, CodingKey {
    case name
    case address
}

enum AddressCodingKeys: String, CodingKey {
    case street
    case city
    case state
}
```

You can see the **address** case in the original set of coding keys – again, it doesn't map to any

property in our type.

Swift now knows that we have two sets of coding keys for our type, but doesn't understand how they relate. The next step is to write a custom **init(from:)** initializer that converts the hierarchical JSON into a flat **User** type.

The **Codable** code we've used so far has looked like this:

```
let container = try decoder.container(keyedBy: CodingKeys.self)
name = try container.decode(String.self, forKey: .name)
```

That applies the default coding keys to whatever container was passed in, then reads out a value.

This time we need a new method called **nestedContainer(keyedBy:)**, which lets us access one container inside another. This is where the linking case comes in: we put **case address** into the **CodingKeys** enum because that's how we tell Swift how to find the data for **AddressCodingKeys**.

Reaching inside a nested container looks like this:

```
let address = try container.nestedContainer(keyedBy:
AddressCodingKeys.self, forKey: .address)
```

That tells Swift where to find the data (**.address**, from the main **CodingKeys** enum), and also what data it will find in there (**AddressCodingKeys**). Once that's done the nested container acts just like the main container we've been using, so we can go ahead and call **decode()** on it as needed.

Add this initializer to the **User** struct:

```
init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy:
CodingKeys.self)
```

```
    name = try container.decode(String.self, forKey: .name)

    let address = try container.nestedContainer(keyedBy:
AddressCodingKeys.self, forKey: .address)
    street = try address.decode(String.self, forKey: .street)
    city = try address.decode(String.self, forKey: .city)
    state = try address.decode(String.self, forKey: .state)
}
```

All that code is correct, but it won't *compile*. Having the **address** case in **CodingKeys** makes Swift read and write JSON to an **address** property by default, which we don't have. We just wrote a custom initializer so that it knows how to handle the nested data, but Swift's default **encode(to:)** method is still being used, so it expects to find an **address** property.

To solve this problem we need to provide our own implementation of **encode(to:)**, which will effectively be the inverse of the initializer we just wrote: encode the name, create a new nested container keyed by **AddressCodingKeys**, then encode the street, city, and state values there.

Add this method to **User** now:

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(name, forKey: .name)

    var address = container.nestedContainer(keyedBy:
AddressCodingKeys.self, forKey: .address)
    try address.encode(street, forKey: .street)
    try address.encode(city, forKey: .city)
    try address.encode(state, forKey: .state)
}
```

Add a few lines of extra code to the end of your playground to make sure it all works correctly:

```
let encoder = JSONEncoder()
let newUserData = try encoder.encode(user)
let newUserString = String(decoding: newUserData, as:
UTF8.self)
```

That converts the **user** struct back into JSON, and you should see it looks the same as our original string – hierarchy and all.

Summary

Archiving can be done using **NSCoding**, **Codable**, or a blend of the two, but regardless of how you do it this is a pattern that runs deep on Apple platforms. Interface Builder, **UserDefaults**, **UserInfo** dictionaries that are persisted, and more all rely on this technology to save and load data efficiently.

Here are my suggested guidelines:

- If you need to work alongside Objective-C code, either yours or Apple's, you probably need to use **NSCoding**.
- When writing your **NSCoding** key names, use constants rather than freely typed strings to avoid typos.
- If **Codable** is possible in your code, it's almost certainly a better solution – even if you need to wrap some **NSCoding** items.
- Try to rely on the built-in encoding/decoding ability of **Codable** before you write your own. Yes, you can customize loading and saving but it just adds extra maintenance burden.
- Remember to use **CodingKeys** to ensure your archived names match the naming conventions of your target. That might be camelCase, in which case **CodingKeys** probably isn't required, but it might also be snake_case.
- Even if you're just loading and saving data locally, you should probably change your date strategy to ISO-8601 for sanity if nothing else.
- If you provide a custom closure for your date decoding strategy make sure it's as small as possible – it could be called a *lot*.
- While you *can* wrap **NSCoding** objects using **Codable**, keep in mind they take up a lot of

space. If you're transferring data over the network you might want to look into a different solution.

- Regardless of whether you use **NSCoding** or **Codable**, both are implementations of the template method pattern covered later in the book.

Bundles

Bundles are probably the easiest to understand of all patterns in this book, and mostly we use them without even thinking about them. On Apple platforms, a bundle is a collection of related sources such as images, strings, and executable code. There's nothing special there, and indeed it's identical to how Java uses JAR archives. However, what's special is that macOS comes configured to treat certain kinds of bundles as packages, which are directories that appear to users a single file.

The best example of a package is any macOS application. If you look in your Applications folders Finder shows you a series of individual files, such as Calculator, FaceTime, and Maps. But behind the scenes what you're really seeing are directories called Calculator.app, FaceTime.app, and Maps.app – Finder automatically recognizes the ".app" extension in directories and treats the result as a single bundle.

To look inside a package, right-click on it and choose Show Package Contents. You can poke around any Apple platform application this way – all the binary code, libraries, images, strings, and more are right there to browse through.

Storing packages like this not only makes installation simply a matter of dragging an application into the Applications folder, but it also stops users from wreaking havoc by deleting or modifying random application files "by accident."

All this matters to us because our own applications are also formed into bundles. All iOS, tvOS, and watchOS apps are bundles, and on macOS everything except command-line tools are also bundles. Normally we don't think about this, but there are two main places where it matters: working with Info.plist and the **Bundle** class.

Inside Info.plist

Every bundle must have an Info.plist file containing its identification and configuration data – what it is and how it works. The operating system needs to be able to read your Info.plist and get from it all your app's static configuration information data, which is significantly faster

than launching your app to query the same information.

By default the Info.plist for an iOS app contains the name of your main storyboard so your app can be started, which orientations you support, as well as the all-important "LSRequiresIPhoneOS" key that marks your app as being for iOS. macOS stores less in there, but you'll still find your app's version number in there.

Those are just the *default* values, though: there are dozens and potentially hundreds more. Should your app terminate when it's sent to the background? Do you support iTunes file sharing? Do you have custom App Transport Security exceptions? Do you need any background modes? What user privacy requests do you plan to make? All those and many more go into Info.plist so they can be read directly by the system.

It's worth looking at the actual code for an Info.plist file at least once, because it's fairly unusual. To do that, right-click on your Info.plist file and choose Open As > Source Code.

Before your application is built the plist – short for "property list" – is plain XML, albeit curious XML. Here's a sample:

```
<key>CFBundlePackageType</key>
<string>APPL</string>
<key>CFBundleShortVersionString</key>
<string>1.0</string>
<key>LSRequiresIPhoneOS</key>
<true/>
<key>UIRequiredDeviceCapabilities</key>
<array>
   <string>armv7</string>
</array>
```

In regular XML the keys would have distinct names and contain their values, rather than the strange linear layout we have in property lists – it's effectively JSON encoded in XML. Once your application is built, the XML plist gets converted to a binary plist for performance

reasons. You can still open it in a text editor, but there'll be some garbage around the character data.

Plist files are capable of storing any capable of property list data, which in Apple's ecosystem means **NSArray**, **NSData**, **NSDate**, **NSDictionary**, **NSNumber**, and **NSString**. For more complex types we need to use archiving to create an **NSData** so it can be saved. As we're working in Swift, our native types (**Date**, **Int**, **String**, etc) will be bridged to their Objective-C counterparts before being archived.

Working with bundles

Our entry point into bundles is handled by the **Bundle** class, which has a variety of methods for loading and working with bundles. Realistically, though, most people will use the main bundle through the **main** property, which contains all the resources for the app you built.

The **Bundle** class is used in very first Hacking with Swift project so that we can read images that are in the bundle:

```
let path = Bundle.main.resourcePath
```

Although you might organize your Xcode project in any number of ways, by default your resources get lumped into a single directory accessible through the **resourcePath** property. If you build your app you can see this for yourself – open the Products group in Xcode, right click on your app name, choose Show in Finder, then follow the same "Show Package Contents" instructions we used for macOS applications earlier. (Obviously you should try this on a project that actually has some resources to copy!)

Rather than dig around in **resourcePath**, it's usually best to ask the **Bundle** class to provide you the path to a specific resource. You can request this as a string if you want, but using URLs is safer and thus preferred:

```
let musicURL = Bundle.main.url(forResource: "music",
withExtension: "m4a")
```

Finally, the **Bundle** class is useful as part of local receipt verification. When an in-app purchase is made using StoreKit, you can read the **transactionReceipt** property of the **SKTransaction** object to perform immediate validation, but if you want to perform validation later you should find the locally saved receipt at the URL specified in **Bundle.main.appStoreReceiptURL**.

Warning: Apple's official documentation says, "This property makes no guarantee about whether there is a file at the URL—only that if a receipt is present, that is its location."

App thinning

Just briefly, I want to add that apps downloaded by users from the App Store have been *thinned* – any assets that are not compatible with the user's device have been removed. For example, if they have an iPhone X they need only the @3x versions of your images, so the @2x versions will be removed. This speeds downloads and reduces storage needs, all done transparently.

App thinning is supported on iOS, tvOS, and watchOS, but sadly not supported on macOS.

Here are my suggested guidelines:

- It's rarely a good idea to dig around in your bundle directly. Use the methods on **Bundle.main** and your code is guaranteed to work even if Apple changes bundle layouts in the future.
- You can read values directly from your Info.plist using the **Bundle.main.object(forInfoDictionaryKey:)** method. For example, **CFBundleShortVersionString** contains your version number – just typecast it to a string.
- Your Info.plist can store custom values as well as Apple's own. It's just XML, so you could easily auto-generate this to include something like a server identifier.

Chapter 4
Language Patterns

Design patterns that are idiomatic to Swift.

Initialization

Some aspects of Swift are powerful, some are quirky, and some are flexible. Initialization is probably the only part of Swift I would describe as being *picky*: it follows highly specific rules, several of which work entirely unlike all other major programming languages, and there is no scope for bending those rules even a little. To be fair, Objective-C initializers were also weird in their own way, so clearly this is something Apple enjoys treating in a special way.

There's no point me parroting Apple's official documentation back to you, so instead I want to try to distill the rules so they are as concise as possible, then provide some suggestions you can follow to make initialization less cumbersome. I'll also try to add some explanation of *why* things are the way they are – although Swift is remarkably fussy about initialization, there is good reason.

General rules

- All stored properties must have a value before initialization ends.
- Your initializer may not call any methods until all its properties have values.
- Using default values is an easy way to eliminate the need for a custom initializer.
- Optionals also don't need to be given a value during initialization, because they default to nil.
- Properties that are declared as constant can be set once during initialization, then cannot be changed further.
- Assigning properties a default value or given them a value during initialization will not trigger property observers.

Initialization of structs

Swift initialization is generally a complex, rule-based affair, but structs are exception to that – struct initialization is delightfully simple, and in fact even comes with a bonus feature in the form of memberwise initialization.

A memberwise initializer is one that includes parameters for all properties specified in the

struct. As an example, consider this struct:

```
struct Person {
    var name: String
}
```

That will automatically have a **Person(name:)** initializer generated for it. As you add more properties, they will be added sequentially to the parameter list. These memberwise initializers are only available for structs, because classes are significantly more complex thanks to inheritance – an advantage for structs, I think.

These memberwise initializers are only generated for you if you don't define your own initializers. As soon as you add any initializers of your own, even empty ones, the memberwise initializer disappears to avoid problems where other developers use the automatic initializer and accidentally bypass your setup code.

So, this struct will only have one initializer, called **init()**:

```
struct Person {
    var name: String

    init() {
        name = "Taylor"
    }
}
```

Losing the memberwise initializer is annoying because it's just so useful, so Swift provides a workaround that lets you keep the automatic memberwise initializer alongside your custom initializers. The trick is this: declare your own initializations inside an extension. For example:

```
struct Person {
    var name: String
}
```

```
extension Person {
    init() {
        name = "Taylor"
    }
}
```

As long as the original, non-extended struct has no initializers, you'll get the synthesized memberwise initializer alongside anything else you provide.

And that's all there is to know about initializing structs – delightfully simple, like I said!

Initialization of classes

Initializing classes in Swift can sometimes feel like the language is fighting against you, because there are rules, more rules, then - just for fun – some more rules. Most of the complexity relates to classes that inherit from other classes, but that isn't much help given how often we need to subclass things like **UIViewController** or **SKScene**.

Here's a summary of Swift's rules for initializing inherited classes.

1. You must call one of the parent's initializers before your own initializer finishes.
2. You cannot modify any properties from parent classes before you have called a parent initializer.
3. You can add convenience initializers in extensions, but not designated initializers.
4. If your child class has none of its own designated initializers, it inherits all the designated initializers from its parent class.
5. If your child class implements all the designated initializers from its parent class, it automatically inherits all the convenience initializers.
6. If rules 4 and 5 don't apply to you – i.e., if your child class has any of its own initializers but doesn't implement all the designated initializers from its parent class – then you don't inherit any initializers from the parent class.

Rule 6 is the one that catches people out the most, and it usually happens when they create a

custom subclass with some extra properties, then try to write an initializer for those properties. Suddenly they find they must now implement other initializers from the parent class, they probably need the **convenience** and **required** keywords, and they need to understand initializer delegation.

Designated vs convenience initializers

So far the rules aren't too bad: if you don't create any initializers or don't inherit from anything, you don't have anything special to worry about. However, you may not fully understand the distinction between designated and convenience initializers, nor why they might be helpful in your code, so I want to focus on them just briefly.

Here's the summary:

- All classes need at least one designated initializer. You may have more than one, but not less.
- If you inherit from another class and have no initializers of your own, you get the parent's designated initializers.
- Otherwise, if all your properties have default values, the designated initializer is provided for you and is empty.
- If you create your own designated initializers you must must ensure all properties have values by the time initialization has finished, and you must call an initializer on your parent class if there is one.
- Convenience initializers are optional, and are there to help make the class easier to use, for example providing default values.
- All convenience initializers must call another initializer *in the same class* before touching any properties. You can't call initializers from a parent class.
- Convenience initializers can call other convenience initializers, but ultimately a designated initializer must be called.

So, designated initializers must call a designated initializer in their parent class if applicable, and convenience initializers must call some other initializer in the current class. That "some

other" could be a convenience initializer, but that in turn will need to call an initializer until eventually a designated initializer is reached.

Required initializers

Some initializers are marked with the **required** keyword, which means that anyone who subclasses this class must implement this initializer. When you implement a required initializer you must also include the **required** keyword so that any further subclasses have the same restriction placed on them.

The **required** keyword *forces* you to implement a specific initializer, which is important. If you recall, classes that have no initializers of their own automatically inherit initializers from their parent, but as soon as they implement one initializer that stops happening.

Some initializers are really important. For example, **UIView** has the following initializer:

```
required init?(coder aDecoder: NSCoder) { }
```

That initializer is the one that gets used when a view is loaded from Interface Builder, which for many developers is *all* their views. It's marked **required** because Apple want to make sure you implement it, otherwise if you subclass **UIView** you could end up making something incompatible with Interface Builder.

Failable initializers

Sometimes initializers might not work, and Swift gives us two ways of marking that: **init?()** and **init!()**. You'll meet the first of those fairly regularly, but the second almost never – it only really appears in code that was imported from Objective-C that pre-dates nullability, and returns an implicitly unwrapped optional from the initializer.

Most initializers have no return value, but failable initializers are the exception: if for some reason your initialization cannot complete, you can return **nil**. That's the only thing they can return – if everything works you return nothing, but if there's a failure you return **nil**. As a result, failable initializers don't have a return type in their signature, i.e.:

```
init?(name: String, breed: String)
```

Note: Swift doesn't require failable initializers to return **nil** at any point; you might want to use it to mark that the initializer could be updated to return **nil** in the future.

One particularly advanced feature of Swift is the ability to override a failable initializer with a nonfailable initializer. That is, while the parent class's initializer might return nil, you can create create a new subclass with the same initializer that won't return nil.

As an example, consider this **Animal** class:

```
class Animal {
    var type: String?

    init() { }

    init?(type: String) {
        if type.isEmpty { return nil }
        self.type = type
    }
}
```

That allows users to create unknown animals (new discoveries, perhaps?), or specify the type of animal as a string. However, it uses a failable initializer: if the **type** string is empty then you can't create the animal, because an empty string isn't a valid type of animal.

If we created a subclass of **Animal** called **Dog**, we could be more relaxed because we have a sensible default: someone could pass in a specific type such as "Poodle" or "Samoyed", but if they sent an empty string we can just say "Dog". Here's how that would look:

```
class Dog: Animal {
    override init(type: String) {
        super.init()
```

```
        if type.isEmpty {
            self.type = "Dog"
        } else {
            self.type = type
        }
    }
}
```

As you can see, both **Animal** and **Dog** have the same **init(type:)** initializer, but the **Dog** version always succeeds whereas the **Animal** version is failable. Although it's possible to override a failable initializer with a nonfailable initializer, the opposite is not true.

One last small thing before moving on: if you call a failable initializer from a nonfailable override and call the failable initializer from inside your override, you must force unwrap it using **!**. You *cannot* use **if let** to try to check whether it worked. For example:

```
override init(type: String) {
    if type.isEmpty {
        super.init(type: "Dog")!
    } else {
        super.init(type: type)!
    }
}
```

Deinitialization

One major benefit of classes over structs is that they can have deinitializers – code that runs when the object is destroyed. Like initializers these are named specially – in fact more so because they don't have a parameter clause, but they are much simpler to work with because Swift automatically calls them when the last reference to an object is released. When working with class inheritance, Swift calls the full stack of deinitializers for you, starting with the child class then moving on to the parent and grandparents.

Deinitializers are mainly useful for when you're working with external resources that Swift can't memory manage for you. For example, I maintain an open source image framework called SwiftGD (see **https://github.com/twostraws/SwiftGD**) – it's like an extremely slimmed-down version of Core Graphics, but designed for use in server-side Swift apps. This wraps a C library called GD, which manages image memory itself using pointers, so Swift isn't able to free up that RAM automatically. As a result, I use classes so that the deinitializers can free up GD's memory automatically, to avoid memory leaks.

To try this out, here's a simple class hierarchy with two deinitializers:

```
class Animal {
    deinit {
        print("I'm an ex-animal")
    }
}

class Dog: Animal {
    deinit {
        print("I'm an ex-dog")
    }
}
```

If you create an instance of **Dog** then destroy it, you'll see "I'm an ex-dog" then "I'm an ex-animal" printed to Xcode's console.

The easiest way to try it is with code like this:

```
do {
    let animal = Dog()
}
```

That will create a temporary scope in your playground, which will force Swift to destroy the variable before the playground ends.

Dynamic creation

One thing I should mention before wrapping up is the ability to initialize classes dynamically using the **NSClassFromString()** function. I should emphasize that this is deeply un-Swifty, but – as with so many Apple platform patterns – is also almost unavoidable.

You see, dynamic creation is closely tied to the archiving pattern that is everywhere on Apple platforms: when you use Interface Builder to add a class name to a view controller or a table view cell, IB just saves that as a string. At runtime, the platform uses **NSClassFromString()** to figure out what the actual associated class is, at which point it can be instantiated.

Swift automatically namespaces all classes by the module they belong to, which is a fancy way of saying if your project is called MyProject and your class is called MyClass, your internal class name is **MyProject.MyClass**. This is the same reason why Interface Builder asks you which module your class name belongs to, with an "Inherit Module From Target" checkbox to get the default "MyProject." prefix.

If you want to try **NSClassFromString()** yourself, create a new iOS Single View App project called MyProject, then add this class somewhere:

```
@objc class Animal: NSObject {
    override init() {
        print("I'm alive!")
    }
}
```

Creating an instance of that class is done by first converting the string "MyProject.Animal" into a class, then typecasting that as **NSObject.Type** – some subclass of **NSObject**. This allows us to call **init()** on it.

Try putting this code into **viewDidLoad()** in ViewController.swift:

```
let className = "MyProject.Animal"
```

```
if let actualClass = NSClassFromString(className) as?
NSObject.Type {
    let animal = actualClass.init()
} else {
    print("No such class found")
}
```

Before **#available** was introduced in Swift 2.0, **NSClassFromString()** was commonly used for availability checking. You would simply pick any class you wanted to check for, put its name into **NSClassFromString()**, then check for a **nil** return value.

Summary

Swift initialization has many rules, but I've tried to make them as succinct as possible. If you're looking for the absolute least you need to know, here are my suggested guidelines:

- Prefer structs rather than classes so that initialization becomes a non-issue.
- Where possible provide default values or make property optionals to reduce the need for initializers even further.
- Convenience initializers are there to make your life easier – use them!
- Failable initializers express failure really cleanly, and don't cause much of a speed bump thanks to **if let**.
- Don't use dynamic creation unless you really like living dangerously. Let Interface Builder carry on with it, but that doesn't mean it needs to taint your own code.

Extensions

Swift's extensions allow you to add or modify functionality in existing data types, which by itself is a common feature of many languages. However, Swift adds two features that make extensions an indispensable feature of the language:

- You can extend *protocols* as well as concrete types, which means you can add functionality to a whole range of types at once.
- You can add *constraints* to your extensions, ensuring your modifications apply only to a specific subset of targets.

Extensions are somewhat similar to what the Gang of Four call the Decorator pattern. Officially, the decorator pattern should let you dynamically (i.e. at runtime) modify behavior in an existing object, but Swift extensions can't be added dynamically and must always apply to all instances of a data type. Apple describes this rather generously as "fulfilling the intent but taking a different path to implementing that intent."

However, on the flip side we do get protocol extensions and constraints, so on balance I think we should all be rather happy with Swift's implementation.

When an extension adds methods to an existing data type, they are treated no differently from that type's own methods – there's no way of knowing at runtime whether you're using an original method or one provided in an extension. However, extensions are *not* able to add most kinds of stored properties, so you must either use computed properties or simulate stored properties using the associative storage pattern.

A common belief is that Swift extensions are the same as Objective-C categories, but that isn't quite accurate. If you extend a class that has been marked as **@objc** then yes, that is identical to an Objective-C category, however for other data types Swift extensions only apply to code where they are visible. In Objective-C, categories can be used whether or not you import whichever header declares them, whereas Swift opts for a safer approach.

Protocol extensions are covered in detail in their own chapter, but it's important to note here

that another key difference between extensions and categories is that Swift can generate extensions Objective-C cannot use. Specifically, while Swift has the concept of protocol extensions, Objective-C does not and won't be able to use any extensions declared on protocols.

Logical grouping

One of the main uses of extensions is for grouping your code logically based on some criteria you specify. This places groups of related methods into the same extension to help make your code easier to understand and make it more flexible in the future. There are four main groupings:

1. Conformance.
2. Visibility.
3. Ownership.
4. Purpose.

The most common logical grouping is protocol conformance. With this approach you create your data type without any protocol conformances, then add those conformances separately and individually, like this:

```
protocol Compressible { }
protocol Printable { }

struct Image { }

extension Image: Compressible { }
extension Image: Printable { }
```

This is how Swift's own standard library is written, which goes some way to explaining why it contains around a thousand extensions. Remember, Swift relies heavily on structs rather than classes, so without subclassing extensions are the only way of building on existing types and protocols.

Well done well, there are two important reasons why this approach is popular. First, you reduce the amount of knowledge a developer must have to work in a given extension. If they are working inside **extension Image: Printable** they ought to be able to understand just one protocol in order to work efficiently. On the other hand, if that extension also added three other protocols, they might find methods from various protocols are interleaved and confusing.

The second reason for grouping extensions by protocol is that it becomes trivial to remove or replace conformance. If **Image** currently conforms to **Printable** but you want to upgrade it to a larger protocol called **Rasterizable**, you can change that all in one place rather than finding methods scattered around. Similarly, if someone decides that security means **Image** needs to conform to **SecurePrintable** but not regular **Printable** you can take away that conformance in just a few seconds.

A popular alternative way to group extensions is based on *visibility*, which can be helpful for times when developers are likely to read your code to what it does.

For example, an **Image** struct might have a public initializer that uses a **URL** to an image on disk, then have a private initializer that loads using a **Data**, designed for internal use only. If you split that code into extensions based on visibility it would look like this:

```
struct Image {
    init(from url: URL) {

    }
}

extension Image {
    private init(data: Data) {

    }
}
```

The advantage to this approach is clarity for folks reading through your code. If they are

looking at the **Image** struct and want to know how it's used, they won't be faced with some public methods, then some private methods, then some more public methods, and so on – they can look simply at the public extension group and ignore the rest.

A third way to group extensions is based on *ownership*, and is usually deployed when code was generated or otherwise provided by an external party. Core Data can generate model classes for you, but modifying generated code is risky because you'll lose your changes if the code is ever regenerated.

To solve this problem, Xcode uses extensions. If you have a class called **User** Xcode will generate a file called Author+CoreDataClass.swift containing an empty **User** class, along with a file called Author+CoreDataProperties.swift that contains an extension to **User** where its model's stored properties and methods are implemented. Using this approach, Xcode can change only the extension file when it needs to regenerate the code, leaving your changes in the class intact.

Tip: It's commonly thought that extensions cannot add stored properties, but that's not strictly true. Although most kinds of extensions cannot do that, you just met the exception: Xcode utilizes the (uncommon) **@NSManaged** keyword to inject new stored properties in an extension. What this means is that Core Data becomes responsible for providing the actual space for storing these properties when your app runs, as opposed to that storage really belonging to your class. It's only ever used with Core Data.

The final way to organize code into extensions is the easiest: *purpose*. If you're working with large data types it can be mentally challenging to have everything in one huge definition unless you're extremely careful about organization. In fact, SwiftLint will start warning you when any file exceeds 400 lines, and will throw a full error if any file exceeds 1000 lines.

Organizing code by purpose means slicing up one big type into several small extensions, with each extension being responsible for specific types of methods. For example, you might implement core business logic in your main type definition, then adding loading core in one extension and saving code in another extension.

Similar to extending by protocol, extending by purpose helps simplifying the amount of information you need to keep in your head while reading or modifying code. If you're looking at the extension that handles saving data, you should effectively be able to forget about the methods to load data that are stored in a different extension – they don't get mingled together.

This is particularly important if two parts of your system are functionally independent. For example, Swift's **String** (through **NSString**) has a range of built-in drawing functionality that allows you to draw text to a rendering context. Apple owns both the string APIs and the drawing APIs so they could have lumped them together, but splitting the drawing code into an extension is sensible because it groups things by purpose and makes it easier for Apple to segregate iOS and macOS code.

Regardless of which approach you take to extensions, one of the benefits to all of them is the ability to place extensions in individual files. This makes source control collaboration easier because developers are less likely to be working on the same file, but – critically for Swift – it can also speed compilation times because Xcode's incremental compilation will only worry about the small extension you changed rather than re-evaluating the whole file.

One last thing: as mentioned in the Initialization chapter, extensions have a bonus use when used with struct initializers. All structs come with a default memberwise initializer unless you implement your own initializers, but as soon as you implement your own initializer that memberwise initializer goes away. This is for safety: the memberwise initializer will ignore any extra work you do in your own initializer, so it would be dangerous to use. If you want to keep the memberwise initializer while also adding your initializers, put those initializers into extensions and Swift will continue to provide its memberwise initializer.

Replacing functionality

As well as adding new functionality, extensions allow you to *replace* functionality too, although here you ought to tread more carefully because you are likely to break things unless you are very careful.

To replace any method – including ones defined on Apple's own data types, simply overwrite

it inside your extension, like this:

```
extension String {
    func trimmingCharacters(in set: CharacterSet) -> String {
        print("I don't think so!")
        return self
    }
}
```

That stops the **trimmingCharacters(in:)** method from working, while printing out a message each time it's called.

These method overrides cannot be used to enhance existing functionality, because when you use them you lose the ability to access the original method. So, be sure you have implemented all the functionality provided by the original, and have some tests to prove it.

Constraints

One of Swift's true power features is its ability to constrain extensions based on whatever criteria you like. This allows you to restrict which types inherit your changes, which in turn means you can write methods that would otherwise have failed the type check.

For example, consider this array of numbers:

```
let numbers = Array(1...100)
```

If we wanted to add a computed property called **total** that summed those numbers, you might write something like this:

```
extension Array {
    var total: Element {
        return reduce(0, +)
    }
}
```

However, that won't compile: Swift has no idea what the array contains, so it could contain things that don't support the + operator or perhaps even a mix of types.

This is where constraints come in, because we can mark our extension as being available only for arrays that contain numeric items, like this:

```
extension Array where Element: Numeric {
    var total: Element {
        return reduce(0, +)
    }
}
```

Only the extension signature has to change, but that change enables the code to run because + has a clear definition for all types that conform to **Numeric**.

You can add as many constraints as you want to a single extension, with each constraint separated by a comma. You can also add concrete type constraints using ==, like this:

```
extension Array where Element == Int {
```

With that precise constraint, only the **Int** type will be affected – not **Int8**, **UInt64**, and so on.

Extension constraints are useful for resolving conflicts – when two extensions declare a method that would apply to the same type.

Swift rule here is straightforward: the most constrained extension is used. As an example, consider the following extension:

```
extension Collection where Element: Hashable {
    var isUnique: Bool {
        return self.count == Set(self).count
    }
}
```

That adds an **isUnique** computed property to all collections that have **Hashable** elements. This allows us to put the collection into a **Set** (where duplicates are not allowed), and ensure that the original collection and the set are equal in size. If they are, the collection was unique.

Using that code, these two lines will both return false:

```
[1, 2, 3, 4, 5, 1].isUnique
"the rain in Spain".isUnique
```

However, you might decide that the **isUnique** implementation doesn't make sense for strings – it checks that each letter is used only once, but what you really care about is each *word* being used only once.

Fortunately, Swift lets us write an extension on **String** that implements the same computed property:

```
extension String {
    var isUnique: Bool {
        let words = self.components(separatedBy: " ")
        return words.count == Set(words).count
    }
}
```

Because **String** is more constrained than **Collection** – i.e., because a concrete data type is more specific than a protocol – Swift will always choose the **String** version. As a result, **"the rain in Spain".isUnique** will now evaluate as true.

If two extensions are *equally* constrained, then one of two things will happen. If both extensions are in your code, Swift will throw an error – you need to remove one, or make one be more constrained than the other. If one extension comes from a module, i.e. a separate build target in Xcode, then your own extension will override the module's.

Summary

Although they can't add true stored properties, extensions are still a powerful way to extend existing types or even replace whole methods – when used judiciously. They are used commonly in Swift codebases of all size, so it's worth taking the time to master them.

To get the most out of extensions, here are my suggested guidelines:

- Use extensions to group functionality logically. Organizing things by protocol is a good start, but you might find purpose easier to begin with.
- Don't replace methods unless you're absolutely sure you know what you're doing. This goes doubly for anything in Apple's user interface libraries, AppKit and UIKit, where there are so many corner cases it's easy to miss something.
- Rather than polluting your namespace with functionality, be sure to lean heavily on Swift's expressive constraints system to limit where your extensions are applied.
- Refer to the associative storage pattern for examples of how extensions can simulate stored properties.

Protocols

Protocols declare an interface that any data type can conform to. When a type conforms to a protocol, Swift ensures all the requirements of the protocol are met: that might involve some required methods, some optional methods, or some instance variables.

Protocols have been an essential part of Apple development since the days of NeXT, although their use has changed dramatically since then. In the earlier days of Objective-C, so-called "informal protocols" were written as categories on **NSObject**, effectively saying that every object implemented every method that was provided. At runtime, the object would be checked to see if it *actually* understood the method, and if so would be called.

Nowadays, Objective-C protocols are mostly *formal* protocols: a named collection of methods and instance variables that concrete types can conform to. Swift uses protocols extensively too – arguably even more so than Objective-C – but still the influence of its parent language looms large, as you'll see.

Naming protocols

There are two aspects to naming protocols: naming the protocols themselves (e.g. **Equatable** and **Hashable**) and naming the methods inside those protocols. The conventions aren't complex, but they should be adhered to closely if you want your code to fit smoothly into the rest of the Apple platforms.

First, naming your protocols:

- Use nouns if your protocol describes a thing. Examples: **Collection**, **Sequence**, **UITableViewDataSource**.
- Use adjectives if your protocol describes an ability. Examples: **Equatable**, **Comparable**, **UITableViewDataSourcePrefetching**.

Apple's Swift guidelines specifically suggest the suffixes "able", "ible", and "ing" for ability protocols, but sometimes it takes some linguistic gymnastics to meet that.

As for naming the methods inside your protocols, you can use whatever names you want. However, if your protocol is designed to describe delegates – i.e. you're writing something like **UITableViewDelegate** – you should try to follow Apple's naming conventions so that it fits alongside other code.

Specifically, that means using **will**, **did**, and **should** to make it clear when method are triggered. However, it also means following the Objective-C style of passing parameters.

In the Delegation chapter I gave the following example:

```
protocol CalendarDelegate {
    func willDisplay(year: Int)
    func didSelect(date: Date)
    func shouldChangeYear() -> Bool
}
```

Because it's important that delegates be notified *which* object triggered the method we renamed the first two methods to this:

```
func calendar(_ calendar: Calendar, willDisplay year: Int)
func calendar(_ calendar: Calendar, didSelect date: Date)
```

But what about **shouldChangeYear()**? If you follow the convention above you'd end up with something like this:

```
func calendar(_ calendar: Calendar, shouldChangeYear) -> Bool
```

However that isn't valid Swift – **shouldChangeYear** is just kind of floating there. It's not a parameter because it doesn't have a data type attached to it, so this doesn't have any meaning.

This is the part that usually confuses folks when they learn Apple development: there are two conventions for naming protocol methods. If your delegate accepts only one parameter it should be structured like this:

```
func textFieldShouldClear(UITextField)
```

And if it accepts more than one it should be structured like this:

```
func textField(UITextField, shouldChangeCharactersIn: NSRange,
replacementString: String)
```

So, this would be correct form of **func shouldChangeYear() -> Bool** so that it notifies the delegate of which calendar triggered the change:

```
func calendarShouldChangeYear(_ calendar: Calendar) -> Bool
```

If your protocol isn't designed to act as a delegate or data source, you have free rein to name them as you wish.

Pure Swift protocols

It's outside the remit of this book to discuss Objective-C protocols as written in Objective-C, but one thing you will encounter extensively are Swift protocols marked as **@objc**. These are subtly different from pure Swift (non-**@objc**) protocols, and it's worth being clear on what those differences are.

For example, consider the following simple protocol:

```
protocol MoviePlayerDelegate {
    func movieDidLoad()
    func movieShouldPause() -> Bool
    func movieWillEnd()
}
```

That could be used in a movie player app so that any view controller could embed movies and be notified when something interesting happened: the movie file loaded, the movie file is about to end, and whether the movie file should pause as a result of a given action.

It's possible that a view controller might not want to implement all three protocol methods, however in Swift they have no choice: if you adopt a protocol you must implement all its requirements because optional methods are not allowed. If you want some methods to be optional, you can split them in two like this:

```
protocol MoviePlayerStatusDelegate {
    func movieDidLoad()
}

protocol MoviePlayerPlaybackDelegate {
    func movieShouldPause() -> Bool
    func movieWillEnd()
}
```

With that approach you can conform to whichever of the two (or both) that you need.

Objective-C protocols

When you use the **@objc** attribute on a Swift protocol three important changes happen: you can mark methods as being optional, you may no longer use Swift structs and enums with that protocol, and you may no longer use protocol extensions.

Optional requirements (an oxymoron, but that's the name Swift uses) are optional in the sense that they are not *required*, as opposed to Swift's optional type. So, a type conforming to such a protocol can implement the method but isn't required to.

However, optional requirements are also optional in the Swift sense too: the method might not exist, so you must check and unwrap it appropriately or use optional chaining. The syntax for this isn't particularly natural in Swift because we wouldn't normally consider thinking of looking for a method that might not be present, but it's worth looking at because there are so many **@objc** protocols on Apple platforms.

Here's an example **@objc** protocol you can put into a playground:

```
@objc protocol MoviePlayerDelegate {
    @objc optional func movieShouldPause() -> Bool
    func movieWillEnd()
}
```

That defines one optional method, **movieShouldPause()**, and one required method, **movieWillEnd()**.

We're going to use that protocol for a delegate in a new struct called **MoviePlayer** – it can be a struct here because its *delegate* will conform to **MoviePlayerDelegate** as opposed to the struct itself. This struct will have one method, which will get called when the user pressed pause.

Add this below your protocol:

```
struct MoviePlayer {
    var delegate: MoviePlayerDelegate?

    func pausePressed() -> Bool {
    }
}
```

As you can see, the **delegate** property is optional because there might not be a delegate assigned. And in the **MoviePlayerDelegate** protocol we made **movieShouldPause()** an optional requirement, so even if a delegate is present it might not implement that method.

As a result, to return a value from **pausePressed()** we need to use optional chaining: if there's a delegate and if it implements **movieShouldPause()** then run it and send back whatever it returned, otherwise assume a default value of true.

Put this line in **pausePressed()**:

```
return delegate?.movieShouldPause?() ?? true
```

Notice the double optional chain there, and in particular the syntactic curiosity of the second – we're not checking the *result* of the method, we're checking if it even exists.

The other major "features" of **@objc** is that it cannot be used with structs or enums, and you cannot create an extension for that protocol. The reason for these restrictions is simply because they aren't available in Objective-C, so there's no sensible way to combine the two.

Class-only protocols

Sometimes protocols should be adopted only by classes because you need to rely on reference semantics. This is commonly happens because you have a **delegate** property that needs to use **weak** storage to avoid retain cycles.

For example, if we had a shopping list app, the user might browse through a list of items that are available, then tap one to show a detail screen. On that detail screen they add a couple of their item to their basket, then head back to the list again.

To avoid tight coupling, this relationship is usually represented with a delegate something like this:

```
protocol ListDelegate {
    func item(_ item: String, didUpdate quantity: Int)
}
```

We could then create the item detail controller using that delegate, so it can post updates back to the main list controller:

```
struct ItemDetailController {
    var delegate: ListDelegate?
}
```

However, this opens the risk of a retain cycle: if the list controller stores a strong reference to the detail controller, and the detail controller stores a strong reference to the list controller, memory will be leaked.

The solution is usually to mark the **delegate** property as being **weak**, like this:

```
weak var delegate: ListDelegate?
```

However, that creates a new problem: the **weak** keyword cannot be used with structs and enums, because they are value types – they always have precisely one owner, so **weak** has no meaning.

Class-only protocols are designed to solve this problem, and are easy to implement – just add the **AnyObject** conformance to your protocols, like this:

```
protocol ListDelegate: AnyObject {
```

Now that **ListDelegate** conforms to the **AnyObject** protocol, it can't be used with structs, and so **weak var delegate: ListDelegate?** is allowed.

Protocol inheritance

Protocols are *composable*, meaning that you can combine them together to make larger protocols. Sometimes this is as simple as using **&** for protocol composition – the **Codable** protocol is actually defined as this:

```
typealias Codable = Decodable & Encodable
```

You can also use protocol inheritance to build larger protocols, like this:

```
protocol Payable { }
protocol NeedsTraining { }
protocol HasRestTime { }
protocol Employee: Payable, NeedsTraining, HasRestTime { }
```

The advantage here is that you can add requirements to the **Employee** protocol above and beyond those it inherited.

Associated types

Protocol associated types are one of the most advanced and complex features of Swift, and when you hit problems with them it's usually guaranteed to be a bad night.

A standard Swift protocol lists methods and instance variables that a type must include in order to conform. For example, you might require a type to have three specific methods and two instances variables, both of which must be read/write.

A protocol with an associated type is an *incomplete* protocol: a protocol with a hole in. That hole must be filled by whichever type conforms to it.

I'm going to use a very simple example to start with, because associated types really are rather tricky. Create a new playground and give it this content:

```
protocol Identifiable {
    var id: Int { get set }
}
```

That creates a new protocol called **Identifiable**, and asks that all conforming types provide a single **id** integer property.

We could make two types that conform to that protocol like this:

```
struct Person: Identifiable {
    var id: Int
}

struct WebPage: Identifiable {
    var id: Int
}
```

Both **Person** and **WebPage** implement the **id** integer, so they are fully conforming.

However, in practice people and web pages aren't identified by integers. For the **Person** struct we really want to identify someone using their social security number, which is a string in the format 000-00-0000. As for **WebPage**, each web page can be uniquely identified by their URL, so we want to use a **URL** for the ID there.

Modify your types to this:

```
struct Person: Identifiable {
    var id: String
}

struct WebPage: Identifiable {
    var id: URL
}
```

However, Xcode will now throw an error, because those two types no longer conform to the **Identifiable** protocol – they don't have **id** integers as they are supposed to.

This is where associated types come in: we can modify the **Identifiable** protocol to say there's a hole in our protocol that must be filled by whoever conforms to the protocol. In this case, we want the hole to be the type that should be used for **id**, because **Person** will fill the hole with **String** and **WebPage** will fill it with **URL**.

Add this to the **Identifiable** protocol now:

```
associatedtype IDType
```

That's effectively making the hole – it's saying that every conforming type must define what **IDType** means.

Filling the hole is done inside each conforming type, using the **typealias** keyword, like this:

```
typealias IDType = String
```

However, Swift can be smart: rather than explicitly specifying what the **IDType** hole should be filled with, it can figure it out based on our types. We've already said that **Person** uses a **String** for its **id** property and **WebPage** uses a **URL**, so we can modify the **Identifiable** protocol to say that the **id** property is actually of type **IDType** and Swift will do the rest.

Change the **Identifiable** protocol to this:

```
protocol Identifiable {
    associatedtype IDType
    var id: IDType { get set }
}
```

Now that **id** has the type of **IDType**, Swift can look at **Person** and see that it should be a string, then look at **WebPage** and see that it should be an integer and infer the **typealias** line for us.

So far this probably seems both clever and straightforward, but the hard part is about to come.

Like most object-oriented languages, Swift supports *polymorphism* – the ability for an object to appear in different forms. For example, if we had an **Animal** class that had two child classes called **Cat** and **Dog**, we could create an array of **Animal** and put cats and dogs freely into there because they can both be used like animals as well as their individual types. Swift supports the same thing with protocols: if **Animal** were a protocol and **Cat** and **Dog** were two structs that conformed to it, you could put them both into an array of **Animal**.

However, associated types confuse things: they *cannot* be used generically, because Swift doesn't understand what that means. Remember, a protocol with an associated type is an *incomplete* protocol – it has a hole in that must be filled by whatever conforms to it. So if you have an array of incomplete protocols, they could potentially all be completed in entirely different ways.

In our example we have **Person** and **WebPage**, and they complete the hole in **Identifiable** in different ways: **Person** uses a string and **WebPage** uses a **URL**. If we tried to use them both generically as **Identifiable**, we could potentially try to compare their **id** values even though

they are entirely different types.

As a result, code like this is legal:

```
let taylor = Person(id: "555-55-5555")
```

Whereas code like this is not:

```
let taylor: Identifiable = Person(id: "555-55-5555")
```

If you try that second line you'll see an error message that gives some developers sleepless nights: "error: protocol 'Identifiable' can only be used as a generic constraint because it has Self or associated type requirements".

At the risk of repeating the obvious, a generic constraint is a constraint on a generic type. For example, if you wrote a **checkIdentification()** method that could be called on any kind of object you might give it this signature:

```
func checkIdentification<T>(object: T)
```

A generic *constraint* allows you to limit which types can be used as **T**. In this case, you're likely to want to say "only allow **Identifiable** types", like this:

```
func checkIdentification<T: Identifiable>(object: T)
```

An associated type requirement is any protocol that has an associated type, like ours does here. The **Self** part of the error message refers to using **Self** – a special associated type that refers to whatever current type is implementing the protocol – inside the protocol for any stored properties or method parameters.

Note: Self with a capital S is used inside protocols to mean "whichever concrete type implemented me," whereas **self** with a lowercase S is used inside types and extensions to mean "my current instance."

You can see both **Self** and **self** in this example code:

```
extension Numeric {
    func squared() -> Self {
        return self * self
    }
}
```

That adds a **squared()** method to all numbers: it returns **Self** (which will be **Int**, **Float**, **Double**, etc, depending how it's used), and inside the method it runs **self * self** to multiply the current instance value (e.g. 5) by itself.

The main reason folks run into trouble with associated types is when they try to use protocols with **Self** requirements. Remember, that's any protocol that requires an instance type of **Self** or a method that accepts **Self** as a parameter.

For example, the **Equatable** protocol requires the following method:

```
static func == (lhs: Self, rhs: Self) -> Bool
```

You can see **Self** right there twice, which means if you make your protocol inherit from **Equatable** you'll immediately lose the ability to use it polymorphically.

If you want **Equatable** – and you should, ideally along with **Comparable** – but don't want to fight with the **Self** requirements, the easiest thing to do is conform to those protocols inside your concrete types rather than in your protocol.

Summary

Protocols are an integral part of any Apple platform development, Swift or otherwise, but there are key differences and similarities between Objective-C and Swift protocol that are important to keep in mind.

Here are my suggested guidelines:

- Name thing protocols as nouns, and ability protocols as adjectives: “able”, “ible”, and “ing”.
- Use **will**, **did**, and **should** for protocols that will be used in delegates or data sources. Remember to pass in the object that triggered the event as the first parameter.
- Pure Swift protocols can use all the power and expressivity of Swift, but cannot have optional requirements.
- **@objc** protocols *can* have optional requirements, but cannot work with structs or enums, and can’t use protocol extensions.
- Use class-only protocols when you want your conforming types to act as reference types, allowing you to use them as **weak**.
- Protocols are composable, which means there’s no point trying to build one super-sized protocol with everything in. Instead, start small and compose them to make bigger things.
- Associated types are a powerful feature when you want them, but they can be annoyingly easy to stumble into by accident. If you take a few simple steps you can avoid them entirely.

Protocol Extensions

It won't surprise you to learn that protocol extensions combine extensions and protocols: you can add new code to protocols. They are arguably the most powerful feature in Swift, and certainly the most hyped thanks to the protocol-oriented programming paradigm (POP), but there are precious few examples of it being used well in production code.

The problem with protocols is that they define requirements rather than implementation – you can say a type must implement certain methods and properties in order to conform to the protocol, but you can't provide implementations of those methods. The problem with *extensions* is that they extend only concrete types, such as **Int** and **Array**, so you need to write an extension for **Int**, **UInt**, **Int8**, **UInt64**, and so on if you want all bases covered.

Protocol extensions combine the best of both of them together in one, allowing you to provide default implementations for methods. You can override those methods implementations in a specific type if you want, but it's not required.

Protocol extensions are already covered extensively in my book Pro Swift, and I don't want to repeat myself here. Instead, I want to focus on protocol extensions as a design pattern: how they are useful as a code architecture technique, and how you can use them to architect applications that would be clumsy if not impossible using class inheritance.

Architecting horizontally

Class inheritance provides us with the ability to build one class on top of another, building functionality and adding properties as you go. This has the advantage of working in a way that maps well to real life – a poodle is a kind of dog, a dog is a kind of animal – but in practice you'll find it leads to code that is unwieldy.

That might sound like heresy to some people, because object-oriented programming has been a foundational programming technique for many years now. However, even long-standing OOP developers are clear on this: composition is always better than inheritance, because it aids flexibility.

Class inheritance is the tightest form of coupling we can make, which is why it's also the least desirable way to architect your software. If you have class D than inherits from class C that inherits from class B that inherits from class A, the tiniest change in class A could break something major. And if you decide you don't want class A at all, you might find it breaks *everything*.

This is a result of vertical architecture: one thing inherits from another, bringing with it all its properties and methods even if they aren't needed, then adds its own on top. If you've ever used **UIStackView** you'll know it has a **backgroundColor** property even though it does nothing – stack views never draw to the screen, so any value you assign there is ignored. However, **UIStackView** inherits from **UIView**, so it gets that property (and many others) whether it needs it or not.

Protocol extensions allow you to architect horizontally, which is what makes protocol-oriented programming such an important design pattern in Swift. Horizontal architecture means all protocols are treated equally and independently, and you select only the protocols you want each type to adopt rather than having methods and properties implicitly thrust onto you.

This is a complete inversion from class inheritance, because now child data types are able to pick and choose what they need rather than a parent class deciding for the child. Even better, your protocols can be split up into small parts that solve individual tasks, isolating them from each other so they are easier to reason about and making them more interchangeable.

So, if you decide that you need to add some more functionality to a data type, just add another protocol conformance. And if you decide to remove or replace that conformance later, it won't have any effect on the other conformances.

Protocol-oriented programming is the practice of basing your architecture around protocol extensions. As Apple said when announcing POP, "start with a protocol" as opposed to an abstract class. You can then add more protocols as your code expands, composing them together as needed like Lego bricks.

While protocol extensions are able to provide default method implementations, they may not

provide instance variable storage. This means each type must provide their own instance variables for whatever protocols they conform to.

As well as increasing decoupling, another major advantage of protocol extensions as compared to class inheritance is their ability to work on structs and enums. While using structs isn't always possible in the OOP worlds of UIKit, AppKit, and WatchKit, if you can use them it does remove reference semantics entirely and in doing so reduce the chance of cross-communication.

POP in action

Let's walk through a playground-friendly example of structuring an application using protocol-oriented programming. We're going to model a library where users can find books, audio books, and magazines that interest them, and take various actions on them.

Following Apple's advice, we're going to start with just some protocols: one for things that can be purchased, one for things that can be printed, one for things that can be borrowed, and one for things that can be listened to:

```
protocol Purchasable {
    var price: Decimal { get set }
    func purchase()
}

protocol Printable {
    var printFile: URL { get set }
    func printOut()
}

protocol Borrowable {
    var borrowers: [String] { get set }
    mutating func lend(to: String)
}
```

```
protocol Listenable {
    var audioFile: URL { get set }
    func playPreview()
}
```

Those four are individual, isolated protocols. We can now construct a conforming type like this:

```
struct Book: Purchasable, Printable, Borrowable {
    var name: String
    var price: Decimal
    var printFile: URL
    var borrowers: [String]

    func purchase() { }
    func printOut() { }
    mutating func lend(to: String) { }
}
```

At this point, POP doesn't have much of an advantage over inheritance other than allowing us to use a struct. We could have rolled **Purchasable**, **Printable**, and **Borrowable** into a single **Product** class, and inherited **Book** from there.

However, consider how we would implement an **AudioBook** struct. This should also be **Purchasable** and **Borrowable**, but obviously it can't be printed. Instead, it must have **Listenable**. Then we would implement **Magazine**, which must have **Purchaseable** and **Printable** but not **Borrowable** or **Listenable**.

You *could* try to solve this with a class hierarchy: put **Product** at the top, then create **PurchaseableProduct** as a child class, then create **PurchaseableBorrowableProduct** from *that*, and finally create **Book** at the bottom of the hierarchy. But then what if you have some out of print books that are borrowable but not purchaseable?

Class hierarchies here are a recipe for spaghetti code: you'll end up with a nest of inheritance as you try to create all the various classes and subclasses you'll need.

But even with regular protocols the situation here isn't great. To implement **AudioBook** we need to write code like this:

```
struct AudioBook: Purchasable, Listenable, Borrowable {
    var name: String
    var price: Decimal
    var audioFile: URL
    var borrowers: [String]

    func purchase() { }
    func playPreview() { }
    mutating func lend(to: String) { }
}
```

Add that to your playground now.

In that struct, the **playPreview()** part is new, but **purchase()** and **lend(to:)** are the same methods we had to implement inside **Book** – repeating them here just duplicates code.

This is where protocol extensions come in: they let us attach default implementations of methods directly to the protocol, and all conforming types can use that implementation rather than adding their own.

To try this out, remove all three methods from **Book** and **AudioBook**. This will make Xcode throw up errors because they no longer conform to their protocols, but that's OK – we're going to replace them with protocol extensions.

Add these extensions to your playground:

```
extension Purchasable {
    func purchase() {
```

```
        print("Bought!")
    }
}

extension Printable {
    func printOut() {
        print("Printing…")
    }
}

extension Borrowable {
    mutating func lend(to person: String) {
        borrowers.append(person)
    }
}

extension Listenable {
    func playPreview() {
        print("Previewing…")
    }
}
```

With those in places, Xcode's errors will go away: both our types conform to their protocols, because the default method implementations will be used.

Now we can go ahead and implement **Magazine** in almost no code at all:

```
struct Magazine: Purchasable {
    var name: String
    var price: Decimal
}
```

It conforms to **Purchasable**, which means it will get the default implementation of **purchase()**

– no need for us to do anything.

However, our default method implementations aren't that great. Take a look at this one:

```
func purchase() {
    print("Bought!")
}
```

It gets applied to everything that conforms to **Purchasable**, but it doesn't actually say *what* was bought. We *could* modify it to take a string, like this:

```
func purchase(item: String) {
    print("Bought \(item)!")
}
```

But that isn't ideal either – we'd need to attach the same string to **printOut()** and **playPreview()**, and in any case these methods are attached to **Book**, **AudioBook**, and **Magazine** so they ought to be able to use their **name** properties rather than adding them as a parameter.

As discussed in the Protocols chapter, one protocol can inherit from another. This is particularly helpful because protocols are able to describe things (e.g. **UITableViewDataSource**) and abilities (e.g. **UIStateRestoring**), which means we can create a new protocol to represent items in our book store.

Add this now:

```
protocol Item {
    var name: String { get set }
}
```

Now, we could add that to the list of protocols implemented by **Book**, **AudioBook**, and **Magazine**, but a better idea is to use protocol inheritance – add them to **Purchasable**, **Printable**, **Borrowable**, and **Listenable**, like this:

```
protocol Purchasable: Item {
```

If you do that for all four protocols, then all conforming types will also conform to **Item**. This doesn't require any extra work because they already have a **name** property, and that's the sole requirement of the **Item** protocol.

With this change in place all our protocols know that **name** exists in their conforming types, so we can rewrite the **purchase()** method to this:

```
extension Purchasable {
    func purchase() {
        print("Bought \(name)!")
    }
}
```

As you can see, protocol-oriented programming lets us build new types complete with functionality easily. Even better, it does so in a way that's composable and isolated, which means we can build bigger protocols by inheriting smaller ones, and we can even take out a single protocol without the whole type hierarchy crashing down.

The limits of protocol extensions

Protocol-oriented programming is not a silver bullet – it is one solution that helps reduce complexity and coupling, but because protocol extensions are not supported in Objective-C there's a limit to how far you can use them in practical Apple development.

When you see POP working beautifully in examples like the one above, remember: in a real app, when you're facing **UIViewController** and **UITableViewDataSource**, there are some things it simply isn't designed to do. You *can't* write an extension for **UITableViewDataSource** that provides default implementations of **cellForRowAt**, because Objective-C is unable to use protocol extensions. In this situation, you're looking for a custom class that conforms to **UITableViewDataSource** – don't try to fight the system.

When working with UIKit, AppKit, and WatchKit, be prepared to use class-only protocols so you're able to use mutation and reference semantics more easily, remember to lean heavily on constrained extensions so that you can write protocol extensions that apply only to a subset of conforming types, and don't be afraid to use associated types – their ability to let conforming types customize the protocol as needed is extraordinarily helpful if you handle it carefully.

Summary

Protocol extensions are a headline feature of Swift, and in protocol-oriented development we have an interesting new approach to developing large-scale apps without large-scale coupling.

Here are my suggested guidelines:

- When extending a protocol, extend the most specific protocol that solves your problem. There's no point extending **Numeric** (all numbers) if all you actually want is **BinaryInteger** (integers) – it may add a maintainability burden and certainly pollutes the namespace.
- Composing protocols is a great way to group things together and can aid readability, but it only really works when your initial protocols are suitably isolated. No one wants an all-singing, all-dancing super protocol that takes an age to conform to or that brings with it dozens of default method implementations.
- Constrained protocol extensions let you implement the same method in multiple different ways, which is the protocol equivalent to the Gang of Four's Composite pattern – multiple extensions working side by side under a single protocol.
- Don't try to fight the system. So much of Apple development relies on Objective-C-compatible code (i.e. classes and **@objc**), and protocol extensions just don't work well there.
- Your protocol extensions don't need to provide default implementations for all methods – it's OK to force types to implement some themselves.

Accessors

The accessor pattern allows us to route property reading and writing through method calls, usually called getters and setters. This allows us to modify property functionality in one central location, improves flexibility, and hides implementation details.

Accessor methods can be annoying in other languages. In Java, they are so common that IDEs such as NetBeans generate them for you – you just define a property called **name** that holds a string, and it will generate **getName()** and **setName()** accessors for you to use.

Don't get me wrong: this is a good feature to have, but it's *annoying* – you either generate them very early and feel like you're wading through mud when you're just trying to make a prototype, or you generate them later (when your prototype inevitably lurches towards being production code) and find you need to refactor everything.

Again, some IDEs will do a passable or even good job of turning pure properties into accessor methods, but in Swift this is a non-problem. You see, in Swift you can go ahead and start with pure properties for as long as you want, and retroactively change them to be accessors. You don't need to rewrite all the call sites, or indeed touch anything other than the declaration of your property inside your data type.

This isn't actually a *Swift* feature: Objective-C did something similar, albeit predictably a little more mangy. If you declared a property called **badger** in an Objective-C class, it would automatically synthesize methods for you called **badger** (retrieve the property value), **setBadger**, **addBadger**, **removeBadger**, **intersectBadger**, **insertBadger**, **countOfBadger**, **replaceBadgerAtIndexes**, and many more – 19 in total.

While that is just an unfortunate side effect of Objective-C's informal protocols, what happened underneath was that whenever you accessed the **badger** property directly Objective-C would automatically route your access through the auto-generated setter and getter, which read from an internal **_badger** property that was invisible externally. If you wanted, you could then retroactively write your own custom getter and setter to add your own functionality.

Swift takes this idea and makes it significantly better. Using Swift you can do all these without

breaking any external code:

1. Convert a stored property into a getter and setter with private storage.
2. Convert a stored property into a computed property with no storage.
3. Convert a stored property into associative storage.
4. Make a property lazy.
5. Add property observers.

All of those effectively let you retrofit getters and/or setters without worrying about refactoring the rest of your project. So, although the accessor pattern is still important, the urgency of using it is gone – you can introduce it later as needed.

Retrofitting accessors

Of the list above, options 1, 4, and 5 are particularly common patterns in Swift: adding getters/setters, making a property lazy, and adding observers. However, they are all mutually exclusive: you choose either getter/setter, lazy, or property observers, but none can be used together. As a result, it's important to be aware when each pattern makes most sense so you can choose wisely:

- Add getters and setters when you need to add validation to your data or otherwise need to modify it before storage. You can also add a getter by itself to make something read-only, or make the setting have a lower visibility than the getter.
- Use lazy properties when you have performance-critical code that should be run no more than once and cached in the future, or when you want to implement the Singleton pattern.
- Use property observers when you must respond to a value changing.

Helpfully, it's usually easy enough to move from property observers to getters and settings, because you can simply move your **didSet** code into your new setter. The only major difference is that there is no equivalent to **willSet** so you need to run your **willSet** code, apply the change, then run your **didSet** code.

Let's look at a practical example in a playground. Start with this simple struct:

```
struct Brewery {
    var bestSellingBeer = "Guinness"
}

var diageo = Brewery()
print(diageo.bestSellingBeer)
```

That will print “Guinness”, as you’d expect. However, a nefarious developer could come in and write code like this:

```
diageo.bestSellingBeer = "Tea"
```

Clearly Diageo’s best-selling beer is *not* tea. With Swift we can disallow this kind of change by specifying a different access visibility for getter and setter. The syntax is a little strange when you first see it: you specify the getter’s visibility first, followed by the setter’s with **set** in parentheses.

Try using this:

```
public private(set) var bestSellingBeer = "Guiness"
```

That means “allow everyone to read the property, but only methods that belong to this class can change it.” As a result, the “Tea” line will now error, because it doesn’t have permission to make such a change.

We could modify this property so that getters and setters are used. You can combine getters and setters with varying visibility, although it does make the syntax look clumsier than ever.

To act as a true getter/setter, you must first define a *backing store* – a private property with a similar name where the actual data is stored. In Swift the convention is that this property should have the same name as your getter/setting property except with a leading underscore.

In our example, that means adding this:

```
private var _bestSellingBeer = "Guinness"
```

With that in place we can now make the **bestSellingBeer** property a computed property that reads and writes the backing store:

```
public private(set) var bestSellingBeer: String {
    get {
        return _bestSellingBeer
    }

    set {
        _bestSellingBeer = newValue
    }
}
```

If you did have **willSet** and **didSet** logic to add, you'd so like this:

```
set {
    // will set code
    _bestSellingBeer = newValue
    // did set code
}
```

As for lazy properties, you write a setter closure that gets immediately applied when the property is first read; Swift generates the getter for you, because it's just a matter of returning what was calculated in the setter.

For example:

```
lazy var revenue: Int = {
    print("Running expensive code")
    return 1_000_000_000
}()
```

Note: Lazy properties will only have their closure run when they are accessed for the first time. If you want the closure to be run always, just omit the **lazy** keyword.

Summary

The accessor pattern is hugely important in some languages, but Swift takes away most if not all of the worry by allowing you to retroactively change how properties behave.

Here are my suggested guidelines:

- Don't implement accessor methods until you need them; it just adds clutter to your code.
- Do give properties the most limited visibility you can. Going from **internal** to **public** won't break any code, but going in the other direction will.
- Always make backing store properties **private**. Even **internal** leaks too much information.
- Worry more about the interfaces you expose and less about the implementation behind them – you can change between stored properties, computed properties, and lazy properties at will, but a bad interface is bad for good.

Keypaths

Keypaths allow you to refer to properties without actually invoking them – you hold a reference to the property itself, rather than reading its value. Their use in Swift is still evolving, and in many ways they are influenced by keypath support in Objective-C, but already some clear design patterns are emerging.

Today the primary usage of Swift keypaths are:

1. Key-value coding.
2. Key-value observing.
3. Bindings on macOS.
4. Acting as adapters

I want to walk you through examples of all four so you can see why keypaths are so useful – and how their use is only likely to grow as Swift developers further.

Key-value coding

Modern Swift keypaths were introduced in Swift Evolution proposal SE-0161, "Smart KeyPaths: Better Key-Value Coding for Swift". However, if you weren't already familiar with key-value coding in the first place, the concept of *better* key-value coding probably doesn't mean much.

Key-value coding (KVC) is more or less the opposite of the associative storage pattern. With associative storage, we simulate properties in an extension that work by dipping into a dictionary, adding safety where there was none before. With KVC we do the opposite: we allow users to access properties using strings, more or less tossing away type safety.

Before you think "not another hideous Objective-C feature creeping into Swift!" and skip over this section, hold on a moment. Swift on iOS has no use for keypaths as strings, which is why they are called *better* KVC. To avoid conflating two very different things, you can find more information on KVC using strings in the Bindings on macOS section below.

Here I want to focus on the *better* KVC using Swift keypaths, and how they can help you build better apps. Let's look at an example of what keypaths do – create a new playground and give it this code:

```
struct Character {
    var name: String
    var city: City
}

struct City {
    var name: String
    var sights: [String]
}

let london = City(name: "London", sights: ["Tower of London",
"Buckingham Palace"])
let bear = Character(name: "Paddington Bear", city: london)
let detective = Character(name: "Sherlock Holmes", city:
london)
let spy = Character(name: "James Bond", city: london)
```

That creates a list of characters, with each one marked as living in London. A keypath allows to refer to properties themselves rather than reading their values. For example, it's easy to see that this code in Swift will print "London":

```
print(spy.city.name)
```

That reads the **name** property. But if we want to read the value later on somewhere else entirely? This is where keypaths can help: you're not asking for the specific value that is stored right now, you're asking for where to look for a value so that when the value changes you can just read whatever it became.

To give you a real-world analogy, think of something like DNS. When you want to visit a

website you enter hackingwithswift.com and that domain name gets converted into an IP address like 151.101.193.140 that your web browser can then connect to. While it's possible to enter that IP address by hand, it means when hackingwithswift.com changes address you won't get updated. Using the domain name – hackingwithswift.com – means your request will get looked up every time, and converted to the IP address.

This is exactly how keypaths work: we can create a reference to where a character's city name is stored without actually reading it directly. We can then apply that reference to any number of other values to read *their* city names.

Keypaths in Swift are identified using a backslash, followed by the path you want to read. These backslashes do rather stick out in code, but that was intentional – in the Swift Evolution proposal it was cited for its "behave differently for a moment" connotation.

Try adding this keypath to your **Character** example

```
let characterName = \Character.name
```

That creates a keypath called **characterName** that refers to the name of any character. We can then use it like this:

```
print(bear[keyPath: characterName])
print(detective[keyPath: characterName])
print(spy[keyPath: characterName])
```

You can read properties within properties just by digging deeper, like this:

```
let cityName = \Character.city.name
print(spy[keyPath: cityName])
```

And you can even subscript arrays like this:

```
let exampleSight = \Character.city.sights[0]
let characterSight = spy[keyPath: exampleSight]
```

```
print(characterSight)
```

Using keypaths in this way allows us to refer to the same property in multiple types without having to repeat ourselves. If you decide you want a different you can just change one line, and it will be applied across the board.

Key-value observing

Now that you've seen KVC, it's time to talk about its counterpart KVO – key-value observing. KVC lets us refer to a property itself rather than its value, which is the difference between "name" and "paul". KVO allows us to observe properties for changes by referencing them as keypaths.

You're probably used to property observers in Swift, because **willSet** and **didSet** are part of the accessor pattern. KVO is the ability to add property observers for things you *didn't* create, such as monitoring someone else's code.

KVO has two requirements, both of which are likely to seem unpleasant at first. First, it can only be used on classes that inherit from **NSObject**. Second, every property that you want to use with KVO must be marked using both **@objc** and **dynamic** so the resulting getter and setter goes through the Objective-C runtime and will trigger observer updates.

Having to inherit from **NSObject** isn't uncommon, but having to mark every property as **@objc dynamic** is deeply un-Swifty. Don't lose heart, though – bear with me.

First, let's examine a pure Swift example. Replace your current playground code with this:

```
class CoffeeShop: NSObject {
    @objc dynamic var remainingBeans = 10_000
}
```

That creates a **CoffeeShop** class with one property that tracks how many coffee beans are remaining. We might want to observe that for a given coffee shop, so that we can buy more beans as needed.

First, we need to create a shop we want to monitor:

```
let blueBottle = CoffeeShop()
```

Now we can call its **observe()** method, passing in the keypath we want to watch and whether we want to be notified of the new value it's changing to or the old value it's changing from. This takes a closure to run when the value updates, which should accept the object that changed as well as the change itself. Using the change parameter is best avoided unless you specifically need it, because it's a nest of optionals – it's much easier to read the new value from the object that was passed in.

Here's how we would observe the **remainingBeans** property being changed:

```
blueBottle.observe(\.remainingBeans, options: .new)
{ coffeeShop, change in
    print(coffeeShop.remainingBeans)
}
```

If you now modify that property several times, you'll see each new value being printed:

```
blueBottle.remainingBeans -= 1
blueBottle.remainingBeans -= 5
blueBottle.remainingBeans -= 3
```

This is neat, but all that **@objc dynamic** is just *unnatural*. Having to use classes is bad enough, but inheriting from **NSObject**, adding **@objc**, and adding **dynamic** are all things that Swift developers are naturally averse to.

However, when you're working with many of Apple's APIs you get all this for free without any extra work. For example, most UIKit classes already fit the criteria by default: they are native Objective-C classes, which means they inherit from **NSObject** and effectively use **@objc dynamic** for most of their properties. As a result, you can observe many of their values changing as needed.

So, we could create an example UIView then monitor its **isHidden** property, like this:

```
let vw = UIView()

vw.observe(\.isHidden, options: .new) { view, change in
    print(view.isHidden)
}

vw.isHidden = true
vw.isHidden = false
vw.isHidden = true
```

This effectively lets you hook into almost everything in UIKit. I say "almost" because not everything in Objective-C is observable – you should look for documentation that says "this property is KVO-compliant" or similar. If you find something that ought to be KVO-compliant and isn't, file a radar with Apple at **http://bugreporter.apple.com** – I wish **fractionComplete** on **UIViewPropertyAnimator** were KVO-compliant, for example.

Before moving on, I should add that just because you *can* observe something doesn't mean you *should*. Even though it has worked this way since the beginning, Apple didn't explicitly design UIKit to be KVO-compliant and could change things at any point in the future.

Note: When you observe something like this, you'll be handed an instance of **NSKeyValueObservation**. This is sort of like being given a ticket when you hand your coat in at a cloakroom – it doesn't do anything special, but it's a reminder of what you did. As soon as that **NSKeyValueObservation** falls out of scope and gets destroyed, your observation will stop.

Bindings on macOS

Believe it or not, it's possible to build real, useful macOS applications using only a dozen lines of code, largely thanks to a technique known as *bindings*. A binding is a connection between some model data (e.g. the current temperature in your office) and a view (e.g. a label showing

that temperature), and they are produced by adding strings to various fields inside Interface Builder.

In the same way that storyboard outlet and action names get converted from strings into real code at runtime, so too do these bindings – if the model data changes it will update the label automatically.

I'm not going to give an example of bindings here, partly because they only exist on macOS, but partly also because so many developers don't use them. Bindings are used in two of the projects in my Hacking with macOS books, but even then I try to make it clear they come with significant downsides:

At the risk of sounding like a broken record, bindings are a really clever piece of technology that allow you to produce software with very little code, but sometimes adding all that "clever" is the last thing you want. Sometimes bindings are the right choice, and you'll find some Cocoa developers who go to great lengths to use bindings in their projects. On the other hand, many people consider bindings to be a debugging black hole where you end up trusting your app's stability to Cocoa rather than your own code.

Acting as adapters

Keypaths are invaluable as a way of letting us reference different types in a natural way. Try creating these two structs in your playground:

```
struct Person {
    var socialSecurityNumber: String
    var name: String
}

struct Book {
    var isbn: String
    var title: String
}
```

I've purposefully made them completely different so you can see how this pattern works.

Both **Person** and **Book** have an identifier that is unique: **socialSecurityNumber** and **isbn** respectively. If we wanted to be able to work with identifiable objects in general, we could try to write a protocol like this:

```
protocol Identifiable {
    var id: String { get set }
}
```

That would work well enough for **Person** and **Book**, but it would struggle for anything that didn't store its identifier as a string – a **WebPage** might use a **URL**, and a **File** might use a **UUID**, for example.

Worse, it locks us into using **id** as the unique identifier for all our data, which isn't a descriptive property name – you need to remember that **id** is actually a social security number for people and an ISBN for books.

Keypaths can help us solve this problem, allowing us to use them as adapters for very different data types – i.e., allow them to be treated the same even though they aren't the same. The Gang of Four book calls this the adapter pattern: something that allows incompatible data types to be used together by wrapping an interface around them.

What we're going to do is create an **Identifiable** protocol that uses an associated type (a hole in the protocol), then use that as a keypath. What this means is that every conforming type will be asked to provide a keypath that points towards whatever property identifies it uniquely – **socialSecurityNumber** and **isbn** for our two example structs.

First, add this protocol:

```
protocol Identifiable {
    associatedtype ID
    static var idKey: WritableKeyPath<Self, ID> { get }
```

```
}
```

WritableKeyPath is one of several variants of Swift's keypath types that let us store keypaths for later. In this case we're saying that the keypath must refer to whichever type conforms to the protocol (**Self**) and it will have the same value as whatever is used to fill the **ID** hole in our protocol.

Now let's update both **Person** and **Book** so they conform to the protocol. This means having adding **Identifiable** to their list of conformances, then defining **idKey** to point to whichever of their properties is their unique identifier:

```
struct Person: Identifiable {
    static let idKey = \Person.socialSecurityNumber
    var socialSecurityNumber: String
    var name: String
}

struct Book: Identifiable {
    static let idKey = \Book.isbn
    var isbn: String
    var title: String
}
```

Swift's type inference is extremely clever here. When it looks at **Person** it will:

1. Remember **socialSecurityNumber** is a **String**.
2. Store that **idKey** points to **Person.socialSecurityNumber**, which is a string.
3. Match **idKey** in **Person** with the same property in **Identifiable**.
4. Resolve **WritableKeyPath<Self, ID>** to **WritableKeyPath<Self, String>**.
5. Understand that **associatedtype ID** is a hole that is being filled by a string.

I know that Swift code can sometimes take a long time to compile, but you have to admit it's pretty darn amazing.

What we've achieved here is that totally disparate data types – structs that are designed to store properties in whichever way works best for them rather than following some arbitrary names imposed by a protocol – are able to be used together. All we care about is that they conform to **Identifiable**: once we know that we also know it has an **idKey** keypath that points to where its identifying property is.

Putting all this together we can print the identifier of any **Identifiable** type like this:

```
func printID<T: Identifiable>(thing: T) {
    print(thing[keyPath: T.idKey])
}

let taylor = Person(socialSecurityNumber: "555-55-5555", name:
"Taylor Swift")
printID(thing: taylor)
```

Yes, that code leverages generics, keypaths, and associated types all in one, and with surprisingly little code – Swift is an extremely powerful language when you really lean on it. The end result is that we've been able to separate our architecture from implementation details – we've let types be expressed naturally, then used protocols to overlay an adapter on top to allow those types to be used together.

Summary

Keypaths as used in modern Swift (as opposed to macOS bindings) are type-safe and efficient, and allow us to write code that would otherwise be impossible. In their simplest form they allow us to reference a property without reading it, but as you've seen they are also useful for key-value observing, and for creating adapters between incompatible types.

Here are my suggested guidelines:

- If you have a project that mixes Objective-C code and Swift code, make your instance variables KVO-compliant where possible.

- If you're writing your own Swift code, using **didSet** and **willSet** is much better than using KVO, not least because it avoids having to use **@objc dynamic** everywhere.
- Keypaths are a great way of identifying specific data inside your types. We used it for identification here, but it could be used to identify which property stores a type's display name, or indeed anything where property *names* might vary when their *meaning* remains the same.
- Don't use bindings on macOS. In the words of David Smith, one of Apple's Cocoa engineers, "Most large Mac apps I'm aware of don't use bindings. It's appealing, but difficult to debug when things go wrong."

Anonymous Types

Anonymous types allow you to deny the compiler knowledge about the real type of data it is working with, usually to reduce coupling or to achieve functionality that would otherwise be impossible.

It is the compiler's job to ensure you create and use your objects as they were intended, which means that sometimes using anonymous types is the equivalent of saying "I know something the compiler doesn't" or "I know better than the compiler." While this might sometimes be the case, usually a better approach is to give the compiler the additional information it needs.

One of the books that was always on the recommended reading list for Objective-C programmers is called Cocoa Design Patterns. It was written years before Swift was announced, and so has not aged terribly well with regards to type safety. For example it says "the ability to send any message to any receiver and do so without compiler warnings eliminates the need for complex and error-prone extensions to the Objective-C language" (Buck and Yateman, Cocoa Design Patterns, 2010) – people really used to think this was a good idea.

Of course, Swift does its very best *not* to give you the ability to send any message to any receiver, and with good reason. Not only does it make your code harder to reason about, but it makes it more complex and error-prone – precisely the things that Cocoa Design Patterns claimed it avoided.

However, as with so many Swift patterns, anonymous types are baked into the language thanks to inheritance from Objective-C. In fact, Swift has two different anonymous types, known as **Any** and **AnyObject**. You will almost certainly need to use both, which means understanding which pattern they are trying to solve and how they vary.

Before you continue, I should make it clear that using anonymous types in Swift should be your last resort, not your first or even your tenth, which is why I've put this chapter at the end of the section on language patterns. They are a useful pattern in Swift because they solve otherwise unsolvable problems, but that doesn't mean they are a good basis for architecting

your code.

AnyObject

The **AnyObject** type can be any kind of *object*, i.e. a descendant of a class but not a struct or enum. This was used as the standard anonymous type before Swift 3, but caused all sorts of headaches because Swift developers lean heavily on value types that required typecasting work alongside **AnyObject**.

However, **AnyObject** carries an important connotation: it effectively treats your data like an Objective-C object of the type **id**. In Objective-C, **id** is any type that can receive messages, but because the compiler had no idea *what* messages it could receive they allowed it to receive *any* message – it was up to the developer to ensure they did something sensible with that power.

Swift bridges that behavior fully, which means that if you create an **AnyObject** type Swift will allow you to call any method you like on it. Literally anything: you can call **viewDidLoad()** on a string or try to read its **popoverController** property if you like.

To add *some* degree of safety to all this, Swift wraps methods as implicitly unwrapped optionals and properties as regular optionals. These are seen very rarely and are mostly used with the utmost caution because when you get them wrong Swift won't warn you.

Consider the following code:

```
let str: String
let regularOptional: String? = "hello"
let implicitlyUnwrapped: String! = "hello"
```

In that situation we're aware that Swift will refuse to compile **str = regularOptional** but will allow **str = implicitlyUnwrapped** because we're effectively saying "trust us." If **implicitlyUnwrapped** happened to be set to **nil** our app would crash because setting a string to **nil** isn't allowed.

The same rules apply to implicitly unwrapped optionals: you can use them as if they were

regular methods even though they may not exist.

To try this out, create a playground with this constant:

```
let message = "hello" as AnyObject
```

We know that **message** is a string, but we're actively denying the compiler knowledge by casting it as **AnyObject**. Swift's string is actually a struct, so this will cause the compiler to bridge **String** to **NSString** so that it can be used with **AnyObject**.

Now type "message." to trigger code completion and marvel at the hundreds – if not thousands! – of methods and properties that exist. Swift is effectively overlaying all methods and properties of all objects in UIKit onto this single constant, allowing you to be as flexible (read: *reckless*) as you wish. If you were to add **import SpriteKit** to the top of the playground, you would also be able to use SpriteKit methods and properties – Swift brings in everything it can see. In addition, Swift brings all your own Swift methods and properties that were marked using **@objc**, making this a real free for all.

However, these methods are implicitly unwrapped, which means you can call them like regular methods if you aren't careful. For example, this will cause your code to crash:

```
message.viewDidLoad()
```

If you ever find yourself in this situation, the correct call to make is this:

```
message.viewDidLoad?()
```

That uses optional chaining to check whether the method actually exists before running it.

Now, all this might sound like a recipe for disaster, so why would we ever need to worry about using **AnyObject** types in Swift?

There are three situations where it's useful:

1. When declaring class-only protocols. In this situation, **AnyObject** marks the protocol as

applying only to objects, but thankfully doesn't bring in the **id** mess.
2. When someone made a fundamental architecture design mistake several years ago and you're now doomed to live with it forever more.
3. When using Objective-C code that bridges very poorly to Swift and requires **AnyObject**-compatible types.

A good example of code that bridges poorly to Swift is **NSCache**. This is an extremely helpful class that Objective-C developers relied on frequently, but is almost unused in Swift because it bridges to Swift so badly. **NSCache** is a generic data structure used to hold temporary data that you want to keep in RAM, but has the added benefit of automatically ejecting data when the system hits memory pressure. Even better, it does this *before* your app gets a memory warning, giving it a pre-emptive chance to free up data before your app needs to start destroying useful things.

Honestly, it feels a bit sad when you see someone using a Swift dictionary for this purpose – it's a failure of Apple to make **NSCache** more Swift friendly, and a failure of teachers like me to teach new developers hard-won lessons from the past. However, you're reading this book because you want to know a *better* way of doing this, so here's an example with **NSCache** where only an **AnyObject** type is acceptable:

```
// define a class that is costly to make, so we want to cache
it if possible
class ExpensiveObjectClass { }

// create a cache that holds those objects, keyed by an
NSString
let cache = NSCache<NSString, ExpensiveObjectClass>()

// prepare to use an expensive object
let myObject: ExpensiveObjectClass

if let cachedVersion = cache.object(forKey: "CachedObject") {
```

```
    // use the cached version
    myObject = cachedVersion
} else {
    // create it from scratch then store in the cache
    myObject = ExpensiveObjectClass()
    cache.setObject(myObject, forKey: "CachedObject")
}
```

In that example, **NSString** works fine but **String** does not, and no amount of Swift bridging magic will help.

Any

The **Any** type is sometimes thought of as a superset of **AnyObject**, but that's only partially true. As you've seen, **AnyObject** does two things: it allows us to refer generically to all class instances, and it brings with it a boat load of Objective-C functionality.

Any takes the first of those two and adds the ability to refer to structs and enums, which makes it significantly more useful in Swift. However, it does not bring with it all the Objective-C cruft, which means you don't get the inherent ability to call practically anything unsafely – a win all around, really.

All that doesn't mean you can't mortally abuse **Any** if you so desire. One member of the Swift compiler team put forward some code that effectively allows us to try **Any** like **AnyObject** so that you can dynamically dispatch methods on **Any**. Here's a simplified version:

```
infix operator =>

func =><T, U>(obj: Any, method: (T) -> U) -> U {
    return method(obj as! T)
}
```

That jumble of parentheses typecasts an **Any** instance to whatever is being used for the method

call (**T**), then calls the method on it passing in any parameter. This means you can force strings to run **UIViewController** methods if you desperately need to:

```
var x = "hello"
(x=>UIViewController.viewDidLoad)
```

Again, this is the kind of thing you should only ever encounter if someone has done something rather dreadful.

Realistically, **Any** is far more likely to be encountered in these situations:

1. When using methods that are designed to pass on data without caring what it is.
2. Heterogenous collections where there is no meaningful superclass or shared protocol.
3. When making **IBAction** connections using Interface Builder.

The first of those happens more often than you might think. For example, the **performSegue()** method has this signature:

```
func performSegue(withIdentifier identifier: String, sender:
Any?)
```

Any is helpful here because there's no meaningful way to identify the sender: it could be a **UITableViewCell** or a **UIButton** that the user interacted with, or perhaps your own controller class that has finished doing some work and is ready to show the result.

The second situation, heterogenous collections, is actually suggested by the Swift compiler when you write code like this:

```
let array = [1, 2.0, "Fish"]
```

Swift will send you the error message "heterogeneous collection literal could only be inferred to '[Any]'; add explicit type annotation if this is intentional." This error is useful because it stops us creating heterogeneous collections by accident – Swift wants you to state explicitly it's what you want, to avoid problems later.

In this case the error message isn't strictly true. Like I said, **Any** is useful for heterogeneous collections where there is no *meaningful* superclass or shared protocol. In this case, the collection could be inferred as **[AnyHashable]**, but is that meaningful? It's down to you – if you're processing objects in the array using a set or dictionary, where hashability is a requirement then using **AnyHashable** is a shared protocol where that functionality matters. If you don't need hashability, then using **AnyHashable** is just a red herring that will throw others off – use **Any** instead.

Collections of **[Any]** are also a useful way to group together objects that use a protocol with associated types. When iterating over the items you can't use the protocol as part of a typecast – such a thing just wouldn't make sense given the hole in the protocol – but you can typecast to concrete types. For example:

```
protocol Identifiable {
    associatedtype ID
    var id: ID { get set }
}

struct Person: Identifiable {
    var id: String
}

let person: Any = Person(id: "Taylor Swift")
let array: [Any] = [person]

for item in array {
    if let item = item as? Person {
        print(item)
    }
}
```

The third instance where you're likely to see **Any** is when making actions using Interface

Builder. You've probably noticed this behavior in the past, but maybe you didn't question it: when you create an **@IBAction** using the assistant editor Xcode will automatically use **sender: Any** even though it can clearly see you're connecting a button, a segmented control, or whatever. It ignores the type by default, throwing away information that is probably valuable. Perhaps even more strangely, changing **Any** to a concrete type such as **UIButton** works just fine.

Contrary to what you might think, this is a feature rather than a bug. By accepting **Any** as a parameter, Interface Builder is allowing extremely loose coupling because its controllers have no idea what triggered a message to be sent. This means you could in theory connect an entirely different storyboard to the same controller without breaking compatibility, although I would think twice before doing such a thing. This is loose coupling in the extreme, and I suspect this goes a step too far for most modern developers.

Summary

Swift offers us two ways to make anonymous types, but nearly all the time you should prefer **Any** over **AnyObject**. Anonymity is primarily a way to let us group disparate things and reduce coupling, although remember that leaning too heavily on Objective-C intrinsics is the path to the dark side.

Here are my suggested guidelines:

- Try using **Any** everywhere you use **AnyObject** or a class that conforms to it. If it succeeds, you're better of for it.
- If you find yourself frequently creating **Any** collections, try creating a protocol that describes your types more precisely – you're giving both the compiler and other developers more information to work with, rather than trying to keep it all in your head.
- Even **Any** is open to abuse, and comes with its own costs. As useful as **Any** is, it ought to be your last resort.
- Although using **Any** for the sender in **IBAction** does allow loose coupling, I think the benefit is dubious at best and wouldn't stop anyone from changing it to a concrete type instead.

Chapter 5
Classic Patterns

Design patterns that are used widely by Apple developers.

Singletons

The singleton pattern restricts one type so that it can be instantiated only once. It's one of the few Gang of Four design patterns that most people recognize by name alone, which says a lot about how often it's used and how important it is.

Singletons are commonly used on Apple platforms, although they are frequently termed "shared instance". You've almost certainly used them yourself: **FileManager**, **UserDefaults**, **UIApplication**, and **UIAccelerometer** are all mostly used through their singleton implementations. How many file systems are there? One. How many instances of your application are there? Also one. These things are inherently singular, and it's the job the single pattern to enforce that singularity – to provide a property that lets us access a shared resource in an easy way.

In practice, the singleton pattern gets a little hazy. To be a true singleton, we must ensure that a type must be instantiated only once ever, which usually means hiding all initializers so that developers are funneled through a shared resource. However, in practice you can instantiate as many instances of **UIDevice** and **FileManager** as you like, and even **UIApplication** has a public initializer – albeit one that triggers a fatal error if you call it.

Singletons can also be hazy because they can skirt close to global variables – macOS developers even get a special global variable called **NSApp** that is the equivalent of running **NSApplication.shared**. Although singletons can be accessed globally, they have a few key differences from global variables:

- They are only created on first access, so it's possible they might never be created.
- You can funnel users through a getter/setter method that lets you track access for debugging purposes.
- They make it easier to change your app architecture if you later move away from singletons – just change from your shared instance to a new instance.

Singletons also have some unhelpful *similarities* with global variables, not least:

- They make testing harder. When all your code can modify the same shared resource you can no longer create isolated unit tests.
- They make concurrency problematic. If you have multiple threads accessing the same shared resource you might encounter locking problems or bugs.
- They make code harder to reason about. If you're writing a function that operates only using the parameters it is given then everything you need to think about is right in front of you.

It won't surprise you to learn that singletons are sometimes described as an *anti-pattern* rather than a pattern.

Singletons in Swift

The basic implementation of a Swift singleton looks like this:

```
class Settings {
    static let shared = Settings()
    var username: String?

    private init() { }
}
```

Adding a private initializer to the **Settings** class prevents it being called from outside the class. However, the class creates an instance of itself as a static property, which means the only instance of the **Settings** class is the one it created: **Settings.shared**.

The use of **class** rather than **struct** is intentional and important: we don't want people taking copies of the struct and modifying the copy. With a class, a reference type, all copies point to the same instance, so we can know for sure they are always modifying the singleton. Even though the property is declared using **let**, with classes that means the *reference* is constant but its *properties* aren't – you can't change the underlying **Settings** object it points to, but you can change the properties inside it.

You'll notice there is no code above to handle synchronization across threads, and neither is there any code to handle lazy loading. The magic is *there doesn't need to be* – in Swift any **static let** property is automatically a singleton and is automatically created lazy.

Tip: If you don't want your singleton to be subclassed make sure you declare it as **final**.

Checking for creation

Sometimes it's useful to know whether your singleton's shared instance has already been created before accessing it – if it's expensive to create, you might prefer to push creation on a background thread if you know you're in a critical task.

In Swift this is done using just two lines of code: one to store whether the shared instance exists already, and one to set that flag to true during initialization.

First, add this property to the **Settings** class:

```
var sharedInstanceExists = false
```

Now it's just a matter of flipping that property during inside your initializer. Add this to **init()**:

```
sharedInstanceExists = true
```

Hiding your singletons

Even when singletons are useful, I still consider them implementation details – something that happens, but I'd rather not expose them in my API. Swift's protocol extensions gives a better approach: it's simple to use and exposes less implementation detail.

As an example, consider the following singleton:

```
class Logger {
    static let shared = Logger()
```

```
    private init() { }

    func log(_ message: String) {
        print(message)
    }
}
```

Anywhere you need to log something you can use **Logger.shared.log()**, or you could even write a static method that removes the **shared** part if you wanted.

But if we don't want to expose the fact that a singleton is being used, a Swiftier solution is to create a **Logging** protocol like this:

```
protocol Logging {
    func log(_ message: String)
}
```

Anything conforming to that protocol must implement a **log()** method. Obviously we don't want to implement that in all our types, so we can write a simple protocol extension that calls our singleton:

```
extension Logging {
    func log(_ message: String) {
        Logger.shared.log(message)
    }
}
```

Finally, we can design any types we want, make them to conform to **Logging**, and call **log()** freely without those types having to be aware a singleton is being used:

```
struct MainScreen: Logging {
    func authenticate() {
        log("Authentication was successful!")
```

```
    }
}

let screen = MainScreen()
screen.authenticate()
```

If you adopt this approach you can switch away from using singletons relatively easily – all the places you use them don't change, and you just need to make them a property of your type instead.

Summary

Whether or not you consider singletons a pattern or an anti-pattern, the simple truth is that they are used extensively on Apple platforms both by Apple and third-party developers.

Here are my suggested guidelines:

- Think carefully whether your singleton is just a global variable in disguise. We all agree that global variables are bad, and a global variable masquerading as a singleton is no better.
- Name your shared instance using Apple's naming conventions: **shared**, **default**, and **current** are all common.
- If your goal is to have a true singleton – something that can exist no more than once – make sure you hide all initializers to external code.
- When you write methods for your singleton, remember they will be called from all parts of your app. If those methods rely on shared state, or if they take a long time to execute, you may encounter concurrency issues.
- If your singleton usage is a mere implementation detail, treat it like one and hide it behind a protocol extension.
- Keep in mind that singletons can become a problem – don't shy a way from removing them when they become a problem, even if they are important. As the saying goes, "don't cling to a mistake just because you spent a lot of time making it."

Responder Chain

The responder chain pattern asks an object to handle an event, but if that object can't or won't handle the event it gets passed on to another, and another, and so on down a sequential line of potential responders. The Gang of Four book calls it the Chain of Responsibility pattern, but Apple developers usually see it as "First Responder" – it's right there in every storyboard whether you've used it or not.

On macOS the responder chain is highly visible to developers: when a user clicks a menu item, there isn't a specific view controller you can attach a method to because potentially the user has five different windows open. Instead, you post your event to the responder chain and let the first responder – the object that is first in line to respond – have its chance at responding. If it can respond to the action it will, otherwise it will pass the action on to the next item in the chain, known as its next responder. In the Selectors chapter I discussed how you can use **sendAction** with **nil** to broadcast a message to the responder chain.

On other Apple platforms the responder chain exists and still works hard, but it's primarily invisible: chains are created implicitly as you add subviews, but you don't explicitly add or remove things. When the user touches their iPhone screen iOS will perform a hit test to figure out which view was under their finger. This is a matter of combining user interface checks with code checks: if the control is not obscured, has a non-zero alpha, has its **isHidden** value set to false, and has **isUserInteractionEnabled** value set to true, then iOS will check whether it implements **touchesBegan()**.

Many **UIView** subclasses implement **touchesBegan()** for you: **UIButton** uses it to depress the button, **UITextField** uses it to control the caret position, and **UISlider** lets users drag a value bubble around. However, many **UIView** subclasses such as **UIStackView** don't care, so they will pass your touch on to the next responder. For views that is usually their container view, but sooner or later you'll run out of container views at which point the next responder is the parent view controller.

If your view controller subclass doesn't implement **touchesBegan()**, then control will go it to its parent view controller. That will have views of its own, in which case they will be checked

to see if they implement the method, and if they don't then the parent view controller will get checked. This continues until all parent view controllers and their views have been checked, at which point your window (**UIWindow**) gets checked, and finally your app delegate.

See the chain in action

In a master-detail app it's possible that a dozen or more things could be checked to see if they implement a method, but as soon as any of them say *yes* the chain stops and that method implementation gets run.

To try this for yourself, create a new iOS app using the Master-Detail App template, then paste this code into DetailViewController.swift just below **import UIKit**:

```
extension UIView {
    override open func touchesBegan(_ touches: Set<UITouch>,
with event: UIEvent?) {
        print("UIView")
        next?.touchesBegan(touches, with: event)
    }
}

extension UIViewController {
    override open func touchesBegan(_ touches: Set<UITouch>,
with event: UIEvent?) {
        print("UIViewController")
        next?.touchesBegan(touches, with: event)
    }
}

extension UIWindow {
    override open func touchesBegan(_ touches: Set<UITouch>,
with event: UIEvent?) {
        print("UIWindow")
```

```
        next?.touchesBegan(touches, with: event)
    }
}

extension AppDelegate {
    override func touchesBegan(_ touches: Set<UITouch>, with
event: UIEvent?) {
        print("AppDelegate")
    }
}
```

That overrides **touchesBegan()** in all views, view controllers, and windows, as well as our **AppDelegate** class, all using a method that prints out where in the chain we are along while passing the message to the next responder.

When you run this app you'll see an empty table view. If you click there you'll see no messages, because **UITableView** is a subclass of **UIView** that handles **touchesBegan()** by itself. Click the + icon in the top right corner (*not* a **UIView** subclass so again you'll see no message), and the current date will be inserted in the table.

When you click that date you'll start to see messages being printed. If you click on the date itself you'll see "UIView" three times but if you click outside the label you'll only see it twice – the additional message comes from the **UILabel** receiving the message and inheriting our method through **UIView**.

Once the table cell is selected, click down below in the empty area of the table to trigger selection, and the detail view will appear. Now try clicking on the label, and you'll see this chain printed in Xcode's debug console: UIView UIViewController UIView UIView UIView UIViewController UIView UIView UIView UIViewController UIView UIViewController UIWindow AppDelegate. And that's just for a raw app template!

This chain happens for all your interactions when you're using any Apple platform, but even though it's invisible it enables loose coupling and all the power that comes with it. You can

think of the responder chain as being like an extended form of delegation: if object A can't handle something it passes it to object B, and so on until something *can* handle it or the chain ends.

The clincher here is that every object in the chain isn't affected by being in a chain, doesn't know its position in the chain, and doesn't care what comes next in the chain – they all just work together like a string of people passing buckets of water.

Walking the chain

There are two primary times when you'll want to dig into the responder chain: working with input fields or finding view controllers.

If you're in a situation where your iOS app has multiple **UITextField** instances lined up, users expect to be able to move between them by pressing Next/Return on their on-screen keyboard. There is no built-in way of making this happen, so we need to write code ourselves using one of several approaches.

The easiest approach is using view tags: give your text fields incrementing tag numbers, then make them all point to a common delegate – it might be your view controller, but it doesn't need to be.

Once that's done you can use the **becomeFirstResponder()** and **resignFirstResponder()** methods to manipulate which view is in control like this:

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {
    let nextTag = textField.tag + 1

    if let nextResponder =
textField.superview?.viewWithTag(nextTag) {
        nextResponder.becomeFirstResponder()
    } else {
        textField.resignFirstResponder()
    }
```

```
    return true
}
```

If you're desperately opposed to using tags, the other solution is to place your labels in an array, find the position of the text field that triggered the event, then move one down in the array.

Note: If you ever need to force the first responder to resign itself and aren't sure which text field is in control, it's easier to use **view.endEditing(true)**.

The other task you might want to do is find the view controller that is responsible for a particular view, usually to communicate something important. This is as easy as walking the responder chain looking for the first **UIViewController** you find, like this:

```
func findViewController() -> UIViewController? {
    if let nextResponder = self.next as? UIViewController {
        return nextResponder
    } else if let nextResponder = self.next as? UIView {
        return nextResponder.findViewController()
    } else {
        return nil
    }
}
```

This is one of those methods you should only use when you really need it – if you're able to call methods directly using a delegate or indirectly by posting notifications. While this approach does remove any sort of coupling, ideally you don't want child views manipulating their view controllers.

Building your own chain

The responder chain is built into Apple platforms thanks to the **UIResponder** and

NSResponder classes that **UIView** and **NSView** inherit from. However, it's a nice and easy pattern to implement in your own code for things when you want the functionality without the attached user interface elements.

There are three ways to build responders:

1. When a responder can handle an event it does so and the chain stops.
2. When a responder can partly handle an event it does so and passes the remainder up the chain.
3. When a responder can handle an event it does so and passes it up the chain in full.

The first type is how UIKit and AppKit interactions work: the first view that can handle an interaction does so and the chain stops.

The second type is a bit like a manager signing off a deal then forwarding it to their manager to sign off, who forwards it onto the CEO to sign off. Each manager adds their own stamp to the deal so that responsibility is shared. In code, you might ask one thing to run its own validation an object, and it can then pass that object onto other responders to validate.

The final type would be used if your user saved a change to a recipe they were editing: you give that to the current view controller so it can update its layouts, then pass it on to a database that saves the change locally, which passes it on to another responder to save that change to the cloud. This is an implementation the Decorator pattern, which is covered in its own chapter.

We're going to implement the most common situation here: a chain of objects, of which only one implements a certain method.

Create a new playground and give it this code:

```
protocol Responder {
    var next: Responder? { get set }
    func run(selector: Selector)
}
```

That creates a new **Responder** protocol so that all conforming types must define what comes next in the chain (if anything), as well as a method that will run a selector if possible or forward it otherwise.

Next we're going to implement the "run selector or forward to chain" class. This needs to inherit from **NSObject** because that gives us the **responds(to:)** method that lets us query at runtime whether the current object can run a selector. If it can we call it, otherwise we call **next?.run()** to pass it down the chain.

Add this class now:

```
class Control: NSObject, Responder {
    var next: Responder?

    init(next: Responder? = nil) {
        self.next = next
    }

    func run(selector: Selector) {
        if responds(to: selector) {
            perform(selector)
        } else {
            print("Forwarding to next responder")
            next?.run(selector: selector)
        }
    }
}
```

Finally, we need a class that actually *will* respond to the selector we're going to send. This will inherit from the **Control** class defined above so that it gets all the default functionality, then add one method on top:

```
class MessagingControl: Control {
```

```
    @objc func printMessage() {
        print("Running!")
    }
}
```

As you can see, there are **print()** calls for "Forwarding to next responder!" and "Running!" so you can see the chain action.

Now it's just a matter of setting up some responding objects and kicking off the chain:

```
let root = MessagingControl()
let first = Control(next: root)
let second = Control(next: first)
let third = Control(next: second)

let action = #selector(MessagingControl.printMessage)
third.run(selector: action)
```

You should see "Forwarding to next responder" three times followed by "Running!" as our chain passes the message along.

Note: I've used selectors here because that's what **UIResponder** and **NSResponder** do, but you could easily pass a message struct – each object can inspect the struct to see whether it contains a message they care about, and pass it on if not.

Summary

Whether you use it directly or not, the responder chain is working away in the background to make sure all objects get their chance to participate in the way your app works.

Here are my suggested guidelines:

- You can invoke the responder chain by posting actions to **nil** and letting the system find the first responder for you.

- The responder chain should end as soon as one object can handle the event fully. If it doesn't, this is more like the decorator pattern.
- If you're implementing your own responder chain, consider adding a boolean return that lets you check whether any object handled the action.
- No matter whether the chain ends with one object, each object handles part of a message, or each object handles all of a message, responder chains are only called in one direction. While they can pass a value back, they shouldn't be *called* backwards.

Template Method

The template method pattern lets us change one or more steps in a process by encouraging subclasses and conforming types to selectively replace default implementations with their own. Subclassing is usually designed so that we can add things to an existing implementation, but the template method focuses on *replacing* implementations. More specifically, you replace methods that are called as part of an existing algorithm or process rather than directly by your own code, which has led to the template method sometimes being called the Hollywood principle: "don't call us, we'll call you."

I already covered one implementation of template methods in the Archiving chapter: when you implement **NSCoding** and **Codable** you provide methods that should be run as part of the archiving process. You don't care how the rest of that process runs, and you don't call the encoding methods yourself. Instead, **NSCoding** and **Codable** take care of setting up the environment and encoding everything fully, with your custom method replacing any default functionality.

Another good example is the **drawRect()** method of views: this causes views to render their contents and is overridden by any subclass of **UIView**/**NSView** that wants custom drawing. However, you don't call **drawRect()** directly – in fact Apple's documentation says explicitly, "you should never call this method directly yourself."

Remember, the key principle is "don't call us, we'll call you". In this case, if you want a view to redraw you need to call its **setNeedsDisplay()** method, which the system can then queue up and execute as needed. This allows the system to coalesce multiple calls into one redraw operation – if you call **setNeedsDisplay()** five times, the view will only be drawn once.

Designing for the template method

The template method pattern is used extensively on all Apple platforms – just think about any method you've overridden that you don't actually call yourself. Common examples are things like **didReceiveMemoryWarning()** and **hitTest()** , but there are countless numbers of them. Often you can replace multiple parts in a given process: **loadView()**, **viewDidLoad()**, and

viewDidAppear() are only a handful of replaceable methods inside **UIViewController**, and you can implement as many or as few as you want.

This is helpful, because when you're designing your own architecture to draw on the template method pattern you have a lot of experience of being the consumer – you should already know how it works.

As you know, all view already have a built-in **drawRect()** method that provides a default implementation. This is important: your process or algorithm ought to work normally even if no parts of it are overridden. Your default implementations don't need to do anything special (or indeed anything at all!), but it's important that it at least exist.

You should also provide clear guidance to subclassers whether your default implementation can, should, must, or must not be called by their overridden implementation. There are no rules here regarding naming that would make your intention clear, so it's down to you to add documentation.

In Apple's own code, you *can* call **super.drawRect()**, but you only *should* call it if you're subclassing something more specific than **UIView**. You *should* call **super.viewDidLoad()** (even though it doesn't do much today, that might change in the future), and you *should* call **super.hitTest()**, because even though you'll add your own logic on top you should at least let it do most of the heavy lifting for you.

In contrast, there are many methods you *must* call: **super.viewWillAppear()**, **super.removeFromParentViewController**, and **super.didReceiveMemoryWarning()** all must be used if you want to avoid problems at runtime. In fact, if you don't call **super.init()** your code won't even compile. Default implementations you *must not* call are rare, but I can think of at least two: **loadView()** and the **start()** method from **Operation**.

The status of all these implementations is always documented clearly – watch out for language like "You should never call this method directly yourself." Developers are used to this approach, so it's a good idea to follow it yourself.

The key is to make sure you implement your algorithm or process in a way that each step is a

single, isolated segment. If you put more than one step in each method someone trying to override that step now needs to do two things rather than just one, which probably isn't what you intended.

An example with classes

Even though you've already used the template pattern method, it's worth looking at a simple example so you can see how it could be applied in your own code.

Language compilers like Swift's own "swiftc" usually work as a series of steps something like this:

1. Lexical analysis: reading your typed letters and converting them to tokens like "function", "condition", and "loop".
2. Parsing: checking that your tokens appear in a meaningful order – something like "func class open public try" is lexically valid in Swift but won't parse.
3. Generating intermediate code: this is internal code that represents all the operations in your program in a simplified form.
4. Optimization: where the compiler rearranges, replaces, or removes instructions in your code to be more efficient.
5. Output as binary: writing the finished executable code.

This structure allows compilers to be hugely flexible: one compiler can read a variety of languages by replacing steps 1 and 2, and target a variety of architectures by replacing step 5, all while sharing the same intermediate code and optimization techniques.

Those five steps can be implemented as independent methods so that any of them can be overridden without affecting the others. Start by creating a new playground, then give it this code:

```
class Compiler {
    func analyze() { }
    func parse() { }
```

```
    func generate() { }
    func optimize() { }
    func output() { }

    func compile(filename: String) {
        analyze()
        parse()
        generate()
        optimize()
        output()
    }
}
```

That has empty implementations of each method (I'm *not* going to build a real compiler, as you might imagine!), each of which are called in sequence when **compile()** is run.

We could then create a subclass called **PascalCompiler** that replaces the first two steps with custom implementations, like this:

```
class PascalCompiler: Compiler {
    override func analyze() {
        print("Analysing Pascal")
    }

    override func parse() {
        print("Parsing Pascal")
    }
}
```

That leaves the other three steps untouched, doing all the work they need to do. You can then create an instance of **PascalCompiler** and run it like this:

```
let compiler = PascalCompiler()
```

```
compiler.compile(filename: "myfile.pas")
```

An example with structs

Using classes is a natural solution to this problem in other languages, and even on Apple platforms it comes easy because that's how Apple does it across the board. However, it's not a *great* solution because subclassing immediately gives us tight coupling.

Also, what would happen if you wanted more flexibility? You just saw how we made a specialized compiler that could (in theory) read Pascal source code, but how would you then create a Pascal compiler that output ARM64 code suitable for iPhone? That would mean having another subclass, this time replacing step 5, but think about the implications: you end up with **PascalARM32**, **PascalARM64**, **PascalX86**, **PascalX8664**, **CSharpARM**, **CSharpX86**, and… well, you get the point.

This is guaranteed to result in a mess of duplication, so you'll be pleased to know that Swift lets us get rid of subclassing in one fell swoop thanks to protocol extensions.

First, we need a protocol that defines the six methods a complete compiler must implement:

```
protocol Compiler {
    func analyze()
    func parse()
    func generate()
    func optimize()
    func output()
    func compile(filename: String)
}
```

We can then create a protocol extension that implements default functionality for all those methods:

```
extension Compiler {
```

```
    func analyze() { }
    func parse() { }
    func generate() { }
    func optimize() { }
    func output() { }

    func compile(filename: String) {
        analyze()
        parse()
        generate()
        optimize()
        output()
    }
}
```

At this point we effectively have the same result as the original **Compiler** class, except now it's a protocol.

The next step is to create a protocol that implements just the part responsible for the first two steps for Pascal:

```
protocol PascalFrontEnd: Compiler { }

extension PascalFrontEnd {
    func analyze() {
        print("Analysing Pascal")
    }

    func parse() {
        print("Parsing Pascal")
    }
}
```

Now we can add a third protocol that implements the final step for the ARM64 architecture:

```
protocol ARM64BackEnd: Compiler { }

extension ARM64BackEnd {
    func output() {
        print("Writing ARM code")
    }
}
```

Finally it's time to create a concrete type: a **PascalForARM** struct that conforms to both **PascalFrontEnd** and **ARM64BackEnd**:

```
struct PascalForARM64: PascalFrontEnd, ARM64BackEnd { }
```

Look at that – our struct *needs no code* because it's just putting protocols together like Lego bricks. Now we can instantiate that struct and run it with some code:

```
let compiler = PascalForARM64()
compiler.compile(filename: "myfile.pas")
```

If this were real code, you'd go ahead and create **ARM32BackEnd**, **CSharpFrontEnd**, and so on: one front-end for each language, and one back-end for each CPU architecture. All possible combinations could then be assembled using no extra code, just like **PascalForARM64**.

Summary

The template method pattern lets users replace one or more parts of your algorithm or process with their own implementation, allowing them to make a useful customization with a small amount of work.

Here are my suggested guidelines:

- Break down your process or algorithm into fine-grained methods otherwise it becomes a

burden to replace one small part.

- Your class or protocol should work normally even if none of its default implementations are replaced. It might not do anything terribly interesting, but it should at least *work*.
- If you take the subclassing approach remember that this immediately involves tight coupling. It's a great pattern, and it's common used in Apple development, but there are lots of times it's not the best solution.
- If you take the protocol-oriented programming approach remember that your code won't be available in Objective-C, which rules it out for many mixed-language projects.
- If the reason you're relying on the template method pattern is so that users can override methods, consider using delegation instead.
- Make sure you document your code clearly so that users know when they can, should, must, and must not call your default implementations.

Enumeration

The enumeration pattern lets us traverse a collection in a standard way, regardless of the internal data structure. So, whether you're working with an array, a set, a dictionary, or your own custom data type, enumeration allow you to traverse them all without having to care about their underlying representation. Note: the Gang of Four call this the iterator pattern, but Apple uses the term "enumeration".

Enumeration is baked into all Swift's collections, so we take for granted that this kind of code works out of the box:

```
let names = ["Taylor", "Justin", "Adele"]
for name in names { print(name) }

let pythons = Set(["Graham", "John", "Terry", "Eric",
"Michael"])
for python in pythons { print(python) }

let movies = [4: "A New Hope", 5: "The Empire Strikes Back", 6:
"Return of the Jedi"]
for movie in movies { print(movie) }
```

But that isn't *magic* – that behavior is enabled by the **IteratorProtocol** and **Sequence** protocols, and you can adopt them in your own types. **IteratorProtocol** has a single required method called **next()** that should return the next item in your sequence, and **Sequence** has a single required method called **makeIterator()** that makes an iterator to go over the sequence.

When you use a **for-in** loop, Swift maps that code to a **while** loop that calls **makeIterator()** to generate an iterator for your type, then calls **next()** on that iterator repeatedly until it gets back **nil** to signal that the loop has ended.

To demonstrate how this is done, we're going to walk through two examples: iterating finite sequences using a linked list, and iterating an infinite sequence using the Fibonacci sequence.

I'll implement them both differently, so you can decide which approach works best for your app.

Iterating finite sequences

Swift doesn't have a built-in linked list type, but they are trivial to construct. We need two classes: one called **LinkedListNode** that stores a value along with a reference to the next node, and one called **LinkedList** that holds a reference to the first node in the list.

These must be classes, because **LinkedListNode** contains a property of the same type as itself, which would lead to infinitely sized structs – something Swift won't allow. To make things more interesting, we'll make both these types generic so we can make lists of integers, strings, arrays, or whatever.

Create a new playground and give it these two types:

```
class LinkedList<T> {
    var start: LinkedListNode<T>?
}

class LinkedListNode<T> {
    var value: T
    var next: LinkedListNode?

    init(value: T) {
        self.value = value
    }
}
```

Using those two we can create a simple linked list of integers like this:

```
let list = LinkedList<Int>()
let first = LinkedListNode(value: 1)
let second = LinkedListNode(value: 1)
```

```
let third = LinkedListNode(value: 2)
let fourth = LinkedListNode(value: 3)
let fifth = LinkedListNode(value: 5)
let sixth = LinkedListNode(value: 8)

list.start = first
first.next = second
second.next = third
third.next = fourth
fourth.next = fifth
fifth.next = sixth
```

What we want to be able to do is loop over that linked list using a **for-in** loop like this:

```
for item in list {
    print(item.value)
}
```

However, that won't work because **LinkedList** doesn't conform to **Sequence** – Swift doesn't understand how to go from node to node in a loop.

To fix this we need to create a new class called **LinkedListIterator**. You need to give your iterator all the information it needs to process all the items in your sequence – in many cases that will be a copy of your data, but for a linked list all it needs is a property to store the first node. Once it has the first node it can get the second, third, fourth, and so on, just by moving through the chain.

The **LinkedListIterator** class must conform to **IteratorProtocol**, which means implementing its sole required method: **next()**. This will be called each time the calling loop goes around, so its job is to move down the list one place and return the next node. That might be **nil** – and indeed will always end in **nil** unless you have a loop in your linked list – so this needs to return an *optional* node.

Add this class to your playground now:

```
class LinkedListIterator<T>: IteratorProtocol {
    // holds the item we're currently reading
    private var current: LinkedListNode<T>?

    // an initializer giving us the start of the list
    init(start: LinkedListNode<T>?) {
        current = start
    }

    // move one place along and return that node
    func next() -> LinkedListNode<T>? {
        current = current?.next
        return current
    }
}
```

The **for-in** loop still doesn't work, because there's one more step: **LinkedList** needs to conform to **Sequence**. This means adding a small method called **makeIterator()**, which creates a new instance of **LinkedListIterator** using the starting node of our list.

Add this extension now:

```
extension LinkedList: Sequence {
    func makeIterator() -> LinkedListIterator<T> {
        return LinkedListIterator(start: start)
    }
}
```

You can learn more about organizing protocols using extensions in the Extensions chapter.

With that in place the **for-in** loop will finally work: Swift sees that **LinkedList** is a sequence,

generates an iterator, then loops over it until the end of the list is reached. In essence, this code:

```
for item in list {
    print(item.value)
}
```

Effectively becomes this:

```
var iterator = list.makeIterator()

while let node = iterator.next() {
    print(node.value)
}
```

Now that the code finally runs, you'll notice a small bug. This bug was intentional, honest – I included it here because it's something you'll hit with your own iterators, guaranteed.

We made our linked list using the first six digits of the Fibonacci sequence: 1, 1, 2, 3, 5, and 8. However, the initial 1 is missed off, so what's printed is 1, 2, 3, 5, 8. This happens because the very first time **next()** is called we run **current = current?.next** – we move one past the initial item, and return it. As a result, there's never a chance for the first node in the list to be returned.

Fortunately the fix for this is trivial thanks to Swift's **defer** statement. This doesn't get much use (a real shame!), but it's perfect here because it lets us run work after a method has returned. So, all we need to do is change **next()** to this:

```
func next() -> LinkedListNode<T>? {
    defer { current = current?.next }
    return current
}
```

That will cause Swift to move one item down the sequence only after returning the current

value, which means our Fibonacci sequence will be printed correctly.

Iterating infinite sequences

The Fibonacci sequence is an infinite sequence, so storing it in a linked list is overkill. Instead, we can create a **FibonacciGenerator** struct that conforms to **Sequence** and continually outputs new numbers in the sequence every time you move further down its list. All it needs to do is store the current number and the previous number, and it can generate as many numbers in the sequence as you need.

Previously we used two different classes working together to implement linked list iteration, but this time we're going to write just one struct that acts as a sequence and an iterator at the same time. When one type conforms to both **Sequence** and **Iterator** you no longer need to implement the **makeIterator()** method.

First we need a struct that will store the previous and current values for the sequence. These will be 0 and 1 by default, which is the start of the Fibonacci sequence.

Comment out the code you currently have in your playground, and replace it with this:

```
struct FibonacciGenerator {
    var previous = 0
    var current = 1
}
```

Next we're going to write an extension to that type that makes it conform to both **Sequence** and **IteratorProtocol**. Because the two protocols are combined together we need only to implement the **next()** method from **IteratorProtocol**, which will:

1. Calculate the next value as being the sum of the previous and current values.
2. Make the previous value the current value.
3. Make the current value the next value.
4. Return the current value.

As you saw with the linked list example, steps 1 to 3 should be done inside a **defer** block so they happen only after returning the current value.

Add this extension to your playground now:

```
extension FibonacciGenerator: Sequence, IteratorProtocol {
    mutating func next() -> Int? {
        defer {
            let next = previous + current
            previous = current
            current = next
        }

        return current
    }
}
```

Now that we have an infinite sequence, we can't call it using a regular **for-in** loop. Try something like this instead:

```
var count = 0

for fib in FibonacciGenerator() {
    print(fib)
    count += 1

    if count == 25 { break }
}
```

You can also step through it by hand if you want, just by calling **next()** as often as you need:

```
var fib = FibonacciGenerator()
fib.next()
```

```
fib.next()
fib.next()
fib.next()
fib.next()
fib.next()
fib.next()
```

Summary

The enumeration pattern allows us to sequentially access objects inside a collection without worrying whether it's an array, a set, or a dictionary. Even better, Swift exposes this functionality to us as **IteratorProtocol** and **Sequence** so we can generate our own enumerable types such as a Fibonacci generator.

Here are my suggested guidelines:

- If your type forms any sort of sequence that you want to iterate through, making it conform to **Sequence** is the best way to loop over it so you can use built-in language syntax like **for-in** loops.
- For simple types you can build your iterator right into your type itself, like we did with the Fibonacci sequence generator. It's probably still best to implement it as an extension, if only for organizational purposes.
- Separating your iterator type is useful for times when you might have several different kinds of iterators that work on the same data.
- Your iterator should always return its first item when **next()** is initially called (or during a **for-in** loop). I know it sounds obvious, but it's an easy mistake to make. Add some extra logic or use **defer** to make that happen.
- Finite or infinite sequences both work great with **Sequence**, so you could even write a sequence that generates random numbers continuously inside a loop. There's no naming convention here, but using something like **Generator** for an infinite sequence helps other developers know it's infinite.

Prototypes

The prototype pattern allows you to register an object or type, then use that as a reference to create future objects. The Gang of Four book gives two main advantages to this pattern: you delay your decision about what objects to create until runtime, and when you actually do create the objects the process may be faster because cloning an existing object is faster than running all your configuration code from scratch.

React Native is a JavaScript wrapper around UIKit and Android user interface code, and it allows web developers to create somewhat native apps in a language they know. I remember when it launched, the React Native developers laughed about how iOS developers needed to re-use table view cells otherwise their code would scroll slowly, whereas React Native's table control didn't need that. About four years later, they announced they were deprecating their own table control and replacing it with a new one that – you guessed it – employed cell re-use to make performance better.

The architecture of **UITableView** isn't there by accident – Apple weren't making a mistake when they designed it the way they did. Instead, it's because creating new objects is *hard*: allocating RAM for everything is slow, configuring all the objects as you need them is even slower, and drawing it all to the screen is the slowest of all.

As a result, **UITableView** makes use of the prototype pattern: you design cell prototypes inside Interface Builder, and rely on the system to create and re-use those intelligently for maximum performance. Even today, on the latest and greatest devices we have, cell re-use in **UITableView** and **UICollectionView** is essential if you want maximum performance.

Two things lie at the core of the prototype pattern:

1. Its ability to separate what you want to create from how it's actually created. In the case of table views, that means we attach a table view cell to a string identifier, then can create new cells using just the identifier.
2. Taking a copy of a prototype object is the equivalent of saying "give me one like this," rather than running all the code to define what "like this" actually means.

Re-using table view cells

We're going to walk through an example of building the prototype pattern in your own code, but first I want to just briefly go over the mechanics of **UITableViewCell** re-use – it's the closest example of the prototype pattern on Apple platforms.

Table view prototyping is done in two different ways: either you use Interface Builder or you use code. In Interface Builder you select your table, set its Content mode to Dynamic Prototypes, then add as many prototype cells as you need. You can then go ahead and design them fully, including creating Auto Layout constraints and attaching outlets to a custom class.

The alternative is to use code, which uses code like this:

```
tableView.register(UITableViewCell.self,
forCellReuseIdentifier: "Cell")
```

That registers a default table view cell – no subtitle or detail description – for the identifier "Cell". The process is identical when using collection views.

Now that there's a re-use identifier registered, we create instances of the table view cell inside the **cellForRowAt** method like this:

```
let cell = tableView.dequeueReusableCell(withIdentifier:
"Cell", for: indexPath)
```

That single line does two things in one:

1. If there is a cell waiting in the re-use queue, it will be dequeued and returned ready for configuration.
2. If there is no cell available – as will be the case for all initial cells – it takes a copy of the prototype you designed and returns that.

Bringing those two operations together means that developers are less likely to make a mistake

and create more table cells than are needed, but it also provides loose coupling: the table view doesn't have to know which design was selected for layout by its controller – you could register your own **UITableViewCell** subclass for the "Cell" identifier, and as long as the outlets remained the same you'd be OK.

Building our own table cache

The prototype pattern is useful in any situation where taking a copy of an object is easier and/or faster than creating it from scratch – a bit like a cooking show where they say "here's one I made earlier" rather than make you wait 45 minutes for something to cook in the oven. When blended with Apple's object re-use system, you get even more speed because object cloning happens only when you can't re-use an existing instance.

I'm going to walk through an implementation of a **TableView** type that works similar to Apple's **UITableView**: you can register cell types with it as string identifiers, then dequeue and enqueue cells as needed.

First we need a protocol that defines what it means to be a table view cell. In theory you'd have lots of setup work here, but we're just going to require three things: cells need an identifier so know their type, an initializer using that identifier, and a **clone()** method that returns an instance of themselves.

Create a new playground and add this protocol:

```
protocol TableViewCell {
    var identifier: String { get set }

    init(identifier: String)
    func clone() -> Self
}
```

Next we need a concrete type that conforms to the protocol. We don't need to worry about the initializer because that matches the memberwise initializer Swift will create for us. Add this

type now:

```
struct ExampleCell: TableViewCell {
    var identifier: String

    func clone() -> ExampleCell {
        return ExampleCell(identifier: identifier)
    }
}
```

Now for the challenging part: we need to create a **TableView** struct that does the business of registering, enqueueing, and dequeuing, cells. This needs to be able to work with anything that conforms to **TableViewCell**.

We're going to build this in four steps so you get a better idea of how it fits together. The first step is the easiest part: we're going to define the type and give it two properties: **identifierCache** will be a dictionary storing an identifier string as key and a prototype instance as its value, and **cellCache** will be a dictionary storing an identifier string as key and an array of instances as its value.

These two work together to give us enqueuing and dequeueing support: when we first register a type we create precisely one instance of it as a prototype and place that into the **identifierCache** property. When later we ask for instances of that identifier, we can pull out that prototype instance, call **clone()** on it, and send that back.

Start by adding this new struct to your playground:

```
struct TableView {
    var identifierCache = [String: TableViewCell]()
    var cellCache = [String: [TableViewCell]]()
}
```

The second step is to write a method for **TableView** that lets us register cell types with a string

identifier. This gives us the same loose coupling that **UITableView** benefits from – the controller can register whatever type of cell it needs without the data source having to care.

So, we're going to write a **register()** method that accepts a **TableViewCell** type to register and an identifier to use for it. It will then create precisely one instance of that cell from scratch as the prototype, from which all others of the same identifier will be created.

Now, what we *want* to receive is the *type of thing that should be registered*, not an instance of that type. Look again at the code used with Apple's implementation:

```
tableView.register(UITableViewCell.self,
forCellReuseIdentifier: "Cell")
```

You don't pass in an instance of the type, but instead a reference to the type itself. This is beneficial because it allows you to change your mind later – rather than creating and storing a prototype instance you could just create them from scratch each time.

We're going to adopt the same approach here, which means accepting **TableViewCell.Type** as our type parameter (a reference to the type) rather than **TableViewCell** (an instance of the type). Inside the method we're going to create one instance of the type that got passed in so we have our prototype, but we don't need to create anything in the **cellCache** property – we'll do that lazily.

Add this method to the **TableView** type now:

```
mutating func register(_ type: TableViewCell.Type, for
identifier: String) {
    identifierCache[identifier] = type.init(identifier:
identifier)
}
```

The third step is to write an **enqueue()** method that takes **TableViewCell** instances that are no longer needed and puts them into the **cellCache** property. This only takes one line of code, but I want to focus on it just briefly because it packs in a lot:

```
cellCache[cell.identifier, default: []].append(cell)
```

When you use **cellCache[cell.identifier]** to read the array cache for a particular, that cache might not exist – we didn't create one in the **register()** method, if you remember. So, Swift will return an optional for **cellCache[cell.identifier]**, which means we need to add more logic to create an empty array and add our cell there.

Fortunately, Swift's dictionaries give us this more advanced subscript method: you can request a key and provide a default value for when the key doesn't exist. So, that single line of code means "fetch the current cache for this identifier or create it if it doesn't exist, then append this cell to it."

Go ahead and add this method to **TableView**:

```
mutating func enqueue(_ cell: TableViewCell) {
    print("Enqueing cell with identifier \(cell.identifier)")
    cellCache[cell.identifier, default: []].append(cell)
}
```

Note: I snuck in a **print()** call there to help you visualize the flow of the code we're writing.

The final step is to create a **dequeue()** method that does all the hard work:

1. If we have a cell for this identifier in our cache, return it.
2. If we have a prototype for this identifier, clone it.
3. If both of those things aren't true, throw an error.

Again, I've added some printing code to help you follow that through. Add this final method to **TableView** now:

```
mutating func dequeue(identifier: String) -> TableViewCell {
    // attempt to find a cached cell for this identifier
    if let cell = cellCache[identifier]?.popLast() {
```

```
            print("Dequeuing cell with identifier \(identifier)")
            return cell
        } else {
            // attempt to find the prototype for this identifier
            if let cellType = identifierCache[identifier] {
                print("Creating new cell with identifier \
(identifier)")
                return cellType.clone()
            } else {
                // no prototype – error!
                fatalError("No worker registered for \(identifier)")
            }
        }
}
```

You can now go ahead and create an instance of **TableView** then register, dequeue, and enqueue freely. Try adding this code to the end of your playground:

```
var tableView = TableView()
tableView.register(ExampleCell.self, for: "Default")

let first = tableView.dequeue(identifier: "Default")
let second = tableView.dequeue(identifier: "Default")
tableView.enqueue(first)

let third = tableView.dequeue(identifier: "Default")
tableView.enqueue(second)
let fourth = tableView.dequeue(identifier: "Default")

let fifth = tableView.dequeue(identifier: "Unknown")
```

That last line attempts to dequeue a cell using an identifier that wasn't register, which will

throw an error.

Summary

The prototype method lets us create one object that acts as an original from which other instances of the same type are copied. By registering this prototype as a string we can allow our table view to create cells without specifying a hard-coded type, which makes for more flexible code.

Here are my suggested guidelines:

- If the cost of creating new objects is near to or the same as cloning them, this probably isn't the pattern for you.
- Apple's own terminology often uses "identifier", "reuse identifier", and **register()**. This isn't explicitly called out as a convention, but it would still seem wise to do the same in your own code.
- When users register their prototype, make sure you request a *type* rather than an *instance* so you can change your behavior later.
- In the **TableView** code example we threw a **fatalError()** if no prototype was found. This is perfectly reasonable and is exactly what **UITableView** does – don't try to be clever here, because it will just lead people to misuse your API.
- If you want to avoid stringly typed identifiers, have users extend a fixed type like **CellIdentifier** with their own constants – just like we did when registering custom notifications in the Notifications chapter.

Facades

The facade pattern is designed to provide a simplified interface to a complex system, allowing it to be used without the tight coupling that would exist if its internals were exposed. The Gang of Four say this has three major advantages, and for a change all of them apply to Swift. A good facade:

- Makes complex APIs easier to read, understand, and use, because their complexity is wrapped up in simpler methods.
- Reduces dependencies on the internals of those APIs, which allows you to change them freely as long as the end result does the same job.
- Lets you wrap a bad architecture in something that works better based on experience, or a complex architecture in something simpler that does the job for most people.

There are lots of examples of the facade pattern on Apple's platforms. In fact, the very structure of Apple's primary frameworks is based on precisely this pattern:

- The Darwin system powers the core of the operating system.
- On top of that are Core Foundation and Foundation.
- On top of those are Core Audio, Core Video, Core Text, Metal, OpenGL, and more.
- On top of those are AVFoundation, MetalKit, SceneKit, SpriteKit, Text Kit, and more.

Apple produces the low-level APIs as part of their Core framework line, but chooses some of them to be wrapped in a Kit facade. The facade never offers anything you can't do by hand in its Core equivalent, but does make it significantly simpler to do common tasks.

Even when you get to the Kit layer, Apple still creates facades to make our life easier. Most of UIKit's controls aren't too difficult – you could imagine creating **UIButton** from scratch if you needed to. But **UITextView**? That's some serious work: being able to render scrolling text across multiple lines, allowing users to edit it (including adding formatting), allowing users to select, copy, and paste, and so on – these are all things we just take for granted, and the existence of **UITextView** allows us to focus on its simple exposed API rather than touching Text Kit directly.

Another more subtle example of facades is **UIImage**: you can make it load JPEGs or PNGs from disk, then render them to the screen. However, you can also make it load vector artwork from PDF files. Vector art is resolution independent, which means **UIImage** will automatically adapt itself so that it draws at the correct resolution for whatever size you're using it, automatically taking into account the retina scale of the device.

Probably my favorite of all Apple facades is one that was used heavily before iOS 7 but since then has almost been forgotten: **UIColor**. Yes, we all use it as a regular type today, but it's actually a facade and was commonly used as such before iOS 7, when many people used its static property **scrollViewTexturedBackgroundColor**. You can still see it in Interface Builder's color selection dropdown, even though the property itself has always been unavailable in Swift.

Before iOS 7 using this property rendered a sort of scratched metal texture rather than a solid color, and was used behind scrolling views to show that you had zoomed out or scrolled too far. Of course Apple didn't want to complicate their API by adding an extra **backgroundImage** property to every view so instead they made **UIColor** a facade that could store images or colors depending on what was needed.

Even though the scroll view texture property isn't available to us, **UIColor** still remains capable of being used as an image using its **patternImage** initializer. For example, if you had a small checkered image that you wanted to tile as the background in a view, you can do it using nothing more than **UIColor**:

```
if let checkers = UIImage(named: "checkers") {
    view.backgroundColor = UIColor(patternImage: checkers)
}
```

Convenience initializers

Rather than going with a dedicated facade, sometimes the best way to wrap complex functionality is to use convenience initializers. Remember, the goal of a facade is to make API

easier to use or to limit or remove the ability to dabble with the internals. In the case of convenience initializers you can accomplish the first of those two easily.

A good example of this is the **NSTextField** component on macOS. If you've written UIKit code before you'll be familiar with using **UILabel** for static text and **UITextField** for editable text, but that distinction doesn't exist on macOS – **NSTextField** handles both responsibilities. This is useful because many kinds of labels can instantly become editable in macOS, for example Finder lets you rename directories by clicking on their label.

macOS 10.12 (Sierra) introduced some convenience initializers for **NSTextField** that made it significantly easier to use and also hid all sort of internals developers just didn't care about. Before macOS Sierra this kind of code was normal:

```
let label = NSTextField(frame: .zero)
label.textColor = NSColor.labelColor()
label.font = NSFont.systemFontOfSize(0)
label.bezeled = false
label.bordered = false
label.drawsBackground = false
label.editable = false
label.selectable = false
label.stringValue = "Hello, world!"
```

And after we just wrote this:

```
let label = NSTextField(labelWithString: "Hello, world!")
```

All the same properties remain exposed if you want to manipulate them directly, but the new convenience initializer takes away almost all the work.

Wrapping C libraries

A common example of the facade pattern is wrapping C libraries for use in Swift. Swift is able

to call C functions directly, but doing so is often unpleasant – they don't work well with native types we use, they often require you to work in a precise and detailed way, and above all they require manual memory management.

As a result, C libraries are a great candidate for the facade pattern: let Swift wrap all the C function calls, then write a layer on top to mask the internals.

As an example, I created a server-side Swift framework called SwiftGD. It's responsible for doing image drawing on servers, where Core Graphics isn't available and GPUImage isn't possible. SwiftGD wraps a C library called GD, but it doesn't just then expose those functions. Instead, it marks the GD internals as being private to the framework, and exposes a simplified wrapper to Swift users that works in a more streamlined way and that handles memory management automatically.

Here's a tiny slice of code so you can see how that looks. I've added some extra comments to provide some context:

```
// the exposed class that others use
public class Image {
   // the hidden, private image representation
   private var internalImage: gdImagePtr

   // a computed property that converts the GD size
representation to a simpler Size struct
   public var size: Size {
      return Size(width: internalImage.pointee.sx, height:
internalImage.pointee.sy)
   }

   // a public initializer that lets users create blank images
of a specific size
   public init?(width: Int, height: Int) {
      internalImage = gdImageCreateTrueColor(Int32(width),
```

```
Int32(height))
    }

    // a private initializer that creates a new Image from
existing GD data, used when resizing and resampling
    private init(gdImage: gdImagePtr) {
        self.internalImage = gdImage
    }
}
```

If you'd like to see more of how SwiftGD is implemented, it's available online under the MIT license: **https://github.com/twostraws/SwiftGD**.

Creating a facade type

It shouldn't surprise you that I'm not going to demonstrate how to rebuild **UITextView** from scratch, but we can look at a simpler example: how to recreate the ability of **UIColor** to hold colors or images, without users knowing which is which when they are being used. In the case of **UIColor** this means you can assign a color to a view's background image regardless of whether it's a color or an image – you don't need to care.

We can model this in a playground using a single **ImageOrColor** struct:

```
struct ImageOrColor {
    private var color: UIColor?
    private var image: UIImage?
}
```

Both of those properties are marked private because we don't want folks digging around in there.

That struct needs two initializers based on what it should do internally, so add this to it:

```
init(color: UIColor) {
```

```
    self.color = color
}

init(image: UIImage) {
    self.image = image
}
```

Finally we're going to give it a method that lets the struct draw itself into a rendering context regardless of whether it contains an image or a color:

```
func fill(rect: CGRect, in ctx: UIGraphicsImageRendererContext)
{
    if let image = image {
        image.draw(in: rect)
    } else if let color = color {
        color.setFill()
        ctx.fill(rect)
    }
}
```

We can now create instances of that struct, then use them freely without worrying how it works internally – it will do the right thing depending on how it was created:

```
let rect = CGRect(x: 0, y: 0, width: 512, height: 512)
let orangeColor = ImageOrColor(color: .red)
let renderer = UIGraphicsImageRenderer(bounds: rect)

let final = renderer.image { ctx in
    orangeColor.fill(rect: rect, in: ctx)
}

final
```

When that code runs, you should be able to preview the **final** line at the end to see the red fill.

Because this particular example is simple it's closer to the abstract factory pattern – something that's rarely seen in Swift. With abstract factories you create one class, which internally creates one of several other private subclasses when it's instantiated. Sometimes you might see a Swift error referencing something like **NSContiguousString** – that's a subclass of the **NSString** abstract factory.

Keep in mind that UIKit is old now: it's been around since before the iPhone launched, although it was private until 2008. If I were implementing this functionality from scratch today I'd start with a **Drawable** protocol and make my image and color classes conform to that rather than squeeze images into a color class.

Summary

The facade pattern lets us wrap complex or unpleasant work in a smarter, simpler interface. In doing so it lets developers use our API more easily, but also means the specifics of our implementation are hidden and so can be refactored at will in the future.

Here are my suggested guidelines:

- Facades are useful any time you must work with an API in the same highly specific way each time – wrap that work up in a facade to make it easier to use.
- If your goal is to hide internals, make sure you mark them all private. Core ML exposes a single **prediction()** method for its vast machine learning system!
- Convenience initializers provide a simple alternative to full facades, allowing you to provide sensible defaults for common situations.
- If you want to build something that wraps multiple types, like **UIColor**, consider whether a protocol is what you really want.
- How much you expose of your internal API is down to you: expose too much and your facade might be useless, but expose too little and your facade is also useless! At least if you expose too *little* you can add more without breaking compatibility.

Decorator

The decorator pattern lets you add functionality to an object without modifying its code. In the pure sense that it's defined by the Gang of Four the decorator pattern is about adding or removing responsibilities to a specific object (as opposed to its entire type) at runtime, but in Apple development we tend to prefer using a mix of delegation and extensions.

Of the two, delegation is certainly the closest match. The goal of decorators is to let us customize the way an object works without needing to subclass it, and that's precisely what we use delegation for. In a strict implementation of the decorator pattern you would create one class that wraps another, and you would make the wrapper object have the same interface as the wrapped object so that it can transparently forward functionality to the wrapped type while adding its own functionality around it.

In delegation the opposite happens: no wrapping happens, and there's no shared interface. Instead, the delegate object is the one that customizes the object around it, so in some respect it's like the decorator pattern in reverse.

I've already covered delegation and extensions in their own chapters, so here I want to focus on a third way the pattern is deployed on Apple platforms: object composition.

Using and re-using table views

Table views are fundamental controls on iOS, macOS, and tvOS – they are used in so many different ways and with so many different styles, but always work in the same basic way.

When they came to design the **UIDatePicker** control for iOS, Apple could have gone back to a blank **UIView** subclass then written all the code from scratch. But they didn't: they decided to embed several instances of **UITableView**, then apply transformation effects to give them the appearance of wheels rolling around on a barrel. If you drop a date picker onto a storyboard then use Apple's view debugger, you'll see exactly what I mean – 12 different table views layered over each other produce the effect. Of course, no developer wants to deal with that mess, so Apple composes all that together into a single **UIDatePicker** class.

UITableView itself is based on an even more ubiquitous control: **UIScrollView**. However, this is where things get a little more interesting, because the **UIScrollView** class is used as a parent class in some places (**UITableView**, **UICollectionView**, **UITextView**) and as a composed property in others (**UIWebView** and **WKWebView**).

This isn't Apple varying from its strict adherence to design patterns, but instead it's a pragmatic choice: in table views, collection views, and text views the scroll view *is* the content you're viewing, whereas in the two web views the scroll view is really just a display mechanism for the content inside. This is usually contrasted as "is-a" vs "has-a" to help clarify whether inheritance or composition makes most sense – a surgeon *is* a doctor and *has* a knife, for example. While composition is preferable to inheritance, don't get yourself into a twist over it.

Accessorizing views

One simple but useful demonstration of the decorator pattern is the use of accessory views. As I said at the beginning of this chapter, the decorator pattern lets you add functionality to an object without modifying its code, and that's what accessory views do: you can modify the way the standard components are presented at runtime by adding your own accessories on top.

On macOS this is primarily done using the **accessoryView** property, which lets you add an **NSView** inside things like an **NSColorPanel** or an **NSAlert** to make it more customized to your uses.

For example, you might show an alert asking a user to enter their name:

```
let username = NSTextField(string: "Anonymous")
let alert = NSAlert()
alert.messageText = "Please enter your desired username."
alert.accessoryView = username
alert.runModal()
print("You entered: \(username.stringValue)")
```

On iOS the equivalent is the **inputAccessoryView** property, which lets you create controls to attach to the on-screen keyboard. For example, Tweetbot uses this to attach a toolbar with buttons for inserting images, locations, and hashtags easily.

For example, you might add a gentle reminder to users as they write a message in your social media app:

```
let label = UILabel()
label.text = "Write something nice!"
label.sizeToFit()
```

Both of these allow us to modify the way another control looks and works without having to subclass.

Decorator in your own types

To demonstrate the power of the decorator pattern, we're going to build a simple **ScoreLabel** struct in a playground, suitable for use in a game. Scores can be formatted in any number of ways, so to represent that we're going to create a **ScoreAccessory** protocol that lets users of our API configure a prefix or suffix accessory for the score. Whether it's a prefix or a suffix, the same method will be run: "here's the score value – what text do you want to return?"

Start by creating a new playground and giving it this protocol:

```
protocol ScoreAccessory {
    func string(for score: Int) -> String
}
```

For the purpose of testing, we're going to define two concrete implementations of that protocol: a **TitleAccessory** struct that always returns the word "SCORE: " (note the space at the end!), and a **PointsAccessory** struct that returns either " POINT" or " POINTS" depending on how many points were passed in (note the space at the start!).

Add these two structs to your playground:

```
struct TitleAccessory: ScoreAccessory {
    func string(for score: Int) -> String {
        return "SCORE: "
    }
}

struct PointsAccessory: ScoreAccessory {
    func string(for score: Int) -> String {
        if score == 1 {
            return " POINT"
        } else {
            return " POINTS"
        }
    }
}
```

Now we can construct the main event: a **ScoreLabel** that tracks the player's current score and is able to rely on two **ScoreAccessory** properties to modify the way the score is printed.

The struct itself needs three properties: **prefix** and **suffix** will both be an optional instance of **ScoreAccessory**, and **score** will track the user's current score number. We're going to attach a **didSet** property observer to the score so that whenever it changes we call a **printScore()** method to print out the new value.

Add this struct now – don't worry about the error on **printScore()** because we'll add that next:

```
struct ScoreLabel {
    var prefix: ScoreAccessory?
    var suffix: ScoreAccessory?

    var score = 0 {
        didSet {
            printScore()
```

```
        }
    }
}
```

The last step is to write a **printScore()** method for **ScoreLabel**, which needs to ask both score formatters for their string given the current score, or use an empty string if either value isn't set. Finally, it will combine those two strings with the current score and print the result.

Add this method to the **ScoreLabel** struct now:

```
func printScore() {
    let start = prefix?.string(for: score) ?? ""
    let end = suffix?.string(for: score) ?? ""
    print("\(start)\(score)\(end)")
}
```

Providing default values is important – these are *accessories*, so your code should work even if they aren't present.

To give the whole thing a try, add this code to the end of your playground:

```
var scoreLabel = ScoreLabel()
scoreLabel.prefix = TitleAccessory()
scoreLabel.suffix = PointsAccessory()
scoreLabel.score += 10
```

That changes the player's score, which triggers the property observer, and in turn calls both accessories. The end result should be that you see "SCORE: 10 POINTS" printed in your playground output.

Summary

The decorator pattern lets us alter the behavior of an object without modifying its code. In

Apple development we normally use delegation and extensions, but adding accessories is also common.

Here are my suggested guidelines:

- If delegation or extensions are possible solutions, explore them first – they are a natural fit on Apple platforms.
- If the decorator pattern is what you want, then remember that the goal of the pattern is to avoid subclassing – if composition is possible you should use it.
- Allowing accessories in your own code is a simple way to let others take part in your process without overriding it.

Flyweight

The flyweight pattern allows us to use one object in many places to avoid excessive memory use. Sometimes that object might contain all the information it needs to be used by itself, but other times it contains only the data that is fixed – individual users need to provide the missing data that makes them unique.

Of all the patterns in this book, the flyweight pattern is the one I like the least. While conceptually it sounds good, it suffers from numerous problems:

1. Swift value types are always copy on write, which means you can duplicate them 100 times or 10,000 times without worrying about the performance implications.
2. Even if they weren't copy on write, most apps have hundreds of megabytes of RAM to work with at the absolute minimum, and the extra complexity to implement (and debug!) flyweight might cost you more than it saves.
3. Sharing objects can easily lead to code that's harder to debug and certainly harder to reason about.

There are three reasons why I'm including it here:

1. Apple uses it extensively, even in places where the efficiency savings must be microscopic.
2. Sometimes you might find the benefit of outweighs the cost, usually when you're sharing partial data.
3. This pattern is popular enough that if I *don't* include it you can bet I'll get complaints.

How does Apple use it?

The Flyweight pattern is useful when taking copies of data is expensive. As I said already this isn't usually a problem in Swift thanks to copy-on-write value types, but of course most of Apple's code is Objective-C where copying *is* a real problem.

A good example from UIKit is **UIFont** – try adding these four lines of code to a playground:

```
let font1 = UIFont.systemFont(ofSize: 32)
let font2 = UIFont.systemFont(ofSize: 32)
let font3 = UIFont.systemFont(ofSize: 32)
let font4 = UIFont.systemFont(ofSize: 32)
```

When you look in the playground results you'll see the same memory address listed for all four fonts even though they were created the same. UIKit is automatically caching the results each time, because one object containing the system font at 32 points is identical to all other objects containing the system font at 32 points.

On the one hand, this means you can check for equality using ===:

```
if font1 === font2 {
    print("Fonts are the same!")
}
```

On the other hand, that might not be what you expected – you created each of the fonts individually, so you would probably expect all four of them to point to different memory. In this case Apple has taken the correct precautions to avoid you breaking things by accident: all the properties of **UIFont** are read only, so you can't accidentally pollute the other instances.

You can imagine that creating fonts is expensive, and so the flyweight pattern here makes sense. However, Apple also uses the flyweight pattern for **NSNumber**: if you create multiple instances of **NSNumber** they all point to the same object.

For example, try this:

```
let number1 = NSNumber(value: 556)
let number2 = NSNumber(value: 556)

if number1 === number2 {
    print("Numbers match!")
}
```

Note: === checks for reference equality, i.e. the same memory address.

Back when **NSNumber** was first implemented having two numbers point to the same memory address might have been an important memory saving, but today computers have thousands of times more RAM so it's likely only still a feature for backwards compatibility reasons.

Flyweight in your own types

While you can apply the flyweight pattern to whole types like Apple does, it just seems wasteful when Swift's structs do most of the hard work for you. When you share *partial* data, however, the situation is different: if ten pieces of data don't change but one piece of data does, sharing the invariant data will help – *as long as you're careful to make it immutable*.

The case for the flyweight pattern becomes stronger when you use classes, where copy on write isn't used. With classes you either have to take a full, deep copy of all the data you want in each instance, or you carve off chunks into immutable instances that are then shared – each object using your flyweight points back to the same, shared reference.

To demonstrate flyweight in action, think of a car manufacturing plant that churns out cars of various models. A medium-sized plant can turn out about finished 2000 cars a day, but it will have many more at various stages of completion.

All the cars for a given model share many fixed characteristics: they all come from the manufacturer, they have the same model name, the same length and width, and the same safety rating. However, individual, finished cars vary: their registration plates differ, their engine sizes differ, and so on.

We can model this using a flyweight that stores partial data: a **CarBase** struct that stores all the invariant information, and a **Car** struct that stores all the varying information. The key thing to remember is that you should ensure all parts of the invariant information are fixed and immutable – don't let anyone change them, because it may cause it to be changed elsewhere.

Create a new playground and give it this struct:

```
struct CarBase {
    let manufacturer: String
    let model: String
    let length: Int
    let width: Int
    let safetyRating: Int
}
```

Every property there is marked as **let** so that they cannot be changed even if the struct itself is created using **var**.

Next we need a **Car** struct that stores a **CarBase** to track its invariant information, then add its own custom settings on top:

```
struct Car {
    let base: CarBase
    var registration: String
    var color: UIColor
}
```

Finally, we can create an instance of **CarBase** and assign it to any number of **Car** instances:

```
let jaguarXJ220 = CarBase(manufacturer: "Jaguar", model:
"XJ220", length: 4930, width: 2009, safetyRating: 4)
var jag1 = Car(base: jaguarXJ220, registration: "X118 CJM",
color: .black)
var jag2 = Car(base: jaguarXJ220, registration: "X119 CJM",
color: .red)
```

That means all cars created from the same XJ220 line used a shared data struct, rather than storing their own copy.

Summary

The flyweight pattern lets us share some or all of an object's data, reducing memory use when many copies might otherwise have existed.

Here are my suggested guidelines:

- As computers increase in power, flyweights decrease in usefulness. Is the extra complexity of adding them really worth their advantages?
- If you're using Swift structs you're probably benefiting from copy on write already, so the argument for flyweights is even harder.
- If you're using classes and sharing partial data, flyweight remains useful. That doesn't mean you *should* use it, but it's certainly worth considering.

Chapter 6
Wrap Up

The Best Laid Plans

I've tried to make this book a blend of things: sometimes it's explained how things work, sometimes it's explained *why* they work like that, and sometimes I've been pretty explicit when a certain pattern is a bad idea.

Designing a large app is complicated. In fact, I'd go so far as to say that designing a large app is a terrible idea. Linus Torvalds – the creator of the Linux operating system – had this to say about large projects:

Nobody should start to undertake a large project. You start with a small trivial project, and you should never expect it to get large. If you do, you'll just overdesign and generally think it is more important than it likely is at that stage. Or worse, you might be scared away by the sheer size of the work you envision.

A few times this book I've added little snippets of my own experience developing large software. Sure, they have happy outcomes, but I included them because it's important you understand that even with the best of designs – even with a team who love design patterns and have whole whiteboards full of architecture diagrams – you will still often face crunch periods where you're under pressure to deliver something.

These times are usually rare – at least in any sensible environment – but they will always crop up sooner or later, and mean you don't get to have the time to plan, test, and put proper thought into your code.

That's OK. Really, don't beat yourself up over it.

Like I said at the beginning of this book, great code is thoughtful code, and obviously I'd love for you to be at a company where you have heaps of time to consider and plan. But if you make mistakes – and you will – or when you find you don't have time to consider all the options fully, please remember that you're doing your best, that most other developers are in exactly the same boat, and the only codebase with no technical debt is "Hello, world."

Guidelines recap

At the end of most chapters I've tried to include some guidelines to help you understand and apply each pattern more easily. After you've read the book once, you might find you want to refer directly to the guidelines as a reminder of what was said in the rest of the chapter, so I've included them here to make printing easier.

Delegation

- It isn't *required* that you use protocols with delegation, but it is certainly a good idea to reduce your coupling.
- Always follow Apple's naming conventions for delegates: use a **delegate** property if there is only one, or put "delegate" at the end if there are more than one.
- Apple also has naming conventions for delegate methods, so you should pass in the object that triggered the event as the first parameter.
- Your types should function even without a delegate set, but you're OK to make them throw errors without a data source if you want.
- If your delegate doesn't need to implement a method, either use **@objc** and mark it optional or provide a default implementation. The latter is nearly always preferable.
- Trying to make one type act as data source or delegate to multiple things is problematic in anything larger than a toy app – try to separate them if possible.

Selectors

- Although selectors are an inevitable feature of any AppKit or UIKit app, that doesn't mean you need to add them in your own types – closures are preferable.
- Sending actions to nil is a smart and simple way to rely on the responder chain to execute an action. This is covered more detail in the Responder Chain chapter.
- Performing selectors with a zero-second delay is one of those things you'll think you never need, but will almost certainly come in useful thanks to various UI quirks – it's a good skill to have in your toolbox.
- If you can think of a legitimate reason to use **NSSelectorFromString()** in new Swift code,

I'd love to hear it.

Notifications

- Plan carefully how granular your notifications should be.
- Name your notifications with a unique prefix so they don't accidentally clash with any libraries you're using.
- Following Apple's own naming conventions, using **will** and **did** appropriately.
- Remember that notifications are synchronous by default, although an asynchronous alternative exists.
- If you end up with lots of notifications, consider adding an object filter. What it means is down to you, but it will help simplify things.
- Delegation is often the better solution if you're notifying only one thing: it is simpler to follow in code, and completely type safe.
- Don't imagine for a moment that it's pleasant writing tests that rely on **NotificationCenter**, because it isn't.

Associative Storage

- If you're exposing an Objective-C **NSDictionary** to Swift add a typedef for your key type so that it's more specific than just **NSString**. For example, Apple uses **typedef NSString *UIApplication.LaunchOptionsKey** for app launch options – it won't affect Objective-C developers but helps in Swift.
- Add a **userInfo** property to data types that need flexible storage. Make it optional if it won't always be used.
- Use extensions to add computed property to data types, allowing access to stored data without forcing users to deal with dictionary access. Make sure your access key is unique.

Archiving

- If you need to work alongside Objective-C code, either yours or Apple's, you probably

need to use **NSCoding**.

- When writing your **NSCoding** key names, use constants rather than freely typed strings to avoid typos.
- If **Codable** is possible in your code, it's almost certainly a better solution – even if you need to wrap some **NSCoding** items.
- Try to rely on the built-in encoding/decoding ability of **Codable** before you write your own. Yes, you can customize loading and saving but it just adds extra maintenance burden.
- Remember to use **CodingKeys** to ensure your archived names match the naming conventions of your target. That might be camelCase, in which case **CodingKeys** probably isn't required, but it might also be snake_case.
- Even if you're just loading and saving data locally, you should probably change your date strategy to ISO-8601 for sanity if nothing else.
- If you provide a custom closure for your date decoding strategy make sure it's as small as possible – it could be called a *lot*.
- While you *can* wrap **NSCoding** objects using **Codable**, keep in mind they take up a lot of space. If you're transferring data over the network you might want to look into a different solution.
- Regardless of whether you use **NSCoding** or **Codable**, both are implementations of the template method pattern covered later in the book.

Bundles

- It's rarely a good idea to dig around in your bundle directly. Use the methods on **Bundle.main** and your code is guaranteed to work even if Apple changes bundle layouts in the future.
- You can read values directly from your Info.plist using the **Bundle.main.object(forInfoDictionaryKey:)** method. For example, **CFBundleShortVersionString** contains your version number – just typecast it to a string.
- Your Info.plist can store custom values as well as Apple's own. It's just XML, so you could easily auto-generate this to include something like a server identifier.

Initialization

- Prefer structs rather than classes so that initialization becomes a non-issue.
- Where possible provide default values or make property optionals to reduce the need for initializers even further.
- Convenience initializers are there to make your life easier – use them!
- Failable initializers express failure really cleanly, and don't cause much of a speed bump thanks to **if let**.
- Don't use dynamic creation unless you really like living dangerously. Let Interface Builder carry on with it, but that doesn't mean it needs to taint your own code.

Extensions

- Use extensions to group functionality logically. Organizing things by protocol is a good start, but you might find purpose easier to begin with.
- Don't replace methods unless you're absolutely sure you know what you're doing. This goes doubly for anything in Apple's user interface libraries, AppKit and UIKit, where there are so many corner cases it's easy to miss something.
- Rather than polluting your namespace with functionality, be sure to lean heavily on Swift's expressive constraints system to limit where your extensions are applied.
- Refer to the associative storage pattern for examples of how extensions can simulate stored properties.

Protocols

- Name thing protocols as nouns, and ability protocols as adjectives: "able", "ible", and "ing".
- Use **will**, **did**, and **should** for protocols that will be used in delegates or data sources. Remember to pass in the object that triggered the event as the first parameter.
- Pure Swift protocols can use all the power and expressivity of Swift, but cannot have optional requirements.

- **@objc** protocols *can* have optional requirements, but cannot work with structs and can't use protocol extensions.
- Use class-only protocols when you want your conforming types to act as reference types, allowing you to use them as **weak**.
- Protocols are composable, which means there's no point trying to build one super-sized protocol with everything in. Instead, start small and compose them to make bigger things.
- Associated types are a powerful feature when you want them, but they can be annoyingly easy to stumble into by accident. If you take a few simple steps you can avoid them entirely.

Protocol Extensions

- When extending a protocol, extend the most specific protocol that solves your problem. There's no point extending **Numeric** (all numbers) if all you actually want is **BinaryInteger** (integers) – it may add a maintainability burden and certainly pollutes the namespace.
- Composing protocols is a great way to group things together and can aid readability, but it only really works when your initial protocols are suitably isolated. No one wants an all-singing, all-dancing super protocol that takes an age to conform to or that brings with it dozens of default method implementations.
- Constrained protocol extensions let you implement the same method in multiple different ways, which is the protocol equivalent to the Gang of Four's Composite pattern – multiple extensions working side by side under a single protocol.
- Don't try to fight the system. So much of Apple development relies on Objective-C-compatible code (i.e. classes and **@objc**), and protocol extensions just don't work well there.
- Your protocol extensions don't need to provide default implementations for all methods – it's OK to force types to implement some themselves.

Accessors

- Don't implement accessor methods until you need them; it just adds clutter to your code.
- Do give properties the most limited visibility you can. Going from **internal** to **public** won't break any code, but going in the other direction will.
- Always make backing store properties private. Even **internal** leaks too much information.
- Worry more about the interfaces you expose and less about the implementation behind them – you can change between stored properties, computed properties, and lazy properties at will, but a bad interface is bad for good.

Keypaths

- If you have a project that mixes Objective-C code and Swift code, make your instance variables KVO-compliant where possible.
- If you're writing your own Swift code, using **didSet** and **willSet** is much better than using KVO, not least because it avoids having to use **@objc dynamic** everywhere.
- Keypaths are a great way of identifying specific data inside your types. We used it for identification here, but it could be used to identify which property stores a type's display name, or indeed anything where property *names* might vary when their *meaning* remains the same.
- Don't use bindings on macOS. In the words of David Smith, one of Apple's Cocoa engineers, "Most large Mac apps I'm aware of don't use bindings. It's appealing, but difficult to debug when things go wrong."

Anonymous Types

- Try using **Any** everywhere you use **AnyObject** or a class that conforms to it. If it succeeds, you're better of for it.
- If you find yourself frequently creating **Any** collections, try creating a protocol that describes your types more precisely – you're giving both the compiler and other developers more information to work with, rather than trying to keep it all in your head.
- Even **Any** is open to abuse, and comes with its own costs. As useful as **Any** is, it ought to be your last resort.

- Although using **Any** for the sender in **IBAction** does allow loose coupling, I think the benefit is dubious at best and wouldn't stop anyone from changing it to a concrete type instead.

Singletons

- Think carefully whether your singleton is just a global variable in disguise. We all agree that global variables are bad, and a global variable masquerading as a singleton is no better.
- Name your shared instance using Apple's naming conventions: **shared**, **default**, and **current** are all common.
- If your goal is to have a true singleton – something that can exist no more than once – make sure you hide all initializers to external code.
- When you write methods for your singleton, remember they will be called from all parts of your app. If those methods rely on shared state, or if they take a long time to execute, you may encounter concurrency issues.
- If your singleton usage is a mere implementation detail, treat it like one and hide it behind a protocol extension.
- Keep in mind that singletons can become a problem – don't shy a way from removing them when they become a problem, even if they are important. As the saying goes, "don't cling to a mistake just because you spent a lot of time making it."

Responder Chain

- You can invoke the responder chain by posting actions to **nil** and letting the system find the first responder for you.
- The responder chain should end as soon as one object can handle the event fully. If it doesn't, this is more like the decorator pattern.
- If you're implementing your own responder chain, consider adding a boolean return that lets you check whether any object handled the action.
- No matter whether the chain ends with one object, each object handles part of a message, or each object handles all of a message, responder chains are only called in one direction.

While they can pass a value back, they shouldn't be *called* backwards.

Template Method

- Break down your process or algorithm into fine-grained methods otherwise it becomes a burden to replace one small part.
- Your class or protocol should work normally even if none of its default implementations are replaced. It might not do anything terribly interesting, but it should at least *work*.
- If you take the subclassing approach remember that this immediately involves tight coupling. It's a great pattern, and it's common used in Apple development, but there are lots of times it's not the best solution.
- If you take the protocol-oriented programming approach remember that your code won't be available in Objective-C, which rules it out for many mixed-language projects.
- If the reason you're relying on the template method pattern is so that users can override methods, consider using delegation instead.
- Make sure you document your code clearly so that users know when they can, should, must, and must not call your default implementations.

Enumeration

- If your type forms any sort of sequence that you want to iterate through, making it conform to **Sequence** is the best way to loop over it so you can use built-in language syntax like **for-in** loops.
- For simple types you can build your iterator right into your type itself, like we did with the Fibonacci sequence generator. It's probably still best to implement it as an extension, if only for organizational purposes.
- Separating your iterator type is useful for times when you might have several different kinds of iterators that work on the same data.
- Your iterator should always return its first item when **next()** is initially called (or during a **for-in** loop). I know it sounds obvious, but it's an easy mistake to make. Add some extra logic or use **defer** to make that happen.

- Finite or infinite sequences both work great with **Sequence**, so you could even write a sequence that generates random numbers continuously inside a loop. There's no naming convention here, but using something like **Generator** for an infinite sequence helps other developers know it's infinite.

Prototypes

- If the cost of creating new objects is near to or the same as cloning them, this probably isn't the pattern for you.
- Apple's own terminology often uses "identifier", "reuse identifier", and **register()**. This isn't explicitly called out as a convention, but it would still seem wise to do the same in your own code.
- When users register their prototype, make sure you request a *type* rather than an *instance* so you can change your behavior later.
- In the **TableView** code example we threw a **fatalError()** if no prototype was found. This is perfectly reasonable and is exactly what **UITableView** does – don't try to be clever here, because it will just lead people to misuse your API.
- If you want to avoid stringly typed identifiers, have users extend a fixed type like **CellIdentifier** with their own constants – just like we did when registering custom notifications in the Notifications chapter.

Facades

- Facades are useful any time you must work with an API in the same highly specific way each time – wrap that work up in a facade to make it easier to use.
- If your goal is to hide internals, make sure you mark them all private. Core ML exposes a single **prediction()** method for its vast machine learning system!
- Convenience initializers provide a simple alternative to full facades, allowing you to provide sensible defaults for common situations.
- If you want to build something that wraps multiple types, like **UIColor**, consider whether a protocol is what you really want.

- How much you expose of your internal API is down to you: expose too much and your facade might be useless, but expose too little and your facade is also useless! At least if you expose too *little* you can add more without breaking compatibility.

Decorator

- If delegation or extensions are possible solutions, explore them first – they are a natural fit on Apple platforms.
- If the decorator pattern is what you want, then remember that the goal of the pattern is to avoid subclassing – if composition is possible you should use it.
- Allowing accessories in your own code is a simple way to let others take part in your process without overriding it.

Flyweight

- As computers increase in power, flyweights decrease in usefulness. Is the extra complexity of adding them really worth their advantages?
- If you're using Swift structs you're probably benefiting from copy on write already, so the argument for flyweights is even harder.
- If you're using classes and sharing partial data, flyweight remains useful. That doesn't mean you *should* use it, but it's certainly worth considering.