PRO SWIFT

BOOK AND VIDEOS

Break out of beginner's Swift with this hands-on guide

Paul Hudson

Contents

Preface

Andy Matuschak (@andy_matuschak), lead mobile developer at Khan Academy

If you're transforming a collection, but you don't need to access all the elements right away (or at all), you may be able to save cycles and allocations by using the lazy family of collection types:

```
let userCellsData = users.lazy.map { user in
   UserCellData(username: user.username, bioAttributedString:
formatBioString(user.bio))
}
```

Welcome

This is not a book for people who are new to Swift. I realize that's obvious given that the book is called Pro Swift, but I want to make sure we're getting off on the right foot. Instead, this is a book for people who have some real-world knowledge of the language – perhaps you've made some apps yourself, perhaps you've read through Apple's official Swift reference guide, or perhaps you even read my first book, Hacking with Swift.

It is *not* required that you have read Hacking with Swift in order to make the most of this book, but I have to admit it certainly helps. At the very least you should have read the introduction chapter, which goes through the basics of Swift in about an hour of coding. If you haven't already read that chapter, I suggest you stop reading, go and buy Hacking with Swift, and read it now – that link gives you $5 off as a thank you for buying Pro Swift.

If you're still here, I'm assuming it means you're ready to learn about what's in this book. Well, I have some good news: I have distilled 50,000 words of knowledge into this little volume, all of which is dedicated to helping you write your best Swift code. I've tried to cover a wide range of topics so that everyone will find something of interest, which means you'll find chapters on closures, chapters on functional programming, chapters on MVC, chapters on operator overloading, and more – it's a real feast for developers who want to see what Swift is truly capable of.

I have purposefully gone way beyond the original chapter list I originally announced, partly because I enjoy giving people more than they expected, but mostly because working with advanced Swift is so much fun. I have to admit that it's easy to slip into writing "Objective-C in Swift" when you're an experienced Apple developer, at which point the language just seems like a bossy school teacher that enjoys pointing out your mistakes. But once you start writing idiomatic Swift – Swift as it was meant to be written – suddenly you realize you have an elegant, efficient, and expressive language that will help you ship better software.

There is no right way to read Pro Swift. You can jump in at whatever chapter interests you if that's what works, although I suspect you'll find it easier just to start at the beginning and work your way through. I make no apologies for jumping around so many topics – I've tried to be as precise as possible while still giving you hands on examples you can try as you read. And please do try the examples!

One more thing: obviously I hope you enjoy all the chapters in the book, but there were a few that I particularly enjoyed writing and I hope you enjoy them as much while reading. They are "Labeled statements", "Throwing functions as parameters", "flatMap()", and "Function composition" – have fun!

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 **CYCLONE**. 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 nephew John, who is right now taking his first steps in programming. He's a smart kid with a lot of ideas, but most importantly he's massively curious about the world around him. I hope one day not too far in the future this book will be useful to him too.

Acknowledgements

A number of people were kind enough to contribute to the creation of this book, and I'm most grateful. In particular, some really great developers took the time to write a short snippet to open each chapter of the book, giving you some fresh, practical insights from Swift developers I respect deeply. In alphabetical order they are Wayne Bishop, Chris Eidhof, Matt Gallagher, Simon Gladman, Wendy Lu, Andy Matuschak, Natasha Murashev, and Veronica Ray – thank you all!

I'm also grateful to Michael Mayer, who helps run the Philly CocoaHeads book club, for providing a second opinion on a chapter I was doubtful about. I ended up removing it from the book – as much as I hate deleting work I've written, I would hate it even more if I was giving you something I thought wasn't up to scratch.

In places I have used small amounts of source code from the official Swift open source project, mostly to illustrate specific techniques. This code is released under the Apache License v2.0 with the following header:

```
Copyright (c) 2014 - 2016 Apple Inc. and the Swift project
authors
Licensed under Apache License v2.0 with Runtime Library
Exception
See http://swift.org/LICENSE.txt for license information
See http://swift.org/CONTRIBUTORS.txt for the list of Swift
project authors
```

The Swift development team continue to do incredible work. My job – and yours! – would be very different without them, and it's important we thank them for their continuing efforts. In particular, Joe Groff (@jckarter) seems to have more hours in the day than anyone else, and I'm grateful for all the times he's answered my questions.

Get in touch

As this is an early release of a new book, I would love to hear your feedback. If you spot any typos or other mistakes, please email me at paul@hackingwithswift.com or tweet me @twostraws and I'll do my best to correct quickly. I also intend to write new chapters in the coming weeks, so please don't hesitate to get in touch with special requests.

Thank you for your support – I love programming in Swift, writing about Swift, and teaching others about Swift. Your support is what makes that possible, and it means a lot to me.

Copyright

Chapter 1
Syntax

Wendy Lu (@wendyluwho), iOS engineer at Pinterest

Use `final` on properties and methods when you know that a declaration does not need to be overridden. This allows the compiler to replace these dynamically dispatched calls with direct calls. You can even mark an entire class as final by attaching the attribute to the class itself.

Pattern matching

`Switch/case` is not a new concept: insert a value, then take one of several courses of action. Swift's focus on safety adds to the mix a requirement that all possible cases be catered for – something you don't get in C without specific warnings enabled – but that's fairly trivial.

What makes Swift's **`switch`** syntax interesting is its flexible, expressive pattern matching. What makes it *doubly* interesting is that since Swift's launch most of this pattern matching has been extended elsewhere, so that same flexible, expressive syntax is available in **`if`** conditions and **`for`** loops.

Admittedly, if you jump in at the deep end you're more likely to sink rather than swim, so I want to work up from basic examples. To refresh your memory, here's a basic **`switch`** statement:

```
let name = "twostraws"

switch name {
case "bilbo":
   print("Hello, Bilbo Baggins!")
case "twostraws":
   print("Hello, Paul Hudson!")
default:
   print("Authentication failed")
}
```

It's easy enough when you're working with a simple string, but things get more complicated when working with two or more values. For example, if we wanted to validate a name as well as a password, we would evaluate them as a tuple:

```
let name = "twostraws"
let password = "fr0st1es"

switch (name, password) {
case ("bilbo", "bagg1n5"):
   print("Hello, Bilbo Baggins!")
```

```
case ("twostraws", "fr0st1es"):
   print("Hello, Paul Hudson!")
default:
   print("Who are you?")
}
```

You can combine the two values into a single tuple if you prefer, like this:

```
let authentication = (name: "twostraws", password: "fr0st1es")

switch authentication {
case ("bilbo", "bagg1n5"):
   print("Hello, Bilbo Baggins!")
case ("twostraws", "fr0st1es"):
   print("Hello, Paul Hudson!")
default:
   print("Who are you?")
}
```

In this instance, both parts of the tuple must match the **`case`** in order for it to be executed.

Partial matches

When working with tuples, there are some occasions when you want a partial match: you care what some values are but don't care about others. In this situation, use underscores to represent "any value is fine", like this:

```
let authentication = (name: "twostraws", password: "fr0st1es",
ipAddress: "127.0.0.1")

switch authentication {
case ("bilbo", "bagg1n5", _):
   print("Hello, Bilbo Baggins!")
case ("twostraws", "fr0st1es", _):
   print("Hello, Paul Hudson!")
```

```
default:
    print("Who are you?")
}
```

Remember: Swift will take the first matching case it finds, so you need to ensure you look for the most specific things first. For example, the code below would print "You could be anybody!" because the first **case** matches immediately, even though later cases are "better" matches because they match more things:

```
switch authentication {
case (_, _, _):
    print("You could be anybody!")
case ("bilbo", "bagg1n5", _):
    print("Hello, Bilbo Baggins!")
case ("twostraws", "fr0st1es", _):
    print("Hello, Paul Hudson!")
default:
    print("Who are you?")
}
```

Finally, if you want to match only part of a tuple, but still want to know what the other part was, you should use **let** syntax.

```
switch authentication {
case ("bilbo", "bagg1n5", _):
    print("Hello, Bilbo Baggins!")
case ("twostraws", let password, _):
    print("Hello, Paul Hudson: your password was \(password)!")
default:
    print("Who are you?")
}
```

Matching calculated tuples

So far we've covered the basic range of pattern-matching syntax that most developers use. From here on I want to give examples of other useful pattern-matching techniques that are less well known.

Tuples are most frequently created using static values, like this:

```
let name = ("Paul", "Hudson")
```

But tuples are like any other data structure in that they can be created using dynamic code. This is particularly useful when you want to narrow the range of values in a tuple down to a smaller subset so that you need only a handful of **`case`** statements.

To give you a practical example, consider the "fizzbuzz" test: write a function that accepts any number, and returns "Fizz" if the number is evenly divisible by 3, "Buzz" if it's evenly divisible by 5, "FizzBuzz" if its evenly divisible by 3 *and* 5, or the original input number in other cases.

We can calculate a tuple to solve this problem, then pass that tuple into a **`switch`** block to create the correct output. Here's the code:

```
func fizzbuzz(number: Int) -> String {
    switch (number % 3 == 0, number % 5 == 0) {
    case (true, false):
        return "Fizz"
    case (false, true):
        return "Buzz"
    case (true, true):
        return "FizzBuzz"
    case (false, false):
        return String(number)
    }
}

print(fizzbuzz(number: 15))
```

This approach breaks down a large input space – any number – into simple combinations of true and false, and we then use tuple pattern matching in the case statements to select the correct output.

Loops

As you've seen, pattern matching using part of a tuple is easy enough: you either tell Swift what should be matched, use **let** to bind a value to a local constant, or use **_** to signal that you don't care what a value is.

We can use this same approach when working with loops, which allows us to loop over items only if they match the criteria we specify. Let's start with a basic example again:

```
let twostraws = (name: "twostraws", password: "fr0st1es")
let bilbo = (name: "bilbo", password: "bagg1n5")
let taylor = (name: "taylor", password: "fr0st1es")

let users = [twostraws, bilbo, taylor]

for user in users {
    print(user.name)
}
```

That creates an array of tuples, then loops over each one and prints its **name** value.

Just like the **switch** blocks we looked at earlier, we can use **case** with a tuple to match specific values inside the tuples. Add this code below the previous loop:

```
for case ("twostraws", "fr0st1es") in users {
    print("User twostraws has the password fr0st1es")
}
```

We also have identical syntax for binding local constants to the values of each tuple, like this:

```
for case (let name, let password) in users {
    print("User \(name) has the password \(password)")
```

```
}
```

Usually, though, it's preferable to re-arrange the **let** to this:

```
for case let (name, password) in users {
    print("User \(name) has the password \(password)")
}
```

The magic comes when you combine these two approaches, and again is syntactically identical to a **switch** example we already saw:

```
for case let (name, "fr0st1es") in users {
    print("User \(name) has the password \"fr0st1es\"")
}
```

That filters the **users** array so that only items with the password "fr0st1es" will be used in the loop, then creates a **name** constant inside the loop for you to work with.

Don't worry if you're staring at **for case let** and seeing three completely different keywords mashed together: it's not obvious what it does until someone explains it to you, and it will take a little time to sink in. But we're only getting started…

Matching optionals

Swift has two ways of matching optionals, and you're likely to meet both. First up is using **.some** and **.none** to match "has a value" and "has no value", and in the code below this is used to check for values and bind them to local constants:

```
let name: String? = "twostraws"
let password: String? = "fr0st1es"

switch (name, password) {
case let (.some(name), .some(password)):
    print("Hello, \(name)")
case let (.some(name), .none):
```

```
    print("Please enter a password.")
default:
    print("Who are you?")
}
```

That code is made more confusing because **`name`** and **`password`** are used for the input constants as well as the locally bound constants. They are different things, though, which is why **`print("Hello, \(name)")`** won't print **`Hello, Optional("twostraws")`** – the **`name`** being used is the locally bound unwrapped optional.

If it's easier to read, here's the same code with different names used for the matched constants:

```
switch (name, password) {
case let (.some(matchedName), .some(matchedPassword)):
    print("Hello, \(matchedName)")
case let (.some(matchedName), .none):
    print("Please enter a password.")
default:
    print("Who are you?")
}
```

The second way Swift matches optionals is using much simpler syntax, although if you have a fear of optionals this might only make it worse:

```
switch (name, password) {
case let (name?, password?):
    print("Hello, \(name)")
case let (username?, nil):
    print("Please enter a password.")
default:
    print("Who are you?")
}
```

This time the question marks work in a similar way as optional chaining: continue only if a value was found.

Both of these methods work equally well in **`for case let`** code. The code below uses them both to filter out **`nil`** values in a loop:

```
import Foundation
let data: [Any?] = ["Bill", nil, 69, "Ted"]

for case let .some(datum) in data {
   print(datum)
}

for case let datum? in data {
   print(datum)
}
```

Matching ranges

You're probably already using pattern matching with ranges, usually with code something like this:

```
let age = 36

switch age {
case 0 ..< 18:
   print("You have the energy and time, but not the money")
case 18 ..< 70:
   print("You have the energy and money, but not the time")
default:
   print("You have the time and money, but not the energy")
}
```

A very similar syntax is also available for regular conditional statements – we could rewrite that code like this:

```
if case 0 ..< 18 = age {
```

```
    print("You have the energy and time, but not the money")
} else if case 18 ..< 70 = age {
    print("You have the energy and money, but not the time")
} else {
    print("You have the time and money, but not the energy")
}
```

That produces identical results to the **`switch`** block while using similar syntax, but I'm not a big fan of this approach. The reason for my dislike is simple readability: I don't think "if case 0 up to 18 equals age" makes sense if you don't already know what it means. A much nicer approach is to use the pattern match operator, **`~=`**, which would look like this:

```
if 0 ..< 18 ~= age {
    print("You have the energy and time, but not the money")
} else if 18 ..< 70 ~= age {
    print("You have the energy and money, but not the time")
} else {
    print("You have the time and money, but not the energy")
}
```

Now the condition reads "if the range 0 up to 18 matches age", which I think makes a lot more sense.

An even cleaner solution becomes clear when you remember that **`0 ..< 18`** creates an instance of a **`Range`** struct, which has its own set of methods. Right now, its **`contains()`** method is particularly useful: it's longer to type than **`~=`** but it's significantly easier to understand:

```
if (0 ..< 18).contains(age) {
    print("You have the energy and time, but not the money")
} else if (18 ..< 70).contains(age) {
    print("You have the energy and money, but not the time")
} else {
    print("You have the time and money, but not the energy")
}
```

You can combine this range matching into our existing tuple matching code, like this:

```
let user = (name: "twostraws", password: "fr0st1es", age: 36)

switch user {
case let (name, _, 0 ..< 18):
   print("\(name) has the energy and time, but no money")
case let (name, _, 18 ..< 70):
   print("\(name) has the money and energy, but no time")
case let (name, _, _):
   print("\(name) has the time and money, but no energy")
}
```

That last case binds the user's name to a local constant called **`name`** irrespective of the two other values in the tuple. This is a catch all, but because Swift looks for the *first* matching case this won't conflict with the other two in the **`switch`** block.

Matching enums and associated values

In my experience, quite a few people don't really understand enums and associated values, and so they struggle to make use of them with pattern matching. There's a whole chapter on enums later in the book, so if you're not already comfortable with enums and associated values you might want to pause here and read that chapter first.

Basic enum matching looks like this:

```
enum WeatherType {
   case cloudy
   case sunny
   case windy
}

let today = WeatherType.cloudy
```

```
switch today {
case .cloudy:
   print("It's cloudy")
case .windy:
   print("It's windy")
default:
   print("It's sunny")
}
```

You'll also have used enums in basic conditional statements, like this:

```
if today == .cloudy {
   print("It's cloudy")
}
```

As soon as you add associated values, things get more complicated because you can use them, filter on them, or ignore them depending on your goal.

First up, the easiest option: creating an associated value but ignoring it:

```
enum WeatherType {
   case cloudy(coverage: Int)
   case sunny
   case windy
}

let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy:
   print("It's cloudy")
case .windy:
   print("It's windy")
default:
   print("It's sunny")
```

```
}
```

Using this approach, the actual **switch** code is unchanged.

Second: creating an associated value and using it. This uses the same local constant bind we've seen several times now:

```
enum WeatherType {
    case cloudy(coverage: Int)
    case sunny
    case windy
}

let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy(let coverage):
    print("It's cloudy with \(coverage)% coverage")
case .windy:
    print("It's windy")
default:
    print("It's sunny")
}
```

Lastly: creating an associated type, binding a local constant to it, but also using that binding to filter for specific values. This uses the **where** keyword to create a requirements clause that clarifies what you're looking for. In our case, the code below prints two different messages depending on the associated value that is used with **cloudy**:

```
enum WeatherType {
    case cloudy(coverage: Int)
    case sunny
    case windy
}
```

```
let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy(let coverage) where coverage < 100:
   print("It's cloudy with \(coverage)% coverage")
case .cloudy(let coverage) where coverage == 100:
   print("You must live in the UK")
case .windy:
   print("It's windy")
default:
   print("It's sunny")
}
```

Now, as promised I'm building up from basic examples, but if you're ready I want to show you how to combine two of these techniques together: associated values and range matching. The code below now prints four different messages: one when coverage is 0, one when it's 100, and two more using ranges from 1 to 50 and 51 to 99:

```
enum WeatherType {
   case cloudy(coverage: Int)
   case sunny
   case windy
}

let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy(let coverage) where coverage == 0:
   print("You must live in Death Valley")
case .cloudy(let coverage) where (1...50).contains(coverage):
   print("It's a bit cloudy, with \(coverage)% coverage")
case .cloudy(let coverage) where (51...99).contains(coverage):
   print("It's very cloudy, with \(coverage)% coverage")
case .cloudy(let coverage) where coverage == 100:
```

```
    print("You must live in the UK")
case .windy:
    print("It's windy")
default:
    print("It's sunny")
}
```

If you want to match associated values in a loop, adding a **`where`** clause is the wrong approach. In fact, this kind of code won't even compile:

```
let forecast: [WeatherType] = [.cloudy(coverage:
40), .sunny, .windy, .cloudy(coverage: 100), .sunny]

for day in forecast where day == .cloudy {
    print(day)
}
```

That code would be fine without associated values, but because the associated value has meaning the **`where`** clause isn't up to the job – it has no way to say "and bind the associated value to a local constant." Instead, you're back to **`case let`** syntax, like this:

```
let forecast: [WeatherType] = [.cloudy(coverage:
40), .sunny, .windy, .cloudy(coverage: 100), .sunny]

for case let .cloudy(coverage) in forecast {
    print("It's cloudy with \(coverage)% coverage")
}
```

If you know the associated value and want to use it as a filter, the syntax is almost the same:

```
let forecast: [WeatherType] = [.cloudy(coverage:
40), .sunny, .windy, .cloudy(coverage: 100), .sunny]

for case .cloudy(40) in forecast {
    print("It's cloudy with 40% coverage")
```

```
}
```

Matching types

You should already know the **`is`** keyword for matching, but you might not know that it can be used as pattern matching in loops and **`switch`** blocks. I think the syntax is quite pleasing, so I want to demonstrate it just briefly:

```
let view: AnyObject = UIButton()

switch view {
case is UIButton:
   print("Found a button")
case is UILabel:
   print("Found a label")
case is UISwitch:
   print("Found a switch")
case is UIView:
   print("Found a view")
default:
   print("Found something else")
}
```

I've used UIKit as an example because you should already know that **`UIButton`** inherits from **`UIView`**, and I need to give you a big warning…

Remember: Swift will take the first matching case it finds, and **`is`** returns true if an object is a specific type or one of its parent classes. So, the above code will print "Found a button", whereas the below code will print "Found a view":

```
let view: AnyObject = UIButton()

switch view {
case is UIView:
   print("Found a view")
```

```
case is UIButton:
   print("Found a button")
case is UILabel:
   print("Found a label")
case is UISwitch:
   print("Found a switch")
default:
   print("Found something else")
}
```

To give you a more useful example, you can use this approach to loop over all subviews in an array and filter for labels:

```
for label in view.subviews where label is UILabel {
   print("Found a label with frame \(label.frame)")
}
```

Even though **where** ensures only **UILabel** objects are processed in the loop, it doesn't actually do any typecasting. This means if you wanted to access a label-specific property of **label**, such as its **text** property, you need to typecast it yourself. In this situation, using **for case let** instead is easier, as this filters and typecasts in one:

```
for case let label as UILabel in view.subviews {
   print("Found a label with text \(label.text)")
}
```

Using the where keyword

To wrap up pattern matching, I want to demonstrate a couple of interesting ways to use **where** clauses so that you can get an idea of what it's capable of.

First, an easy one: loop over an array of numbers and print only the odd ones. This is trivial using **where** and modulus, but it demonstrates that your **where** clause can contain calculations:

```
for number in numbers where number % 2 == 1 {
    print(number)
}
```

You can also call methods, like this:

```
let celebrities = ["Michael Jackson", "Taylor Swift", "Michael Caine", "Adele Adkins", "Michael Jordan"]

for name in celebrities where !name.hasPrefix("Michael") {
    print(name)
}
```

That will print "Taylor Swift" and "Adele Adkins". If you want to make your **where** clause more complicated, just add operators such as **&&**:

```
let celebrities = ["Michael Jackson", "Taylor Swift", "Michael Caine", "Adele Adkins", "Michael Jordan"]

for name in celebrities where name.hasPrefix("Michael") && name.characters.count == 13 {
    print(name)
}
```

That will print "Michael Caine".

While it's possible to use **where** to strip out optionals, I wouldn't recommend it. Consider the example below:

```
let celebrities: [String?] = ["Michael Jackson", nil, "Michael Caine", nil, "Michael Jordan"]

for name in celebrities where name != nil {
    print(name)
}
```

That certainly works, but it does nothing about the optionality of the strings in the loop so it prints out this:

```
Optional("Michael Jackson")
Optional("Michael Caine")
Optional("Michael Jordan")
```

Instead, use **`for case let`** to handle optionality, and use **`where`** to focus on filtering values. Here's the preferred way of writing that loop:

```
for case let name? in celebrities {
    print(name)
}
```

When that runs, **`name`** will only contain the strings that had values, so its output will be:

```
Michael Jackson
Michael Caine
Michael Jordan
```

Nil coalescing

Swift optionals are one of the fundamental ways it guarantees program safety: a variable can only be used if it definitely has a value. The problem is that optionals make your code a bit harder to read and write, because you need to unwrap them safely.

One alternative is to explicitly unwrap optionals using `!`. This is also known as the "crash operator" because if you use `!` with an optional that is nil, your program will die immediately and your users will be baying for blood.

A smarter alternative is the nil coalescing operator, `??`, which allows you to access an optional and provide a default value if the optional is nil.

Consider this optional:

```
let name: String? = "Taylor"
```

That's a constant called `name` that contains either a string or nil. If you try to print that using `print(name)` you'll see `Optional("Taylor")` rather than just "Taylor", which isn't really what you want.

Using nil coalescing allows us to use an optional's value or provide a default value if it's nil. So, you could write this:

```
let name: String? = "Taylor"
let unwrappedName = name ?? "Anonymous"
print(unwrappedName)
```

That will print "Taylor": `name` was a `String?`, but `unwrappedName` is guaranteed to be a regular `String` – not optional – because of the nil coalescing operator. To see the default value in action, try this instead:

```
let name: String? = nil
let unwrappedName = name ?? "Anonymous"
print(unwrappedName)
```

That will now print "Anonymous", because the default value is used instead.

Of course, you don't need a separate constant when using nil coalescing – you can write it inline, like this:

```
let name: String? = "Taylor"
print(name ?? "Anonymous")
```

As you can imagine, nil coalescing is great for ensuring sensible values are in place before you use them, but it's particularly useful in removing some optionality from your code. For example:

```
func returnsOptionalName() -> String? {
    return nil
}

let returnedName = returnsOptionalName() ?? "Anonymous"
print(returnedName)
```

Using this approach, **`returnedName`** is a **`String`** rather than a **`String?`** because it's guaranteed to have a value.

So far, so straightforward. However, nil coalescing gets more interesting when you combine it with the **`try?`** keyword.

Consider a simple app that lets a user type and save text. When the app runs, it wants to load whatever the user typed previously, so it probably uses code like this:

```
do {
    let savedText = try String(contentsOfFile: "saved.txt")
    print(savedText)
} catch {
    print("Failed to load saved text.")
}
```

If the file exists, it will be loaded into the **`savedText`** constant. If not, the **`contentsOfFile`** initializer will throw an exception, and “Failed to load saved text” will be

printed. In practice, you'd want to extend this so that **savedText** always has a value, so you end up with something like this:

```
let savedText: String

do {
    savedText = try String(contentsOfFile: "saved.txt")
} catch {
    print("Failed to load saved text.")
    savedText = "Hello, world!"
}

print(savedText)
```

That's a lot of code and it doesn't really accomplish very much. Fortunately, there's a better way: nil coalescing. Remember, **try** has three variants: **try** attempts some code and might throw an exception, **try!** attempts some code and crashes your app if it fails, and **try?** attempts some code and returns nil if the call failed.

That last one is where nil coalescing steps up to the plate, because this exactly matches our previous examples: we want to work with an optional value, and provide a sensible default if the optional is nil. In fact, using nil coalescing we can rewrite all that into just two lines of code:

```
let savedText = (try? String(contentsOfFile: "saved.txt")) ??
"Hello, world!"
print(savedText)
```

That means "try loading the file, but if loading fails use this default text instead" – a neater solution and much more readable.

Combining **try?** with nil coalescing is perfect for situations when a failed **try** isn't an error, and I think you'll find this pattern useful in your own code.

Guard

The **guard** keyword has been with us since Swift 2.0, but because it does four things in one you'd be forgiven for not using it fully.

The first use is the most obvious: **guard** is used for early returns, which means you exit a function if some preconditions are not satisfied. For example, we could write a rather biased function to give an award to a named person:

```
func giveAward(to name: String) {
    guard name == "Taylor Swift" else {
        print("No way!")
        return
    }

    print("Congratulations, \(name)!")
}

giveAward(to: "Taylor Swift")
```

Using that **guard** statement in the **giveAward(to:)** method ensures that only Taylor Swift wins awards. It's biased like I said, but the precondition is clear and this code will only run when requirements I have put in place are satisfied.

This initial example looks almost identical to using **if**, but **guard** has one massive advantage: it makes your intention clear, not only to people but also to the compiler. This is an early return, meaning that you want to exit the method if your preconditions aren't satisfied. Using **guard** makes that clear: this condition isn't part of the functionality of the method, it's just there to ensure the actual code is safe to run. It's also clear to the compiler, meaning that if you remove the **return** statement your code will no longer build – Swift knows this is an early return, so it will not let you forget to exit.

The second use of guard is a happy side effect of the first: using **guard** and early returns allows you to reduce your indent level. Some developers believe very strongly that early returns must never be used, and instead each function should return from only one place. This forces extra indents in the main body of your function code, something like this:

```
func giveAward(to name: String) -> String {
    let message: String

    if name == "Taylor Swift" {
        message = "Congratulations, \(name)!"
    } else {
        message = "No way!"
    }

    return message
}

giveAward(to: "Taylor Swift")
```

With **`guard`**, you get your preconditions resolved immediately, and lose the extra indent – hurray for neat code!

The third thing that **`guard`** brings us is a visibility increase for the happy path. This is a common concept in software design and testing, and refers to the path that your code will take when no exceptions or errors happen. Thanks to **`guard`**, common errors are removed immediately, and the remainder of your code might all be the happy path.

That's all the easy stuff out of the way, but **`guard`** still has one more feature I want to discuss, and it's an important differentiator between **`guard`** and **`if`**: when you use **`guard`** to check and unwrap an optional, that optional stays in scope.

To demonstrate this, I'm going to rewrite the **`giveAward(to:)`** method so that it takes an optional string:

```
func giveAward(to name: String?) {
    guard let winner = name else {
        print("No one won the award")
        return
    }
```

```
    print("Congratulations, \(winner)!")
}
```

With a regular **`if-let`**, the **`winner`** constant would only be usable inside the braces that belong to **`guard`**. However, **`guard`** keeps its optional unwraps in scope, so **`winner`** stays around for the second **`print()`** statement. This code reads "try to unwrap name into winner so I can use it, but if you can't then print a message and exit."

There's one last feature of **`guard`** I want to touch on, but it's not new. Instead, it's just a different way of using what you already know. The feature is this: **`guard`** lets you exit *any* scope if preconditions fail, not just functions and methods. This means you can exit a **`switch`** block or a loop by using guard, and it carries the same meaning: the contents of this scope should only be executed if these preconditions are true.

To give you a simple example, this loop counts from 1 to 100, printing out all the numbers that are evenly divisible by 8:

```
for i in 1...100 {
    guard i % 8 == 0 else { continue }

    print(i)
}
```

Can you rewrite that using **`where`**? Give it a try – it's easier than you think!

Lazy loading

Lazy loading is one of the most important, system-wide performance optimizations that Swift coders work with. It's endemic in iOS, as anyone who has tried to manipulate a view controller's view before it's shown can tell you. Objective-C didn't have a concept of lazy properties, so you had to write your own boilerplate code each time you wanted this behavior. Happily, Swift has it baked right in, so you can claim immediate performance benefits with hardly any code.

But first: a reminder of what lazy properties are. Consider this class:

```
class Singer {
    let name: String

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

    func reversedName() -> String {
        return "\(name.uppercased()) backwards is \
(String(name.uppercased().characters.reversed()))!"
    }
}

let taylor = Singer(name: "Taylor Swift")
print(taylor.reversedName())
```

That will print "TAYLOR SWIFT backwards is TFIWS ROLYAT!" when run.

So each **`Singer`** has a single property called **`name`**, and a single method that does a small amount of processing to that property. Obviously in your own code these functions are likely to do more significant work, but I'm trying to keep it simple here.

Every time you want to print the message "TAYLOR SWIFT backwards is TFIWS ROLYAT!" you need to call the **`reversedName()`** method – the work it does isn't stored, and if that work is non-trivial then calling the method repeatedly is wasteful.

An alternative is to create an additional property to store **reversedName** so that it's calculated only once, like this:

```
class Singer {
    let name: String
    let reversedName: String

    init(name: String) {
        self.name = name
        reversedName = "\(name.uppercased()) backwards is \
(String(name.uppercased().characters.reversed()))!"
    }
}

let taylor = Singer(name: "Taylor Swift")
print(taylor.reversedName)
```

That's a performance improvement for situations when you use **reversedName** a lot, but now causes your code to run *slower* if you never use **reversedName** – it gets calculated regardless of whether it's used, whereas when **reversedName()** was a method it would only be calculated when called.

Lazy properties are the middle ground: they are properties that are calculated only once and stored, but never calculated if they aren't used. So if your code uses a lazy property repeatedly you only pay the performance cost once, and if they are never used then the code is never run. It's a win-win!

Lazy closures

The easiest way to get started with the **lazy** keyword is using closures. Yes, I know it's rare to see "closures" and "easiest" in the same sentence, but there's a reason this book isn't called "Newbie Swift"!

The syntax here is a little unusual at first:

```
lazy var yourVariableName: SomeType = {
    return SomeType(whatever: "foobar")
}()
```

Yes, you need to explicitly declare the type. Yes, you need that **=** sign. Yes, you need the parentheses after the closing brace. It's a little unusual, like I said, but it's all there for a reason: you're creating the closure, applying it immediately (rather than later on), and assigning its result back to **yourVariableName**.

Using this approach, we can convert our **reversedName()** method into a lazy property like this:

```
class Singer {
    let name: String

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

    lazy var reversedName: String = {
        return "\(self.name.uppercased()) backwards is \
(String(self.name.uppercased().characters.reversed()))!"
    }()
}

let taylor = Singer(name: "Taylor Swift")
print(taylor.reversedName)
```

Note: as it's now a property rather than a method, we need to use **print(taylor.reversedName)** rather than **print(taylor.reversedName())** to access the value.

That's it: the property is now lazy, which means the code inside the closure will be executed only the first time we read the **reversedName** property.

"But Paul," I hear you say, "you're using **self** inside a closure that's owned by the object – why are you giving me strong reference cycles?" Don't worry: this code is quite safe. Swift is smart enough to realize what's going on, and no reference cycle will be created.

Under the hood, any closure like this – one that is immediately applied – is considered to be "non-escaping", which in our situation means it won't be used anywhere else. That is, this closure can't be stored as a property and called later on. Not only does this automatically ensure **self** is considered to be **unowned**, but it also enables the Swift compiler to make some extra optimizations because it has more information about the closure.

Lazy methods

A common complaint people have when using **lazy** is that it clutters up their code: rather than having a neat separation of properties and methods, lazy properties become a gray area where properties and functionality mix together. There is a simple solution to this: create methods to separate your lazy properties from the code they rely on.

If you want to use this approach, I suggest you mark as private the separate method you create so that it doesn't get used by accident. Something like this ought to do the trick:

```
class Singer {
    let name: String

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

    lazy var reversedName: String = self.getReversedName()

    private func getReversedName() -> String {
        return "\(name.uppercased()) backwards is \
(String(name.uppercased().characters.reversed()))!"
    }
}
```

```
let taylor = Singer(name: "Taylor Swift")
print(taylor.reversedName)
```

Lazy singletons

Singletons are one of several common programming patterns that I'm not particularly fond of. If you're not familiar with them, a singleton is a value or object that is designed (and coded) to be created only once, and shared across a program. For example, if your app uses a logger, you might create a logger object once when the app runs, and have all your other code use that shared instance.

The reason I'm not a big fan of singletons is a simple one: they are all too often used like global variables. Many people will preach that global variables are bad then happily abuse singletons in much the same way, which is just sloppy.

That being said, there are good reasons for using singletons, and indeed Apple uses them on occasion. If your object literally can only exist once – such as an instance of **`UIApplication`** – then singletons make sense. On iOS, things like **`UIDevice`** make sense as singletons, again because they can exist only once. Singletons are also useful (at least compared to global variables!) if you want to add extra code when they are used.

So: singletons have a place, as long as you consider their use carefully. If you think singletons are the perfect choice for you, I have some good news: Swift makes singletons insanely easy.

To give you a practical example, we're going to create a **`Singer`** class that will have a **`MusicPlayer`** class as a property. This needs to be a singleton, because no matter how many singers our app tracks, we want all their songs to go through the same music player so that music doesn't overlap.

Here's the **`MusicPlayer`** class:

```
class MusicPlayer {
    init() {
        print("Ready to play songs!")
    }
```

```
}
```

It doesn't do anything other than print a message when it's created.

Here's the basic **`Singer`** class that also does nothing other than print a message when it's created:

```
class Singer {
    init() {
        print("Creating a new singer")
    }
}
```

Now for the singleton: if we want to give our **`Singer`** class a **`MusicPlayer`** singleton property, we need to add just one line of code inside the **`Singer`** class:

```
static let musicPlayer = MusicPlayer()
```

That's it. The **`static`** part means this property is shared by the class rather than instances of the class, meaning that you use **`Singer.musicPlayer`** rather than **`taylor.musicPlayer`**. The **`let`** part of course means that it's a constant.

You might be wondering what all this has to do with lazy properties, and it's time to find out – put this code into a playground:

```
class MusicPlayer {
    init() {
        print("Ready to play songs!")
    }
}

class Singer {
    static let musicPlayer = MusicPlayer()

    init() {
```

```
        print("Creating a new singer")
    }
}

let taylor = Singer()
```

When it runs, the output is "Creating a new singer" – the "Ready to play songs!" message won't appear. If you add one more line to the end of your playground, only then will the message appear:

```
Singer.musicPlayer
```

Yes: all Swift **`static let`** singletons are automatically lazy – they only get created when they are needed. It's so easy to do, and yet perfectly efficient too. Thanks, Swift team!

Lazy sequences

Now that you understand lazy properties, I want to explain briefly the usefulness of lazy sequences. These are similar to lazy properties in that they delay work until necessary, but they aren't *quite* as efficient as you'll see shortly.

Let's start with a trivial example: the Fibonacci sequence. As a reminder, this is a sequence of numbers starting with 0 and 1, where every following number is the sum of adding the previous two. So the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on.

We can write a naïve function that calculates the Fibonacci number at a particular point in the sequence like this:

```
func fibonacci(of num: Int) -> Int {
    if num < 2 {
        return num
    } else {
        return fibonacci(of: num - 1) + fibonacci(of: num - 2)
    }
}
```

This is a recursive function: it calls itself. It's a naïve implementation because it doesn't cache the results as it goes, which means all the adding done by **`fibonacci(of: num - 1)`** won't get re-used by **`fibonacci(of: num - 2)`** even though it could be. However, this implementation is perfect for demonstrating the usefulness (and drawbacks!) of lazy sequences.

Open a playground and add this code:

```
func fibonacci(of num: Int) -> Int {
    if num < 2 {
        return num
    } else {
        return fibonacci(of: num - 1) + fibonacci(of: num - 2)
    }
}

let fibonacciSequence = (0...20).map(fibonacci)
print(fibonacciSequence[10])
```

That calculates the first 21 numbers of the Fibonacci sequence, and prints out the 11th: 55. I asked you to put this in a playground because Xcode will show you how often the code is executed, and you'll see the **`return num`** line being called 28,656 times – a huge amount of work. If you try using **`0...21`** for the range – just one number higher! – you'll see that number goes up to 46,367 times.

Like I said, it's a naïve implementation, and it really doesn't scale well. Can you imagine using **`0...199`**? And what if you only need a few numbers rather than all of them?

This is where lazy sequences come into play: you give it a sequence to work with and tell it what code you want to run just like you would with a normal sequence, but now that code is executed on demand as you access items. So, we could prepare to generate the first 200 numbers in the Fibonacci, then only use the 20th value by using the **`lazy`** property of a sequence:

```
let lazyFibonacciSequence = Array(0...199).lazy.map(fibonacci)
print(lazyFibonacciSequence[19])
```

Note: you need to add the **`Array`** in there to ensure that Swift creates the lazy map on an array, rather than on the range **`0...199`**.

That new code takes a small amount of time to run, because all the other calculations are never run – no time is wasted.

However, as clever as lazy sequences are, they do have a drawback that lazy properties do not: they have no memoization. This is a common optimization technique that stores the result of computationally expensive code so that it doesn't need to be created again. This is essentially what regular lazy variables offer us: a guarantee not only that a property won't be created if it isn't used, but that it won't be created repeatedly when used again and again.

As I said, lazy sequences have *no* memoization, which means requesting the same data twice will require the work to be done twice. Try this now:

```
let lazyFibonacciSequence = Array(0...199).lazy.map(fibonacci)
print(lazyFibonacciSequence[19])
print(lazyFibonacciSequence[19])
print(lazyFibonacciSequence[19])
```

You'll see the code now takes three times as long to run. So, use lazy sequences when necessary, but remember there are situations when they might actually slow you down!

Destructuring

Destructuring (also known as decomposition) is a smart way to transfer data into and out from tuples, and when you start to understand it you'll realize how destructuring and pattern matching are closely linked. Destructuring has three uses: pulling a tuple apart into multiple values, assigning multiple things simultaneously, and switching values.

Consider this tuple:

```
let data = ("one", "two", "three")
```

If you want to create three different constants out of those three values, without destructuring you'd need to write this:

```
let one = data.0
let two = data.1
let three = data.2
```

With destructuring you can write this:

```
let (one, two, three) = data
```

Swift pulls the **`data`** tuple apart into those three individual constants, all in a single line of code.

This technique is particularly helpful when you're working with functions that return tuples, which is commonly used when you want to return multiple values. It's common to want to split up those return values so you can refer to them on your terms, particularly if they have no names in the tuple. For example:

```
func getPerson() -> (String, Int) {
    return ("Taylor Swift", 26)
}

let (name, age) = getPerson()
print("\(name) is \(age) years old")
```

If you want to ignore values during destructuring, use _, like this:

```
let (_, age) = getPerson()
print("That person is \(age) years old")
```

You can use this same technique to assign multiple things simultaneously, either using fixed values or using function calls. For example:

```
let (captain, chef) = ("Janeway", "Neelix")
let (engineer, pilot) = (getEngineer(), getPilot())
```

This is particularly useful when you're working with values that are closely related, such as coordinates for a rectangle, and can help improve readability.

Finally, tuple destructuring is good for swapping values. Now, I'll be honest: this technique is rarely useful outside of interviews, and even then it's a pretty poor choice for an interview question. However, I want to show it to you because I think it demonstrates just how graceful Swift can be.

So, here goes: given two integers, A and B, how do you swap them without using a third variable?

Have a think for a moment, and perhaps even try some code in a playground.

Here's how you would solve this in most languages:

```
var a = 10
var b = 20

a = a + b
b = a - b
a = a - b

print(a)
print(b)
```

In Swift, thanks to destructuring, you can write it in a single line:

```
(b, a) = (a, b)
```

Graceful, efficient, and quite beautiful I think. If you're ever asked this question in an interview, you should be able to ace it!

Labeled statements

Labels have been in use for a long time, but they largely fell out of favor when developers started to frown on **goto**. Swift brings them back, but without **goto**: instead they are used with loops to let you exit them more easily.

Here's some code that creates a grid of strings, and marks one of those squares with an "x" for where some treasure is – it's a hard-coded position here, but in a real game you'd obviously randomize it. The code then has two loops to try to find the treasure, with one loop nested inside the other: loop through all the rows in the board, then loop through each column in each row.

Here's the code:

```
var board = [[String]](repeating: [String](repeating: "",
count: 10), count: 5)

board[3][5] = "x"

for (rowIndex, cols) in board.enumerated() {
   for (colIndex, col) in cols.enumerated() {
      if col == "x" {
         print("Found the treasure at row \(rowIndex) col \
(colIndex)!")
      }
   }
}
```

Given that the treasure can appear once on the board, this code is quite wasteful: even though the treasure is found early on in the search, it carries on looking. If you're thinking it's time to deploy **break** you'd be right, at least partially. Here's how it might look:

```
for (rowIndex, cols) in board.enumerated() {
   for (colIndex, col) in cols.enumerated() {
      if col == "x" {
         print("Found the treasure at row \(rowIndex) col \
```

```
(colIndex)!")
            break
        }
    }
}
```

However, **break** only exits one level of loop, so it would exit the **for (colIndex, col)** loop then continue running the **for (rowIndex, cols)** loop. Yes, it wastes *less* time, but it's still wasting *some*. You could add a boolean variable that gets set to true when treasure is found, which you then use to break the outer loop, but Swift has a better solution: labeled statements.

Labeled statements let you give any loop a name, which allows you to refer to a specific loop when using **break** or **continue**. To create a label, just write a name then a colon before any loop. You can then use **break yourLabelName** or **continue yourLabelName** to refer directly to it.

So, the least wasteful way to write that code is like so:

```
var board = [[String]](repeating: [String](repeating: "",
count: 10), count: 5)

board[5][3] = "x"

rowLoop: for (rowIndex, cols) in board.enumerated() {
    for (colIndex, col) in cols.enumerated() {
        if col == "x" {
            print("Found the treasure at row \(rowIndex) col \
(colIndex)!")
            break rowLoop
        }
    }
}
```

That immediately jumps out of both loops, and continues on after the end of the **for**

`(rowIndex, cols)` loop – perfect.

Labeling your loops is clever, but Swift takes it a step further: it lets you label `if` statements then `break` from them as if they were a loop. This is extraordinarily useful when you find yourself deeply nested inside several conditions and want to bail out immediately, and without it you can end up with a pyramid of increasingly indented conditions.

Here's a worked example so you can see it in action:

```
if userRequestedPrint() {
    if documentSaved() {
        if userAuthenticated() {
            if connectToNetwork() {
                if uploadDocument("resignation.doc") {
                    if printDocument() {
                        print("Printed successfully!")
                    }
                }
            }
        }
    }
}
```

That code goes through a series of checks to allow a user to print a document: don't try running it, because those functions aren't real!

If all the conditions evaluate to `true`, then you'll see "Printed successfully!".

What labeled statements let you do is create early returns for your `if` statements. They run normally, but at any time you feel necessary you can exit any conditional statement. For example, we can rewrite the above pyramid into this:

```
printing: if userRequestedPrint() {
    if !documentSaved() { break printing }
    if !userAuthenticated() { break printing }
    if !connectToNetwork() { break printing }
```

```
    if !uploadDocument("work.doc") { break printing }
    if !printDocument() { break printing }

    print("Printed successfully!")
}
```

That takes up fewer lines, forces less indenting on people reading the code, and the happy path is immediately clear.

If you wanted to, you could even use **`guard`** to make your intention even clearer, like this:

```
printing: if userRequestedPrint() {
    guard documentSaved() else { break printing }
    guard userAuthenticated() else { break printing }
    guard connectToNetwork() else { break printing }
    guard uploadDocument("work.doc") else { break printing }
    guard printDocument() else { break printing }

    print("Printed successfully!")
}
```

For the sake of readability, I prefer to test for positive conditions rather than negative. That is, I'd prefer to test for **`if documentSaved()`** rather than **`if !documentSaved()`** because it's a little easier to understand, and **`guard`** does exactly that.

Nested functions, classes and structs

Swift lets you nest one data type inside another, e.g. a struct within a struct, an enum within a class, or a function within a function. This is most commonly used to help you mentally group things together by logical behavior, but can sometimes have access semantics attached to stop nested data types being used incorrectly.

Let's deal with the easy situation first: using nested types for logical grouping. Consider the code below, which defines an enum called **`London`**:

```
enum London {
    static let coordinates = (lat: 51.507222, long: -0.1275)

    enum SubwayLines {
        case bakerloo, central, circle, district, elizabeth,
hammersmithCity, jubilee, metropolitan, northern, piccadilly,
victoria, waterlooCity
    }

    enum Places {
        case buckinghamPalace, cityHall, oldBailey, piccadilly,
stPaulsCathedral
    }
}
```

That enum has one constant called **`coordinates`**, then two nested enums: **`SubwayLines`** and **`Places`**. But, notably, it has no cases of its own – it's just being used as a wrapper for other data.

This has two immediate benefits: first, any IDE with code completion makes it quick and easy to drill down to specific items by listing possible options as you type, for example **`London.Places.cityHall`**. Second, because you're effectively making namespaced constants, you can use sensible names like "Piccadilly" without worrying whether you mean the subway line or the place, or whether you mean London Piccadilly or Manchester Piccadilly.

If you extend this technique further, you'll realize you can use it for storyboard IDs, table view cell IDs, image names, and more – effectively doing away with the stringly typed resources that are so prevalent on Apple's platforms. For example:

```
enum R {
    enum Storyboards: String {
        case main, detail, upgrade, share, help
    }

    enum Images: String {
        case welcome, home, about, button
    }
}
```

Bonus points if you understand why I used the name **R** for this. To make the technique work for images, just name your images the same as your enum case, with ".png" on the end, e.g. “about.png".

(To put you out of your misery: I used the name R for the resources because this is the exact approach Android uses. If you think Android using it makes R a bad idea, you need to re-evaluate your life choices.)

Nested types also work with other data types, for example you can have a struct that contains its own enum:

```
struct Cat {
    enum Breed {
        case britishShortHair, burmese, persian, ragdoll,
russianBlue, scottishFold, siamese
    }

    var name: String
    var breed: Breed
}
```

You can also place structs within structs when they are used together, for example:

```
struct Deck {
    struct Card {
        enum Suit {
            case hearts, diamonds, clubs, spades
        }

        var rank: Int
        var suit: Suit
    }

    var cards = [Card]()
}
```

As you can see in that last example, you can nest as many times as you want to – an enum within a struct within another struct is perfectly legal.

Nesting with semantics

Nesting for logical grouping doesn't stop you from referring to any of the nested types, although if you nest things too much it gets a bit cumbersome:

```
let home = R.Images.home
let burmese = Cat.Breed.burmese
let hearts = Deck.Card.Suit.hearts
```

However, Swift allows you to assign access control modifiers to nested types to control how they can be used. This is useful when a nested type has been designed to work specifically inside its parent: if the **`Card`** struct must only be used by the **`Deck`** struct, then you need access control.

Be warned: if a property uses a private type, the property itself must be private. To demonstrate, let's look at the **`Deck`** example again:

```
struct Deck {
    struct Card {
        enum Suit {
            case hearts, diamonds, clubs, spades
        }

        var rank: Int
        var suit: Suit
    }

    var cards = [Card]()
}
```

If we wanted that **`Suit`** enum to be private so that only **`Card`** instances can use it, we'd need to use **`private enum Suit`**. However, this has the knock-on effect of requiring that the **`suit`** property of **`Card`** also be private, otherwise it would be accessible in places where the **`Suit`** enum was not. So, the updated code is this:

```
struct Deck {
    struct Card {
        private enum Suit {
            case hearts, diamonds, clubs, spades
        }

        var rank: Int
        private var suit: Suit
    }

    var cards = [Card]()
}
```

Nested functions

Nested functions are an interesting corner case of access control for nested types, because they

are automatically restricted to their enclosing function unless you specify otherwise. Swift implements nested functions as named closures, which means they automatically capture values from their enclosing function.

To illustrate nested functions, I'm going to create a function that calculates the distance between two points using one of three distance calculation techniques: Euclidean (using Pythagoras's theorem), Euclidean squared (using Pythagoras's theorem, but avoiding the **`sqrt()`** call for performance reasons), and Manhattan. If you're not familiar with these terms, "Euclidean distance" is basically drawing a straight line between two points, and "Manhattan distance" uses rectilinear geometry to calculate the absolute difference of two Cartesian coordinates. Try to imagine a taxi cab moving through a city where the blocks are square: you can move two blocks up then two to the right, or two to the right and two up, or one up, one right, one up, one right – the actual distance traveled is the same.

First, the code for the types we're going to work with:

```
import Foundation

struct Point {
    let x: Double
    let y: Double
}

enum DistanceTechnique {
    case euclidean
    case euclideanSquared
    case manhattan
}
```

I've created my own **`Point`** class to avoid relying on **`CGPoint`** and thus Core Graphics. We're going to create three functions, each of which will be nested inside one parent function. The point of this chapter isn't to explain distance calculation, so let's get them out of the way quickly:

```
func calculateEuclideanDistanceSquared(start: Point, end:
```

```
Point) -> Double {
    let deltaX = start.x - end.x
    let deltaY = start.y - end.y

    return deltaX * deltaX + deltaY * deltaY
}

func calculateEuclideanDistance(start: Point, end: Point) ->
Double {
    return sqrt(calculateEuclideanDistanceSquared(start: start,
end: end))
}

func calculateManhattanDistance(start: Point, end: Point) ->
Double {
    return abs(start.x - end.x) + abs(start.y - end.y)
}
```

The first function, **`calculateEuclideanDistanceSquared()`**, uses Pythagoras's theorem to calculate the straight line distance between two points. If it's been a while since you were at school, this function considers the X and Y delta between two points to be two edges of a triangle, then calculates the hypotenuse of that triangle to be the distance between the two points – $A^2 + B^2 = C^2$.

The second function, **`calculateEuclideanDistance()`** builds on the **`calculateEuclideanDistanceSquared()`** function by calculating the square root of the result to give the true distance. If you need to calculate distances very frequently, e.g. every time the user's finger moves, having this call to **`sqrt()`** might become a performance liability, which is why the **`calculateEuclideanDistanceSquared()`** function is there.

Finally, the third function is **`calculateManhattanDistance()`**, which calculates the sum of the absolute distances between the two points' X and Y coordinates, as if you were in a taxi cab moving through a city built around square blocks.

With those three nested functions in place, it's now just a matter of choosing the correct option based on the technique that was requested:

```
switch technique {
case .euclidean:
   return calculateEuclideanDistance(start: start, end: end)
case .euclideanSquared:
   return calculateEuclideanDistanceSquared(start: start, end:
end)
case .manhattan:
   return calculateManhattanDistance(start: start, end: end)
}
```

That's it! Here's the complete code:

```
import Foundation

struct Point {
   let x: Double
   let y: Double
}

enum DistanceTechnique {
   case euclidean
   case euclideanSquared
   case manhattan
}

func calculateDistance(start: Point, end: Point, technique:
DistanceTechnique) -> Double {
   func calculateEuclideanDistanceSquared(start: Point, end:
Point) -> Double {
      let deltaX = start.x - end.x
      let deltaY = start.y - end.y
```

```
        return deltaX * deltaX + deltaY * deltaY
    }

    func calculateEuclideanDistance(start: Point, end: Point) ->
Double {
        return sqrt(calculateEuclideanDistanceSquared(start:
start, end: end))
    }

    func calculateManhattanDistance(start: Point, end: Point) ->
Double {
        return abs(start.x - end.x) + abs(start.y - end.y)
    }

    switch technique {
    case .euclidean:
        return calculateEuclideanDistance(start: start, end: end)
    case .euclideanSquared:
        return calculateEuclideanDistanceSquared(start: start,
end: end)
    case .manhattan:
        return calculateManhattanDistance(start: start, end: end)
    }
}

let distance = calculateDistance(start: Point(x: 10, y: 10),
end: Point(x: 100, y: 100), technique: .euclidean)
```

Now, all that code is perfectly valid, but it's also more verbose than it needs to be. As a reminder, functions are just named closures and so they capture any values from their enclosing function.

In this context, it means we don't need to make the three nested functions accept any

parameters, because they are identical to the parameters accepted by the enclosing functions – if we remove them, they'll just be automatically captured. This helps make our intention clearer: these nested functions are just different ways of operating on the same data rather than using specific values.

Here's the code to rewrite the **`calculateDistance()`** function so that it eliminates parameters from its nested functions and relies on capturing instead:

```
func calculateDistance(start: Point, end: Point, technique:
DistanceTechnique) -> Double {
    func calculateEuclideanDistanceSquared() -> Double {
        let deltaX = start.x - end.x
        let deltaY = start.y - end.y

        return deltaX * deltaX + deltaY * deltaY
    }

    func calculateEuclideanDistance() -> Double {
        return sqrt(calculateEuclideanDistanceSquared())
    }

    func calculateManhattanDistance() -> Double {
        return abs(start.x - end.x) + abs(start.y - end.y)
    }

    switch technique {
    case .euclidean:
        return calculateEuclideanDistance()
    case .euclideanSquared:
        return calculateEuclideanDistanceSquared()
    case .manhattan:
        return calculateManhattanDistance()
    }
}
```

Returning nested functions

Nested functions are automatically restricted to their enclosing function unless you specify otherwise, i.e. if you return them. Remember, functions are first-class data types in Swift, so you can use one function to return another based on specific criteria. In our case, we could make the **`calculateDistance()`** function into **`createDistanceAlgorithm()`**, which accepts only a technique parameter and returns one of its three nested functions depending on which technique was requested.

I know it's obvious, but it bears repeating that when you use this approach your nested function stops being private – it gets sent back as a return value for anyone to use.

Here's how to rewrite **`calculateDistance()`** so that it returns one of three functions:

```
func createDistanceAlgorithm(technique: DistanceTechnique) ->
(Point, Point) -> Double {
   func calculateEuclideanDistanceSquared(start: Point, end:
Point) -> Double {
      let deltaX = start.x - end.x
      let deltaY = start.y - end.y

      return deltaX * deltaX + deltaY * deltaY
   }

   func calculateEuclideanDistance(start: Point, end: Point) ->
Double {
      return sqrt(calculateEuclideanDistanceSquared(start:
start, end: end))
   }

   func calculateManhattanDistance(start: Point, end: Point) ->
Double {
      return abs(start.x - end.x) + abs(start.y - end.y)
   }
```

```
    switch technique {
    case .euclidean:
        return calculateEuclideanDistance
    case .euclideanSquared:
        return calculateEuclideanDistanceSquared
    case .manhattan:
        return calculateManhattanDistance
    }
}
```

Note that each of the three functions needs to accept parameters now, because they will be called later on like this:

```
let distanceAlgorithm =
createDistanceAlgorithm(technique: .euclidean)
let distance = distanceAlgorithm(Point(x: 10, y: 10), Point(x:
100, y: 100))
```

Documentation markup

Swift has special syntax that lets you embed Markdown-formatted text into your source code, which gets parsed by Xcode and displayed in the Quick Help system pane – press Alt+Cmd+2 while coding to bring it up on the right of the Xcode window. Using specially formatted code comments, you can document what parameters should be passed in, what the return value will contain, any errors that can be thrown, and more.

This documentation is *not* the same as the regular inline comments you add to particular code. These special comments are placed before your functions and classes and are used to show information in the Quick Help pane, as well as in the autocomplete popup, and are formatted so that both humans and Xcode can read them.

Let's get the easy stuff out of the way: unless you use one of the special keywords covered later, everything you write in a Markdown comment is shown as description text in the Quick Help pane. If you just start typing text, what you write will be used as a brief description in the autocomplete popup. Xcode can usually fit 20-30 words in the autocomplete space, but that's too long for real world use – aim for about 10 words of concise description.

Markdown comments start with **`/**`** and end with **`*/`**, like this:

```
/**
Call this function to grok some globs.
*/
func myGreatFunction() {
    // do stuff
}
```

In this text you can use a selection of Markdown formatting, as shown below:

```
Place text in `backticks` to mark code; on your keyboard these
usually share a key with tilde, ~.
* You can write bullets by starting with an asterisk then a
space.
    * Indent your asterisks to create sublists
1. You can write numbered listed by starting with 1.
```

```
1. Subsequent items can also be numbered 1. and Xcode will
renumber them automatically.

If you want to write a link, [place your text in brackets](and
your link in parentheses)
# Headings start with a # symbol
## Subheadings start with ##
### Sub-subheadings start with ### and are the most common
heading style you'll come across
Write a *single asterisk* around words to make them italic
Write **two asterisks** around words to make them bold
```

Documentation keywords

Beyond using text to describe your functions, Swift lets you add special keywords that get displayed in the Quick Help pane. There are quite a few of these, but most do nothing other than show a title and some text in Quick Help. There are six that I generally recommend as being useful, and you can learn them in a few minutes.

First: the **`Returns`** keyword lets you specify what value the caller can expect back when the function runs successfully. Remember that autocomplete already reports the data type of the return value, so this field is there to describe what the data actually means – we know it's a string, but how will it be formatted?

```
- Returns: A string containing a date formatted as RFC-822
```

Next is the **`Parameter`** keyword. This lets you specify the name of a parameter and describe what it contains. Again, autocomplete will say what data type must be used, so this is your chance to provide some detail: "The name of a Taylor Swift album". You can include as many **`Parameter`** lines as you have parameters.

```
- Parameter album: The name of a Taylor Swift album
- Parameter track: The track number to load
```

Third is the **`Throws`** keyword, which lets you specify a comma-separated list of the error types that can be thrown by the function:

```
- Throws: LoadError.networkFailed, LoadError.writeFailed
```

Fourth is **`Precondition`**, which should be used to describe the correct state of your program before your function is called. If you're using pure functions, this precondition ought to rely only on the parameters being passed into the function, for example **`inputArray.count > 0`**:

```
- Precondition: inputArray.count > 0
```

Fifth is **`Complexity`**, which is popular in the Swift standard library. This isn't formatted specially in Quick Help, but it's useful information for others working with your code. This should be written using Big O notation, for example:

```
- Complexity: O(1)
```

Finally there is the **`Authors`** keyword, which sounds useful at first but I remain dubious. As you might imagine, this is used to write the name of a function's author into the Quick Help pane, which is helpful when you need to figure out who to complain to and/or praise for their work. But because Xcode places **`Authors`** above **`Returns`**, **`Throws`**, and **`Parameter`**, adding a credit just pushes down the actually important fields. Give it a try and see what you think, but remember that documentation is there to be *useful* first.

```
- Authors: Paul Hudson
```

If you include more freeform text between the documentation keywords, it will just be flowed into the correct position in Quick Help.

Chapter 2
Types

Matt Gallagher, author of CocoaWithLove.com

My favorite Swift one-liner is downcasting and filtering an array using **`flatMap()`**:

```
let myCustomViews = allViews.flatMap { $0 as? MyCustomView }
```

The line looks simple but it is packed with great Swift features that are most apparent if you compare with the closest out-of-the-box equivalent in Objective-C:

```
NSArray<MyCustomView *> *myCustomViews = (NSArray<MyCustomView
*> *)[allViews filteredArrayUsingPredicate:
   [NSPredicate predicateWithBlock:^BOOL(id _Nonnull
evaluatedObject, NSDictionary<NSString *,id> * _Nullable
bindings) {
      return [evaluatedObject isKindOfClass:[MyCustomView
class]];
   }]
];
```

Useful initializers

Understanding how initializers work isn't easy in Swift, but it's also something you probably learned quite a while ago so I'm not going to go over it again here. Instead, I want to focus on some interesting initializers that might help you use common Swift types more efficiently.

Repeating values

Easily my favorite initializer for strings and arrays is **`repeating:count:`**, which lets you create large values quickly. For example, you can create headings in the Markdown text format by writing equals signs underneath some text, like this:

```
This is a heading
=================
```

Markdown is a useful format because it can be parsed by computers while also being visually attractive to humans, and that underlining gives us a great example for **`repeating:count:`**. To use this initializer, give it a string for its first parameter and the number of times to repeat for its second parameter, like this:

```
let heading = "This is a heading"
let underline = String(repeating: "=", count:
heading.characters.count)
```

You can do the same for arrays like this:

```
let equalsArray = [String](repeating: "=", count:
heading.characters.count)
```

This array initializer is flexible enough that you can use it to create multi-dimensional arrays very easily. For example, this creates a 10x10 array ready to be filled:

```
var board = [[String]](repeating: [String](repeating: "",
count: 10), count: 10)
```

Converting to and from numbers

It hurts my brain when I see code like this:

```
let str1 = "\(someInteger)"
```

It's wasteful and unnecessary, and yet string interpolation is such a neat feature that you'd be forgiven for using it – and in fact I'm pretty sure I've used it a few times without thinking!

But Swift has a simple, better way to create a string from an integer using an initializer:

```
let str2 = String(someInteger)
```

Things are slightly harder when converting the other way, because you might try to pass in an invalid number, like this:

```
let int1 = Int("elephant")
```

So, this initializer returns **`Int?`**: if you gave it a valid number you'll get back an **`Int`**, otherwise you'll get back nil.

If you don't want an optional, you should unwrap the result:

```
if let int2 = Int("1989") {
    print(int2)
}
```

Alternatively, use the nil coalescing operator (**`??`**) to provide a sensible default, like this:

```
let int3 = Int("1989") ?? 0
print(int3)
```

Swift has variations on these two initializers that deal with variable radices. For example, if you want to work with hexadecimal (base 16), you can ask Swift to give you the string representation of a number in hex like this:

```
let str3 = String(28, radix: 16)
```

That will set **str3** to "1c". If you preferred "1C" – i.e., uppercase – try this instead:

```
let str4 = String(28, radix: 16, uppercase: true)
```

To convert that back into an integer - remembering that it's optional! – use this:

```
let int4 = Int("1C", radix: 16)
```

Unique arrays

If you have an array with repeated values and you want to find a fast way to remove duplicates, you're looking for **Set**. This is a built-in data type that has initializers to and from generic arrays, which means you can de-dupe an array quickly and efficiently just using initializers:

```
let scores = [5, 3, 6, 1, 3, 5, 3, 9]
let scoresSet = Set(scores)
let uniqueScores = Array(scoresSet)
```

That's all it takes – it's no wonder I'm such a big fan of sets!

Dictionary capacities

One simple initializer to end on: if you're adding items individually to a dictionary but know how many items you intend to add overall, create your dictionary using the **minimumCapacity:** initializer like this:

```
var dictionary = Dictionary<String, String>(minimumCapacity:
100)
```

This helps Swift optimize execution by allocating enough space up front. Note: behind the scenes, Swift's dictionaries increase their capacity in powers of 2, so when you request a non-power-of-2 number like 100 you will actually get back a dictionary with a minimum capacity of 128. Remember, this is a *minimum* capacity – if you add more objects, it's not a problem.

Enums

I already talked about enum associated values in the pattern matching chapter, but here I want to focus on enums themselves because they are deceptively powerful.

Let's start with a very simple enum to track some basic colors:

```
enum Color {
    case unknown
    case blue
    case green
    case pink
    case purple
    case red
}
```

If you prefer, you can write all the cases on one line, like this:

```
enum Color {
    case unknown, blue, green, pink, purple, red
}
```

For testing purposes, let's wrap that in a simple struct that represents toys:

```
struct Toy {
    let name: String
    let color: Color
}
```

Swift's type inference can see that the **`color`** property of a **`Toy`** is a **`Color`** enum, which means you don't need to write **`Color.blue`** when creating a toy. For example, we can create two toys like this:

```
let barbie = Toy(name: "Barbie", color: .pink)
let raceCar = Toy(name: "Lightning McQueen", color: .red)
```

Raw values

Let's start off with raw values: a data type that underlies each enum case. Enums don't have a raw value by default, so if you want one you need to declare it. For example, we could give our colors an integer raw value like this:

```
enum Color: Int {
    case unknown, blue, green, pink, purple, red
}
```

Just by adding `: Int` Swift has given each color a matching integer, starting from 0 and counting upwards. That is, Unknown is equal to 0, Blue is equal to 1, and so on. Sometimes the default values aren't useful to you, so you can specify individual integers for every raw value if you want. Alternatively, you can specify a different starting point to have Xcode count upwards from there.

For example, we could create an enum for the four inner planets of our solar system like this:

```
enum Planet: Int {
    case mercury = 1
    case venus
    case earth
    case mars
    case unknown
}
```

By specifically giving Mercury the value of 1, Xcode will count upwards from there: Venus is 2, Earth is 3, and Mars is 4.

Now that the planets are numbered sensibly, we can find the raw value of any planet like this:

```
let marsNumber = Planet.mars.rawValue
```

Going the other way isn't quite so easy: yes, you can create a `Planet` enum from a number now that we have raw values, but doing so creates an *optional* enum. This is because you could

try to create a planet with the raw value 99, which doesn't exist – at least not *yet*.

Fortunately, I added an Unknown case to the planets for times when an invalid planet number has been requested, so we can use nil coalescing to create a **`Planet`** enum from its raw value with a sensible default:

```
let mars = Planet(rawValue: 556) ?? Planet.unknown
```

Having numbers is fine for planets, but when it comes to colors you might find it easier to use strings instead. Unless you have very specific needs, just specifying **`String`** as the raw data type for your enum is enough to give them meaningful names – Swift automatically maps your enum name to a string. For example, this will print "Pink":

```
enum Color: String {
    case unknown, blue, green, pink, purple, red
}

let pink = Color.pink.rawValue
print(pink)
```

Regardless of the data type of your raw values, or even if you have one or not, Swift will automatically stringify your enums when they are used as part of string interpolation. Used in this way doesn't make them a string, though, so if you want to call any string methods you need to create a string from them. For example:

```
let barbie = Toy(name: "Barbie", color: .pink)
let raceCar = Toy(name: "Lightning McQueen", color: .red)

// regular string interpolation
print("The \(barbie.name) toy is \(barbie.color)")

// get the string form of the Color then call a method on it
print("The \(barbie.name) toy is \
(barbie.color.rawValue.uppercased())")
```

Computed properties and methods

Enums aren't quite as powerful as structs and classes, but they do let you encapsulate some useful functionality inside them. For example, you can't give enums stored properties unless they are static because doing so wouldn't make sense, but you can add *computed* properties that return a value after running some code.

To put this in some useful context for you, let's add a computed property to the **`Color`** enum that prints a brief description of the color. Apologies ahead of time for mixing Terry Pratchett, Prince and Les Misérables references in one piece of code:

```
enum Color {
    case unknown, blue, green, pink, purple, red

    var description: String {
        switch self {
        case .unknown:
            return "the color of magic"
        case .blue:
            return "the color of the sky"
        case .green:
            return "the color of grass"
        case .pink:
            return "the color of carnations"
        case .purple:
            return "the color of rain"
        case .red:
            return "the color of desire"
        }
    }
}

let barbie = Toy(name: "Barbie", color: .pink)
print("This \(barbie.name) toy is \(barbie.color.description)")
```

Of course, computed properties are just syntactic sugar around good old methods, so it should come as no surprise that you can add methods directly to enums too. Let's do that now by adding two new methods to the **`Color`** enum, **`forBoys()`** and **`forGirls()`**, to determine whether a toy is for girls or boys based on its color – just add this below the **`description`** computed property we just added:

```
func forBoys() -> Bool {
    return true
}

func forGirls() -> Bool {
    return true
}
```

In case you were wondering, deciding which toys are for boys or girls based on their color is a bit 1970s: those functions both return true for a reason!

So: our enum now has a raw value, a computed property and some methods. I hope you can see why I described enums as "deceptively powerful" – they can do a lot!

Arrays

Arrays are one of the true workhorses of Swift. Sure, they are important in most apps, but their use of generics makes them type-safe while adding some useful features. I'm not going to go over their basic usage; instead, I want to walk you through some useful methods you might not know.

First: sorting. As long as the element type your array stores adopts the **`Comparable`** protocol, you get the methods **`sorted()`** and **`sort()`** – the former returns a sorted array, whereas the latter modifies the array you call it on. If you don't adopt the **`Comparable`** protocol, you can use alternate versions of **`sorted()`** and **`sort()`** that let you specify how items should be sorted.

To demonstrate the examples below we'll be using these two arrays:

```
var names = ["Taylor", "Timothy", "Tyler", "Thomas", "Tobias",
"Tabitha"]
let numbers = [5, 3, 1, 9, 5, 2, 7, 8]
```

To sort the **`names`** array alphabetically, use either **`sort()`** or **`sorted()`** depending on your needs:

```
let sorted = names.sorted()
```

Once that code runs, **`sorted`** will contain **`["Tabitha", "Taylor", "Thomas", "Timothy", "Tobias", "Tyler"]`**.

If you want to write your own sorting function – which is required if you don't adopt **`Comparable`** but optional otherwise – write a closure that accepts two strings and returns true if the first string should be ordered before the second.

For example, we could write a string sorting algorithm that behaves identically to a regular alphabetical sort, with the exception that it always places the name "Taylor" at the front. It's what Ms Swift would want, I'm sure:

```
names.sort {
    print("Comparing \($0) and \($1)")
```

```
    if ($0 == "Taylor") {
        return true
    } else if $1 == "Taylor" {
        return false
    } else {
        return $0 < $1
    }
}
```

That code uses **`sort()`** rather than **`sorted()`**, which causes the array to be sorted in place rather than returning a new, sorted array. I also added a **`print()`** call in there so you can see exactly how **`sort()`** works. Here's the output:

```
Comparing Timothy and Taylor
Comparing Tyler and Timothy
Comparing Thomas and Tyler
Comparing Thomas and Timothy
Comparing Thomas and Taylor
Comparing Tobias and Tyler
Comparing Tobias and Timothy
Comparing Tabitha and Tyler
Comparing Tabitha and Tobias
Comparing Tabitha and Timothy
Comparing Tabitha and Thomas
Comparing Tabitha and Taylor
```

As you can see, names can appear as **`$0`** or **`$1`** as the algorithm progresses, which is why I compare against both possibilities in the custom sort function.

Sorting is easy enough, but adopting **`Comparable`** also enables two more useful methods: **`min()`** and **`max()`**. Just like **`sort()`** these have alternatives that accept a closure if you don't adopt **`Comparable`**, but the code is identical because the operation is identical: should item A appear before item B?

Using the **`numbers`** array from earlier, we can find the highest and lowest number in the array

in two lines of code:

```
let lowest = numbers.min()
let highest = numbers.max()
```

For strings, **`min()`** returns whichever string comes first after sorting, and **`max()`** returns whichever string comes last. If you try re-using the same closure I gave for custom sorting, including the **`print()`** statement, you'll see that **`min()`** and **`max()`** are actually more efficient than using **`sort()`** because they don't need to move every item.

Conforming to Comparable

Working with **`sort()`**, **`min()`**, and **`max()`** is easy enough for basic data types like strings and integers. But how do you sort something else entirely, like types of cheese or breeds of dog? I've already shown you how to write custom closures, but that approach becomes cumbersome if you have to sort more than once – you end up duplicating code, which creates a maintenance nightmare.

The smarter solution is to implement the **`Comparable`** protocol, which in turn requires you to use operator overloading. We'll get into more detail on that later, but for now I want to show you just enough to get **`Comparable`** working. First, here's a basic **`Dog`** struct that holds a couple of pieces of information:

```
struct Dog {
    var breed: String
    var age: Int
}
```

For the purposes of testing, we'll create three dogs and group them in an array:

```
let poppy = Dog(breed: "Poodle", age: 5)
let rusty = Dog(breed: "Labrador", age: 2)
let rover = Dog(breed: "Corgi", age: 11)
var dogs = [poppy, rusty, rover]
```

Because the **`Dog`** struct doesn't conform to **`Comparable`**, we don't get simple **`sort()`** and **`sorted()`** methods on the **`dogs`** array, we only get ones that require a custom closure to run.

To begin the process of making **`Dog`** adopt **`Comparable`** we need to declare it as conforming, like this:

```
struct Dog: Comparable {
    var breed: String
    var age: Int
}
```

You'll get errors now, and that's OK.

The next step is what confuses people when they try it for the first time: you need to implement two new functions, but they have unusual names that take a little getting used to when working with operator overloading, which is what we need to do.

Add these two functions inside the **`Dog`** struct:

```
static func <(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.age < rhs.age
}

static func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.age == rhs.age
}
```

To be clear, your code should look like this:

```
struct Dog: Comparable {
    var breed: String
    var age: Int

    static func <(lhs: Dog, rhs: Dog) -> Bool {
        return lhs.age < rhs.age
    }
```

```
    static func ==(lhs: Dog, rhs: Dog) -> Bool {
        return lhs.age == rhs.age
    }
}
```

The function names are unusual if you haven't worked with operator overloading before, but I hope you can see exactly what they do: the **<** function is used when you write **`dog1 < dog2`** and the **==** function is used when you write **`dog1 == dog2`**.

Those two steps are enough to implement **`Comparable`** fully, so you can now sort the **`dogs`** array easily:

```
dogs.sort()
```

Adding and removing items

You're almost certainly already using the **`append()`**, **`insert()`** and **`remove(at:)`** methods of arrays, but I want to make sure you're aware of other ways to add and remove items.

If you want to add two arrays together, you can use **+**, or **+=** to add in place. For example:

```
let poppy = Dog(breed: "Poodle", age: 5)
let rusty = Dog(breed: "Labrador", age: 2)
let rover = Dog(breed: "Corgi", age: 11)
var dogs = [poppy, rusty, rover]

let beethoven = Dog(breed: "St Bernard", age: 8)
dogs += [beethoven]
```

When it comes to removing items, there are two interesting ways of removing the last item: **`removeLast()`** and **`popLast()`**. They both remove the final item in an array and return it to you, but **`popLast()`** is optional whereas **`removeLast()`** is not. Think that through for a moment: **`dogs.removeLast()`** must return an instance of the **`Dog`** struct. What happens if

the array is empty? Well, the answer is "bad things" – your app will crash.

If it's even slightly possible your array might be empty when you try to remove an item, use **`popLast()`** instead so you can safely check the return value:

```
if let dog = dogs.popLast() {
    // do stuff with `dog`
}
```

Note: **`removeLast()`** has a counterpart called **`removeFirst()`** to remove and return the initial item in an array. Sadly, **`popLast()`** has no equivalent.

Emptiness and capacity

Here are two more small tips I want to demonstrate: **`isEmpty`** and **`reserveCapacity()`**. The first of these, **`isEmpty`**, returns true if the array has no items added to it. This is both shorter and more efficient than using **`someArray.count == 0`**, but for some reason gets less use.

The **`reserveCapacity()`** method lets you tell iOS how many items you intend to store in the array. This isn't a hard limit – i.e., if you reserve a capacity of 10 you can go ahead and store 20 if you want – but it does allow iOS to optimize object storage by ensuring you have enough space for your suggested capacity.

Warning: using **`reserveCapacity()`** is *not* a free operation. Behind the scenes, Swift will create a new array that contains the same values, with space for the capacity you asked for. It will *not* just extend the existing array. The reason for this is that the method guarantees the resulting array will have contiguous storage (i.e., that all items are stored next to each other rather than scattered around RAM), so Swift will do a lot of moving around. This applies even if you've already called **`reserveCapacity()`** – try putting this code into a playground to see for yourself:

```
import Foundation

let start = CFAbsoluteTimeGetCurrent()
```

```
var array = Array(1...1000000)
array.reserveCapacity(1000000)
array.reserveCapacity(1000000)

let end = CFAbsoluteTimeGetCurrent() - start
print("Took \(end) seconds")
```

When that code runs, you'll see a hefty pause both times **`reserveCapacity()`** is called. Because **`reserveCapacity()`** is an O(n) call (where "n" is the **`count`** value of the array), you should call it before adding items to the array.

Contiguous arrays

Swift offers two main kinds of arrays, but nearly always you find only one is used. First, let's unpick the syntax a little: you should know that these two lines of code are functionally identical:

```
let array1 = [Int]()
let array2 = Array<Int>()
```

The first line is syntactic sugar for the second. So far, so easy. But I want to introduce you to the importance of the **`ContiguousArray`** container, which looks like this:

```
let array3 = ContiguousArray<Int>(1...1000000)
```

That's it. Contiguous arrays have all the properties and methods you're used to – **`count`**, **`sort()`**, **`min()`**, **`map()`**, and more – but because all items are guaranteed to be stored contiguously (i.e., next to each other) you can get increased performance.

Apple's official documentation says to use **`ContiguousArray`** when "you need C array performance", whereas you should regular **`Array`** when you want something that is "optimized for efficient conversions from Cocoa and back." The documentation goes on to say that the performance of **`Array`** and **`ContiguousArray`** is identical when used with non-class types, which means you will definitely get a performance improvement when working

with classes.

The reason for this is quite subtle: Swift arrays can be bridged to **NSArray**, which was the array type used by Objective-C developers. For historical reasons, **NSArray** was unable to store value types, such as integers, unless they were wrapped inside an object. So, the Swift compiler can be clever: if you make a regular Swift array containing value types, it knows you can't try to bridge that to an **NSArray**, so it can perform extra optimizations to increase performance.

That being said, I've found **ContiguousArray** to be faster than **Array** no matter what, even with basic types such as **Int**. To give you a trivial example, the code below adds up all the numbers from 1 to 1 million:

```
let array2 = Array<Int>(1...1000000)
let array3 = ContiguousArray<Int>(1...1000000)

var start = CFAbsoluteTimeGetCurrent()
array2.reduce(0, +)
var end = CFAbsoluteTimeGetCurrent() - start
print("Took \(end) seconds")

start = CFAbsoluteTimeGetCurrent()
array3.reduce(0, +)
end = CFAbsoluteTimeGetCurrent() - start
print("Took \(end) seconds")
```

When I ran that code, **Array** took 0.25 seconds and **ContiguousArray** took 0.13 seconds. It's not vastly superior given that we just looped over a million elements, but if you want that extra jolt of performance in your app or game you should definitely give **ContiguousArray** a try.

Sets

Understanding the difference between sets and arrays – and knowing when each one is the right choice – is an important skill in any Swift developer's toolbox. Sets can be thought of as unordered arrays that cannot contain duplicate elements. If you add the same element more than once, it will appear only once in the set. The combination of lacking duplicates and not tracking order allows sets to be significantly faster than arrays, because items are stored according to a hash rather than an incrementing integer index.

To put this into context, checking to see whether an array contains an item has the complexity O(n), which means "it depends on how many elements you have in the array." This is because **`Array.contains()`** needs to check every element from 0 upwards, so if you have 50 elements it needs to do 50 checks. Checking whether a *set* contains an item has the complexity O(1), which means "it will always run at the same speed no matter how many elements you have." This is because sets work like dictionaries: a key is generated by creating a hash of your object, and that key points directly to where the object is stored.

The basics

The best way to experiment with sets is using a playground. Try typing this in:

```
var set1 = Set<Int>([1, 2, 3, 4, 5])
```

When that runs, you'll see "{5, 2, 3, 1, 4}" in the output window. Like I said, sets are unordered so you might see something different in your Xcode window.

That creates a new set from an array, but you can create them from ranges too, just like arrays:

```
var set2 = Set(1...100)
```

You can also add items to them individually, although the method is named **`insert()`** rather than **`append()`** to reflect its unordered nature:

```
set1.insert(6)
set1.insert(7)
```

To check whether an item exists in your set, use the lightning fast **`contains()`** method:

```
if set1.contains(3) {
    print("Number 3 is in there!")
}
```

And use **`remove()`** to remove items from the set:

```
set1.remove(3)
```

Arrays and sets

Arrays and sets work well when used together, so it's no surprise they are almost interchangeable. First, both arrays and sets have constructors that take the other type, like this:

```
var set1 = Set<Int>([1, 2, 3, 4, 5])
var array1 = Array(set1)
var set2 = Set(array1)
```

In fact, converting an array into a set and back is the fastest way to remove all its duplicates, and it's just two lines of code.

Second, several methods of sets return arrays rather than sets because doing so is more useful. For example, the **`sorted()`**, **`map()`** and **`filter()`** methods on sets return an array.

So, while you can loop over sets directly like this:

```
for number in set1 {
    print(number)
}
```

…you can also sort the set into a sensible order first, like this:

```
for number in set1.sorted() {
    print(number)
}
```

Sets, like arrays, have the method **`removeFirst()`** to remove an item from the front of the set. It's use is different, though: because sets are unordered you really have no idea what that first item is going to be, so **`removeFirst()`** effectively means "give me any object so I can process it." Cunningly, sets have a **`popFirst()`** method, which arrays do not – I really wish I knew why!

Set operations

Sets come with a number of methods that allow you to manipulate them in interesting ways. For example, you can create the union of two sets – i.e. the merger of two sets – like this:

```
let spaceships1 = Set(["Serenity", "Nostromo", "Enterprise"])
let spaceships2 = Set(["Voyager", "Serenity", "Executor"])

let union = spaceships1.union(spaceships2)
```

When that code runs, **`union`** will contain five items because the duplicate "Serenity" will appear only once.

Two other useful set operations are **`intersection()`** and **`symmetricDifference()`**. The former returns a new set that contains only elements that exist in both sets, and the latter does the opposite: it returns only elements that do not exist in both. In code it looks like this:

```
let intersection = spaceships1.intersection(spaceships2)
let difference = spaceships1.symmetricDifference(spaceships2)
```

When that runs, **`intersection`** will contains Serenity and **`difference`** will contains Nostromo, Enterprise, Voyager and Executor.

Note: **`union()`**, **`intersection()`**, and **`symmetricDifference()`** all have in-place alternatives that modify the set directly, called by adding **`form`** to the method. So, **`formUnion()`**, **`formIntersection()`**, and **`formSymmetricDifference()`**.

Sets have several query methods that return true or false depending on what you provide them.

These methods are:

- **`A.isSubset(of: B)`**: returns true if all of set A's items are also in set B.
- **`A.isSuperset(of: B)`**: returns true if all of set B's items are also in set A.
- **`A.isDisjoint(with: B)`**: returns true if none of set B's items are also in set A.
- **`A.isStrictSubset(of: B)`**: returns true if all of set A's items are also in set B, but A and B are not equal
- **`A.isStrictSuperset(of: B)`**: returns true if all of set B's items are also in set A, but A and B are not equal

Sets distinguish between subsets and strict (or "proper") subsets, with the difference being that the latter necessarily excludes identical sets. That is, Set A is a subset of Set B if every item in Set A is also in Set B. On the other hand, Set A is a strict subset of Set B if every element in Set A is also in Set B, but Set B contains at least one item that is missing from Set A.

The code below demonstrates them all individually, and I've marked each method's return value in a comment:

```
let spaceships1 = Set(["Serenity", "Nostromo", "Enterprise"])
let spaceships2 = Set(["Voyager", "Serenity", "Star
Destroyer"])
let spaceships3 = Set(["Galactica", "Sulaco", "Minbari"])
let spaceships1and2 = spaceships1.union(spaceships2)

spaceships1.isSubset(of: spaceships1and2) // true
spaceships1.isSubset(of: spaceships1) // true
spaceships1.isSubset(of: spaceships2) // false
spaceships1.isStrictSubset(of: spaceships1and2) // true
spaceships1.isStrictSubset(of: spaceships1) // false

spaceships1and2.isSuperset(of: spaceships2) // true
spaceships1and2.isSuperset(of: spaceships3) // false
spaceships1and2.isStrictSuperset(of: spaceships1) // true

spaceships1.isDisjoint(with: spaceships2) // false
```

NSCountedSet

The Foundation library has a specialized set called **`NSCountedSet`**, and it is a set with a twist: items can still appear only once, but if you try to add them more than once it will keep track of the count as if they were actually there. This means you get all the speed of non-duplicated sets, but you also get the ability to count how many times items would appear if duplicates were allowed.

You can create an **`NSCountedSet`** from Swift arrays or sets depending on your need. In the example below I create one large array (with a duplicated item), add it all to the counted set, then print out the counts for two values:

```
import Foundation

var spaceships = ["Serenity", "Nostromo", "Enterprise"]
spaceships += ["Voyager", "Serenity", "Star Destroyer"]
spaceships += ["Galactica", "Sulaco", "Minbari"]

let countedSet = NSCountedSet(array: spaceships)

print(countedSet.count(for: "Serenity")) // 2
print(countedSet.count(for: "Sulaco")) // 1
```

As you can see, you can use **`count(for:)`** to retrieve how many times (in theory) an element appears in the counted set. You can pull out an array of all objects using the **`countedSet.allObjects`** property, but be warned: **`NSCountedSet`** does *not* support generics, so you'll need to typecast it back to **`[String]`**.

Tuples

Tuples are like simplified, anonymous structs: they are value types that carry distinct fields of information, but they don't need to be formally defined. This lack of formal definition makes them easy to create and throw away, so they are commonly used when you need a function to return multiple values.

In the chapters on pattern matching and destructuring, I covered how tuples are used in other ways – they really are pervasive little things. How pervasive? Well, consider this code:

```
func doNothing() { }
let result = doNothing()
```

Consider this: what data type does the **`result`** constant have? As you might have guessed given the name of this chapter, it's a tuple: **`()`**. Behind the scenes, Swift maps the **`Void`** data type (the default for functions with no explicit return type) to an empty tuple.

Now consider this: every type in Swift – integers, strings, and so on – is effectively a single-element tuple of itself. Take a look at the code below:

```
let int1: (Int) = 1
let int2: Int = (1)
```

That code is perfectly correct: assigning an integer to a single-element tuple and assigning a single-element tuple to an integer both do exactly the same thing. As the Apple documentation says, "if there is only one element inside the parentheses, the type [of a tuple] is simply the type of that element." They are literally the same thing, so you can even write this:

```
var singleTuple = (value: 42)
singleTuple = 69
```

When Swift compiles the first line, it basically ignores the label and makes it a single-element tuple containing an integer – which is in turn identical to an integer. In practice, this means you can't add labels to single-element tuples – if you try to force a data type, you'll get an error:

```
var thisIsAllowed = (value: 42)
```

```
var thisIsNot: (value: Int) = (value: 42)
```

So, if you return nothing from a function you get a tuple, if you return several values from a function you get a tuple, and if you return a single value you effectively also get a tuple. I think it's safe to say you're already a frequent tuple user whether you knew it or not!

Now, I'm going to cover a few interesting aspects of tuples below, but first you should know there are a couple of downsides to tuples. Specifically, you can't add methods to tuples or make them implement protocols – if this is what you want to do, you're looking for structs instead.

Tuples have types

It's easy to think of tuples as some sort of open dumping ground for data, but that's not true: they are strongly typed, just like everything else in Swift. This means you can't change the type of a tuple once it's been created – code like this will simply not compile:

```
var singer = ("Taylor", "Swift")
singer = ("Taylor", "Swift", 26)
```

If you don't name the elements of your tuple, you can access them using numbers counting from 0, like this:

```
var singer = ("Taylor", "Swift")
print(singer.0)
```

If you have tuples within tuples – which is not uncommon – you need to use **`0.0`** and so on, like this:

```
var singer = (first: "Taylor", last: "Swift", address: ("555
Taylor Swift Avenue", "No, this isn't real", "Nashville"))
print(singer.2.2) // Nashville
```

This is built-in behavior, but that doesn't mean it's recommended. You can – and usually should – name your elements so you can access them more sensibly:

```
var singer = (first: "Taylor", last: "Swift")
print(singer.last)
```

These names form part of the type, so code like this will not compile:

```
var singer = (first: "Taylor", last: "Swift")
singer = (first: "Justin", fish: "Trout")
```

Tuples and closures

You can't add methods to tuples, but you can add closures. The distinction is very fine, I agree, but it is important: adding a closure to a tuple is just like adding any other value, you're literally attaching the code as a data type to the tuple. Because it's not a method the declaration is a little different, but here's an example to get you started:

```
var singer = (first: "Taylor", last: "Swift", sing: { (lyrics:
String) in
    print("\(lyrics)")
})

singer.sing("Haters gonna hate")
```

Note: these closures can't access sibling elements, which means code like this won't work:

```
print("My name is \(first): \(lyrics)")
```

Returning multiple values

Tuples are commonly used to return multiple values from a function. In fact, if this was the only thing that tuples gave us, they'd still be a huge feature of Swift compared to other languages – including Objective-C.

Here's an example of a Swift function returning several values in a single tuple:

```
func fetchWeather() -> (type: String, cloudCover: Int, high:
```

```
Int, low: Int) {
    return ("Sunny", 50, 32, 26)
}

let weather = fetchWeather()
print(weather.type)
```

You don't have to name the elements, of course, but it is certainly good practice so that other developers know what to expect.

If you prefer to destructure the results of a tuple-returning function, that's easy to do too:

```
let (type, cloud, high, low) = fetchWeather()
```

In comparison, if Swift *didn't* have tuples then we'd have to rely on returning an array and typecasting as needed, like this:

```
import Foundation

func fetchWeather() -> [Any] {
    return ["Sunny", 50, 32, 26]
}

let weather = fetchWeather()
let weatherType = weather[0] as! String
let weatherCloud = weather[1] as! Int
let weatherHigh = weather[2] as! Int
let weatherLow = weather[3] as! Int
```

Or – worse – using **inout** variables, like this:

```
func fetchWeather(type: inout String, cloudCover: inout Int,
high: inout Int, low: inout Int) {
    type = "Sunny"
    cloudCover = 50
```

```
    high = 32
    low = 26
}

var weatherType = ""
var weatherCloud = 0
var weatherHigh = 0
var weatherLow = 0

fetchWeather(type: &weatherType, cloudCover: &weatherCloud,
high: &weatherHigh, low: &weatherLow)
```

Seriously: if **`inout`** is the answer, you're probably asking the wrong question.

Optional tuples

Tuples can contain optional elements, and you can also have optional tuples. That might sound similar, but the difference is huge: optional elements are individual items inside a tuple such as **`String?`** and **`Int?`**, whereas an optional tuple is where the whole structure may or may not exist.

A tuple with optional elements must exist, but its optional elements may be nil. An optional tuple must either have all its elements filled, or be nil. An optional tuple with optional elements may or may not exist, and each of its optional elements may or may not exist.

Swift can't use type interference when you work with optional tuples because each element in a tuple has its own type. So, you need to declare exactly what you want, like this:

```
let optionalElements: (String?, String?) = ("Taylor", nil)
let optionalTuple: (String, String)? = ("Taylor", "Swift")
let optionalBoth: (String?, String?)? = (nil, "Swift")
```

Broadly speaking, optional elements are common, optional tuples less so.

Comparing tuples

Swift lets you compare tuples up to arity six as long they have identical types. That means you can compare tuples that contain up to six items using **==**, and it will return true if all six items in one tuple match their counterpart in a second tuple.

For example, the code below will print "No match":

```
let singer = (first: "Taylor", last: "Swift")
let person = (first: "Justin", last: "Bieber")

if singer == person {
    print("Match!")
} else {
    print("No match")
}
```

Be warned, though: tuple comparison ignores element labels and focuses only on types, which can have unexpected results. For example, the code below will print "Match!" even though the tuple labels are different:

```
let singer = (first: "Taylor", last: "Swift")
let bird = (name: "Taylor", breed: "Swift")

if singer == bird {
    print("Match!")
} else {
    print("No match")
}
```

Typealias

You've seen how powerful, flexible, and useful tuples can be, but sometimes you will want to formalize things just a bit. To give you a Swift-themed example, consider these two tuples, representing Taylor Swift's parents:

```
let father = (first: "Scott", last: "Swift")
let mother = (first: "Andrea", last: "Finlay")
```

(No, I don't have a collection of Taylor Swift facts, but I *can* use Wikipedia!)

When they married, Andrea Finlay became Andrea Swift, and they became husband and wife. We could write a simple function to represent that event:

```
func marryTaylorsParents(man: (first: String, last: String),
woman: (first: String, last: String)) -> (husband: (first:
String, last: String), wife: (first: String, last: String)) {
    return (man, (woman.first, man.last))
}
```

Note: I've used "man" and "wife", and also had the wife take her husband's surname, because that's what happened with Taylor Swift's parents. Clearly this is only one type of marriage, and I hope you can understand that this is a simplified example, not a political statement.

The **`father`** and **`mother`** tuples seemed nice enough in isolation, but that **`marryTaylorsParents()`** function looks pretty grim. Repeating **`(first: String, last: String)`** again and again makes it hard to read and hard to change.

Swift's solution is simple: the **`typealias`** keyword. This is *not* specific to tuples, but it's certainly most useful here: it lets you attach an alternate name to a type. For example, we could create a **`typealias`** like this:

```
typealias Name = (first: String, last: String)
```

Using that, the **`marryTaylorsParents()`** function becomes significantly shorter:

```
func marryTaylorsParents(man: Name, woman: Name) -> (husband:
Name, wife: Name) {
    return (man, (woman.first, man.last))
}
```

Generics

Even though generics are an advanced topic in Swift, you use them all the time: **`[String]`** is an example of you using the **`Array`** structure to store strings, which is an example of generics. The truth is that *using* generics is straightforward, but *creating* them takes a little getting used to. In this chapter I want to demonstrate how (and why!) to create your own generics, starting with a function, then a struct, and finally wrapping a Foundation type.

Let's start with a simple problem that demonstrates what generics are and why they are important: we're going to create a very simple generic function.

Imagine a function that is designed to print out some debug information about a string. It might look like this:

```
func inspectString(_ value: String) {
    print("Received String with the value \(value)")
}

inspectString("Haters gonna hate")
```

Now let's create the same function that prints information about an integer:

```
func inspectInt(_ value: Int) {
    print("Received Int with the value \(value)")
}

inspectInt(42)
```

Now let's create the same function that prints information about a Double. Actually… let's not. This is clearly very boring code, and we'd need to extend it to floats, booleans, arrays, dictionaries and lots more. There's a smarter solution called generic programming, and it allows us to write functions that work with types that get specified later on. Generic code in Swift uses Pulp Fiction brackets, **`<`** and **`>`**, so it stands out pretty clearly!

To create a generic form of our **`inspectString()`** function, we would write this:

```
func inspect<SomeType>(_ value: SomeType) {
```

Note the use of **SomeType**: there's one in angle brackets directly after the function name, and one to describe the **value** parameter. The first one in the angle brackets is the most important, because it defines your placeholder data type: **inspect<SomeType>()** means "there's a function called **inspect()** that can be used with any sort of data type, but regardless of what data type is used I want to refer to it as **SomeType**." So, the parameter **value: SomeType** should now make more sense: **SomeType** will be replaced with whatever data type is used to call the function.

As you'll see in a few minutes, placeholder data types are also used for return values. But first here's the final version of the **inspect()** function that prints out correct information no matter what data is thrown at it:

```
func inspect<T>(_ value: T) {
    print("Received \(type(of: value)) with the value \(value)")
}

inspect("Haters gonna hate")
inspect(56)
```

I've used the **type(of:)** function so that Swift writes 'String', 'Int' and so on correctly. Notice that I've also used **T** rather than **SomeType**, which is a common coding convention: your first placeholder data type is named T, your second U, your third V and so on. In practice, I find this convention unhelpful and unclear, so although I'll be using it here it's only because you're going to have to get used to it.

Now, you might be wondering what benefit generics bring to this function – couldn't it just use **Any** for its parameter type, and do away with the placeholder data type altogether? In this case it could because the placeholder is used only once, so this is functionally identical:

```
func inspect(_ value: Any) {
    print("Received \(type(of: value)) with the value \(value)")
}
```

However, if we wanted our function to accept two parameters of the same type, the difference between **Any** and a placeholder becomes clearer. For example:

```
func inspect<T>(_ value1: T, _ value2: T) {
   print("1. Received \(type(of: value1)) with the value \
(value1)")
   print("2. Received \(type(of: value2)) with the value \
(value2)")
}
```

That now accepts two parameters of type **T**, which is our placeholder data type. Again, we don't know what that will be, which is why we give it an abstract name like "T" rather than a specific data type like **Int** or **String**. However, *both* parameters are of type **T**, meaning that they must *both* be the same type whatever that ends up being. So, this code is legal:

```
inspect(42, 42)
```

But this would not work, because it mixes data types:

```
inspect(42, "Dolphin")
```

If we had used **Any** for the data types, then Swift wouldn't ensure both parameters are the same type – one could be an **Int** and the other a **String**. So, this code would be perfectly fine:

```
func inspect(_ value1: Any, _ value2: Any) {
   print("1. Received \(type(of: value1)) with the value \
(value1)")
   print("2. Received \(type(of: value2)) with the value \
(value2)")
}

inspect(42, "Dolphin")
```

Limiting generics

You will often want to limit your generics so that they can operate only on similar types of data, and Swift makes this both simple and easy. This next function will square any two integers, regardless of whether they are **`Int`**, **`UInt`**, **`Int64`**, and so on:

```
func square<T: Integer>(_ value: T) -> T {
    return value * value
}
```

Notice that I've added a placeholder data type for the return value. In this instance it means that the function will return a value of the same data type it accepted.

Extending **`square()`** so that it supports other kinds of numbers – i.e. doubles and floats – is harder, because there is no protocol that covers all the built-in numeric types. So, let's create one:

```
protocol Numeric {
    static func *(lhs: Self, rhs: Self) -> Self
}
```

That doesn't contain any code, it just defines a protocol called **`Numeric`** and states that anything conforming to that protocol must be able to multiply itself. We want to apply that protocol to **`Float`**, **`Double`**, and **`Int`**, so add these three lines just below the protocol:

```
extension Float: Numeric {}
extension Double: Numeric {}
extension Int: Numeric {}
```

With that new protocol in place, you can square whatever you want:

```
func square<T: Numeric>(_ value: T) -> T {
    return value * value
}

square(42)
```

```
square(42.556)
```

Creating a generic data type

Now that you've mastered generic functions, let's take it up a notch with a fully generic data type: we're going to create a generic struct. When you're creating a generic data type you need to declare your placeholder data type as part of the struct's name, and that placeholder is then available to use in every property and method as needed.

The struct we're going to build is called "deque", which is a common abstract data type that means "double-ended queue." A regular queue is one where you add things to the end of the queue, and remove them from the front. A *deque* (pronounced "deck") is a queue where you can add things to the beginning or the end, and remove things from the beginning or end too. I chose to use a deque here because it's very simple to do by re-using Swift's built-in arrays – the concept is the key here, not the implementation!

To create a deque struct, we're going to give it a stored array property that will itself be generic, because it needs to hold whatever data type our deque stores. We'll also add four methods: **`pushBack()`** and **`pushFront()`** will accept a parameter of type **`T`** and add it to the right place, and **`popBack()`** and **`popFront()`** will return a **`T?`** (a placeholder optional data type) that will return a value from the back or front if one exists.

There's only one tiny complexity, which is that arrays don't have a **`popFirst()`** method that returns **`T?`**, so we need to add some extra code to run when the array is empty. Here's the code:

```
struct deque<T> {
    var array = [T]()

    mutating func pushBack(_ obj: T) {
        array.append(obj)
    }

    mutating func pushFront(_ obj: T) {
        array.insert(obj, at: 0)
    }
```

```
    mutating func popBack() -> T? {
        return array.popLast()
    }

    mutating func popFront() -> T? {
        if array.isEmpty {
            return nil
        } else {
            return array.removeFirst()
        }
    }
}
```

With that struct in place, we can start using it immediately:

```
var testDeque = deque<Int>()
testDeque.pushBack(5)
testDeque.pushFront(2)
testDeque.pushFront(1)
testDeque.popBack()
```

Working with Cocoa types

Cocoa data types – **`NSArray`**, **`NSDictionary`**, and so on – have been available to use since the earliest releases of Swift, but they can be difficult to work with because Objective-C's support for generics is both recent and limited.

In the case of **`NSCountedSet`**, one of my favorite Foundation types, generics aren't supported at all. This means you lose the automatic type safety bestowed by the Swift compiler, which in turn makes you one step closer to being a JavaScript programmer – and you don't want *that*, do you? Of course not.

Fortunately, I'm going to demonstrate to you how to create your own generic data type by creating a generic wrapper around **`NSCountedSet`**. This is just like a regular **`Set`** in that

each item is only stored once, but it has the added benefit that it keeps count of how many times you tried to add or remove objects. This means it can say "you added the number 5 twenty times" even though it's actually only in there once.

The basic code for this isn't too hard, although you do need to have imported **`Foundation`** in order to access **`NSCountedSet`**:

```
import Foundation

struct CustomCountedSet<T: Any> {
    let internalSet = NSCountedSet()

    mutating func add(_ obj: T) {
        internalSet.add(obj)
    }

    mutating func remove(_ obj: T) {
        internalSet.remove(obj)
    }

    func count(for obj: T) -> Int {
        return internalSet.count(for: obj)
    }
}
```

With that new data type in place, you can use it like so:

```
var countedSet = CustomCountedSet<String>()
countedSet.add("Hello")
countedSet.add("Hello")
countedSet.count(for: "Hello")

var countedSet2 = CustomCountedSet<Int>()
countedSet2.add(5)
countedSet2.count(for: 5)
```

All our struct does is wrap **`NSCountedSet`** with type safety, but that's always a welcome improvement. Given Apple's direction of travel in Swift 3, I wouldn't be surprised if they re-implement **`NSCountedSet`** as a generic struct-based **`CountedSet`** in the future – we'll see!

Chapter 3
References and Values

Natasha Murashev (@natashatherobot), author and speaker

Next time you get a crash, follow these instructions to get right to the problem: click on the **`objc_exception_throw`** in your thread, then type "po $arg1" into the debug area to get the human-readable version of the error. If you use exception breakpoints, you can even add the "po $arg1" command there so it's automatic.

What's the difference?

Understanding and exploiting the difference between reference types and value types is a critically important skill for any serious Swift developer. It's not "useful knowledge," or "good to know," but critically important – I'm not saying that lightly.

Over the next few chapters I'm going to go into detail about references and values so that you can learn for yourself, but first I want to start by explaining the difference between them. I've met too many developers who either don't understand or don't care – either of which is a mistake – so this is a sensible place to start.

I'm a big Star Wars fan, and you might remember that the end of the original Star Wars film (later titled "A New Hope") featured an attack on the Death Star. One of the Rebel pilots – Wedge Antilles – flew to attack the Death Star, but was damaged and had to fly back to base. Another Rebel pilot – Luke Skywalker – was also attacking the Death Star, but used the Force to save the day, at least until the next movie.

To illustrate the difference between references and values, I want to recreate this movie scene in simple Swift code. It's not complete, but it's enough you can follow along:

```
// create a target
var target = Target()

// set its location to be the Death Star
target.location = "Death Star"

// tell Luke to attack the Death Star
luke.target = target

// oh no – Wedge is hit! We need to
// tell him to fly home
target.location = "Rebel Base"
wedge.target = target

// report back to base
print(luke.target.location)
```

```
print(wedge.target.location)
```

Now, the question is this: when those pilots report back to base, what will they say? The answer is, "it depends." And what it depends on is – you guessed it – the difference between references and values.

You see, if **`Target`** was a class, which is a reference type in Swift, it would look like this:

```
class Target {
    var location = ""
}
```

And if **`Target`** was a struct, which is a value type in Swift, it would look like this:

```
struct Target {
    var location = ""
}
```

The code is almost identical, with just one word different. But the result is big: if **`Target`** was a struct then Luke would target the Death Star and Wedge would target the Rebel Base, but if it was a class then both Luke and Wedge would target the Rebel Base – which would make a very unhappy ending for the Rebels, and some deeply confused cinemagoers.

The reason for this behavior is that when you assign a reference type in more than one place, all those places point to the same piece of data. So even though Luke has a **`target`** property and Wedge has a **`target`** property, they are both pointing to the same instance of the **`Target`** class – changing one means the other changes too.

On the other hand, value types always have only one owner. When you assign a value type in more than one place, all those places point to individual copies of that value. When the code **`luke.target = target`** is run, Luke gets his own unique copy of the **`Target`** instance. So, when that gets changed to "Rebel Base", he doesn't care – he continues his attack on the Death Star.

Swift is an aggressively value-oriented language, meaning that most of its data types are values

rather than references. Booleans, integers, strings, tuples, and enums are all value types. Even arrays and dictionaries are value types, so the code below will print 3 and not 4:

```
var a = [1, 2, 3]
var b = a
a.append(4)
print(b.count)
```

Value types are simpler than reference types, which is no bad thing. When you're giving a value to work with, you can be sure its value won't change by surprise because you have your own unique copy. You can also compare value types easily, and it doesn't matter *how* they got their value – as long as two values look the same, they *are* the same. For example, the code below will print "Equal" even though **a** and **b** were created very differently:

```
let a = [1, 2, 3]
let b = Array(1...3)
if a == b { print("Equal") }
```

In short, references are shared on assignment and so can have multiple owners, whereas values are copied on assignment and therefore only ever have one owner.

Closures are references

This bit might hurt your brain, so if you rarely copy closures then feel free to skip over this chapter entirely.

As I already said, booleans, numbers, strings, arrays, dictionaries, structs, and more are all value types in Swift. Classes are reference types, but so are closures. For simple closures, this doesn't really matter. For example, the code below stores a trivial closure in **`printGreeting`**, calls it, assigns it to **`copyGreeting`**, then calls *that*:

```
let printGreeting = { print("Hello!") }
printGreeting()

let copyGreeting = printGreeting
copyGreeting()
```

In that code, the closure could be a value type and it wouldn't affect the output. Where things become complicated is when closures capture values, because any captured values are shared amongst any variables pointing at the same closure.

To give you a practical example, the code below is a **`createIncrementer()`** function that accepts no parameters and returns a closure:

```
func createIncrementer() -> () -> Void {
    var counter = 0

    return {
        counter += 1
        print(counter)
    }
}
```

Inside the **`createIncrementer()`** function is a single variable, **`counter`**, which is initialized to 0. Because that variable is used inside the closure that gets returned, it will be captured. So, we can follow that function with code like this:

```
let incrementer = createIncrementer()
incrementer()
incrementer()
```

The first line calls **`createIncrementer()`** and stores its returned closure in **`incrementer`**. The following two lines call **`incrementer()`** twice, triggering the closure, so you'll see 1 then 2 being printed out – the counter is moving upwards as expected.

Now for the important bit: because closures are references, if we create another reference to **`incrementer`**, they will share the same closure, and thus will also share the same **`counter`** variable. For example:

```
let incrementer = createIncrementer()
incrementer()
incrementer()

let incrementerCopy = incrementer
incrementerCopy()
incrementer()
```

When that code is run, you'll see 1, 2, 3, 4 being printed out, because both **`incrementer`** and **`incrementerCopy`** are pointing at the exact same closure, and therefore at the same captured value.

To repeat myself: if you use closures infrequently this is not likely to be a problem, and if you aren't using them to capture values then you're safe. Otherwise, step carefully: working with reference types can be hard enough without also introducing closure capturing.

Why use structs?

If you're faced with the choice between a class or a struct – i.e. a reference or a value – there are good reasons for choosing structs. This comparison applies to all reference and value types, but the most common time you need to make a choice is when it comes down to a class or a struct, so that's the context I'll be using here.

I already mentioned that value types are copied rather than shared, which means if your code has three different things pointing at the same struct they will each have their own copy – there's no chance of them treading on each others' toes. Once you understand this pass-by-copy behavior, you start to realize that it brings with it another set of benefits.

If you work with apps that are in any way complicated, one of the biggest benefit of value types is that they are inherently thread-safe. That is, if you're doing work on one or more background threads, it's not possible to cause race conditions for value types. To avoid confusion, let me briefly explain what this is before I explain the reason.

First, race conditions: this is the name given to a common class of bug caused by two pieces of code running in parallel where the order they finish affects the state of your program. For example, imagine I'm driving to your house to give you $1000, and a friend of yours is also driving to your house to demand payment for $200. I get there first, and give you the money. The friend gets there a few minutes later, and you give her $200 – all is well. Alternatively, your friend might drive faster than me and get there first, in which case they demand $200 and you haven't had any money from me – suddenly you have a fight on your hands.

In software, this might mean trying to act on a result before you've received it, or trying to work with an object that hasn't been created yet. Regardless, the result is bad: your software behaves inconsistently depending on which action happens first, which makes the problem hard to find and hard to solve.

Second, thread-safe: this is a term that means your code is written in such a way that multiple threads can use your data structures without affecting each other. If Thread A modifies something that Thread B was using, that is *not* thread-safe.

Value types are inherently thread-safe because they aren't shared between threads – each thread would get its own copy of your data to work with, and can manipulate that copy as

much as it wants without affecting other threads. Every copy is independent of other copies, so race conditions based on your data go away.

Taking away (or at least dramatically reducing) threading issues is great, but value types have one even bigger benefit: they greatly reduce the number of relationships that exist in your code. To give you an example, it's common to see Core Data apps that set up their database in the app delegate, then pass references to the database or individual objects between view controllers. Every time you pass an object between view controllers, you're connecting them bi-directionally – the child view controller could modify that object in any number of ways, and all those changes silently appear in the parent view controller.

As a result of this, the object relationships in your app can look less like a tree and more like an explosion in a spaghetti factory – relationships everywhere, changes that anyone could have made can appear and propagate throughout your app, and it all becomes very hard to reason about. That is, it becomes hard to say "how does this code work?" when your objects are entangled.

This becomes a non-issue when you use value types: as soon as you pass a struct from one view controller to another, the child has its own independent copy – there is no implicit relationship formed there, and no chance that the child can screw up its value and break the parent.

One last reason for preferring structs rather than classes: they come with memberwise initialization. This means the compiler automatically produces an initializer that provides default values for each of the struct's properties. For example:

```
struct Person {
    var name: String
    var age: Int
    var favoriteIceCream: String
}

let taylor = Person(name: "Taylor Swift", age: 26,
favoriteIceCream: "Chocolate")
```

It's a small thing, I know, but it's immensely practical and makes structs that little bit easier to use.

Why use classes?

There are some good reasons for using classes over structs, although at least one of them can come back to bite you. Reference types are a bit like names or handles, or even pointers if you're an old-school developer: when you pass them around, everyone points at the same value. This means you can create a resource such as a database connection, and share it between lots of objects in your app without having to create new connections.

Cunningly, this sharing aspect of classes is the strong reason both for and against using value types rather than references. The case for is one of flexibility and performance: in the short term, it's easy to pass one shared object around and have everyone modify it as needed. It's hacky, and it's brittle, but it's certainly easy to code. In fact, I would suggest that this approach is the *default* approach for many developers, particularly those who come from an Objective-C background.

This approach is also likely to run very quickly, because you don't need to create copies of the object every time you pass it somewhere else. Instead, you create your object once and pass it wherever it's needed – as long as your code is written to be thread-safe, objects can even be using across threads without worrying too much.

One feature that objects give you that structs don't is inheritance: the ability to take an existing class and build upon it. For many years now – easily one or even two generations of developers – inheritance has been a fundamental concept in software development, and now is deeply ingrained in our industry. It allows you to take an existing class, large or small, and build upon it in any way you want. You might add small tweaks, or massive changes. Some languages (not Swift fortunately!) allow you to inherit from multiple classes – intellectually very interesting, but rarely of use and never required.

Finally, one major reason for using classes is that Cocoa Touch itself is written using classes: UIKit, SpriteKit, MapKit, Core Location, and more are all object-oriented, as are protocols such as **`NSCoding`** and **`NSCopying`**. If you are writing Swift code that needs to work with Cocoa or Cocoa Touch, such as saving a custom data type to **`NSUserDefaults`**, chances you are you will need classes at some point.

Choosing between structs and classes

Whether to choose between a struct or a class depends on whether you want reference or value type behavior. Some of the advantages of each can actually be disadvantages depending on your context, so you'll need to read the advice below and make up your own mind.

First, the benefit of thread safety and freedom from race conditions is a major reason to use value types. It's easy to imagine that we're very clever programmers and can handle multithreading in our sleep, but the truth is that it's just not how our brain works: computers can literally do two, four, eight or more complicated tasks *simultaneously*, which is extremely hard for mere mortals to understand never mind debug.

Joel Spolsky wrote an excellent essay on this topic called "The Duct Tape Programmer,"[1] where he discusses the principle that smart programmers adopt simple solutions – the equivalent duct tape and WD-40. In the essay, he cites multi-threading as one example of complicated code, and says that "one principle duct tape programmers understand well is that any kind of coding technique that's even slightly complicated is going to doom your project."

You should also not underestimate the value of simplifying the relationships in your app. As I mentioned already, every time you pass an object from one place to another, you're implicitly creating a relationship that can backfire at any point in the future. If you did this only once or twice it might not be so hard to keep track of, but how often have you seen objects passed dozens of times? If you've ever used Core Data, you'll know that it's pretty much the antithesis of simple relationship modeling.

So yes, objects let you share data between different parts of your app, but do you really need that? It's usually more efficient in terms of absolute performance, but it can create a hugely complicated mess of relationships inside your architecture. Some value types – specifically, the ones built into Swift itself – have an optimization called copy on write that makes passing them as performant as passing objects, because Swift won't copy the value unless you try to change it. Sadly, this optimization does not come with your own structs, so you either code it yourself or take a (small) performance hit.

Another major advantage of classes is the ability to create a new class by inheriting from another. This is a powerful feature with a proven track record, and has the added benefit that many millions of developers understand the technique and have used it extensively. But as

powerful as it is, inheritance has its own problems: if you design a smart, useful and clear architecture that goes from A to B to C and perhaps even to D, what happens if you change your mind later – trying to remove B or put E where B was, for example? The answer is that it gets very messy.

Although inheritance is still a valuable tool in anyone's toolkit – particularly when it comes to building on Cocoa and Cocoa Touch – a newer, simpler approach is gaining popularity quickly: protocol-oriented development. This is where you add individual pieces of functionality horizontally rather than vertically, which allows you to change your mind as often as you want without causing problems. Swift's powerful ability to extend types as well as extensions makes inheritance far less useful than before: we make powerful data types through composition of functionality, rather than hierarchical inheritance.

Again, a lot of your choice will depend on your context. However, based on the above I want to offer a few summarizing points to help guide you.

First, I would recommend you choose structs over classes where possible. If you can't live without inheritance, or desperately need shared ownership of data, then go with classes, but structs should always be your default starting point. I prefer to use structs in my own code, but in this book I occasionally use classes because I'm trying to cover all bases.

Second, if you must use a class, declare it **`final`**. This gives you a performance boost right now, but it also should be your default position: unless you have specifically thought, "yes, this class is safe to be subclassed by others" then it's a mistake to allow it to happen. Do not underestimate the complexity of robust subclassability!

Third, struct or class, declare your properties as constants where possible. Immutability – data that cannot be changed – is baked right into Swift thanks to the **`let`** keyword, and it's a good habit to stay with.

Fourth, if you find yourself starting with a class no matter what, is this just a hangover from other programming languages? Objective-C developers use classes for almost everything, so classes can be a hard habit to shake. If you're someone who writes Swift as if it were Objective-C, you're missing half the fun – and half the efficiency – of this new language, so I suggest you give value types a thorough go before going back to classes.

Mixing classes and structs

Once you understand value types and reference types, it's much easier to choose between a class and a struct – at least at first. But as your application grows, you might find your situation becomes less black and white: your app might work 95% of the time with value types, but maybe once or twice everything would have been a whole lot easier if you had a reference type instead.

All is not lost: bearing in mind everything I've said about the relative advantages and disadvantages of both, there is a way to share value types in several places if you're convinced it's the right solution. The technique is called boxing – not the punchy, sweaty sort of boxing, but as in "placing something inside a box." This approach wraps a value type inside a reference type so you can share it more easily, and is commonly seen in languages such as C# and Java.

Note: I'm not going to continue repeating the warning that sharing rather than copying values raises the complexity of your program; please just take it as read!

I want to walk you through a practical example so you can see for yourself how it works. First, here's our **`Person`** struct again:

```
struct Person {
    var name: String
    var age: Int
    var favoriteIceCream: String
}

let taylor = Person(name: "Taylor Swift", age: 26,
favoriteIceCream: "Chocolate")
```

If we want to share that **`taylor`** struct across multiple objects, we'd create a **`PersonBox`** class like this:

```
final class PersonBox {
    var person: Person

    init(person: Person) {
```

```
        self.person = person
    }
}

let box = PersonBox(person: taylor)
```

That is a class container around the **`Person`** struct, and as a reference type will be shared rather than copied.

Finally, let's create a **`TestContainer`** class that simulates some parts of your app, such as different view controllers:

```
final class TestContainer {
    var box: PersonBox!
}

let container1 = TestContainer()
let container2 = TestContainer()

container1.box = box
container2.box = box
```

That creates two containers, each of which point at the same **`PersonBox`** object, which in turn means they are pointing at the same **`Person`** struct. To prove this works, we could write code like the below:

```
print(container1.box.person.name)
print(container2.box.person.name)

box.person.name = "Not Taylor"

print(container1.box.person.name)
print(container2.box.person.name)
```

That will print "Taylor Swift" twice, then "Not Taylor" twice, proving that changing the value

in one container changes the value in the other.

If you intend to make extensive use of boxing and unboxing (warning warning blah blah you could write this yourself, I hope!), you might want to consider creating a generic **Box** class like this:

```
final class Box<T> {
    var value: T

    init(value: T) {
        self.value = value
    }
}

final class TestContainer {
    var box: Box<Person>!
}
```

That way you can share other types of structs without having to create lots of different box classes.

It's undeniable that this approach weakens the power and safety of value types a little, but at least it makes explicit which situations give you the safety of value types and which don't – you're stating "this bit is explicitly shared" rather than implicitly sharing everything.

There's one more thing you should know, and it's something that's fairly obvious if you're coming from an Objective-C background: boxing and unboxing can be helpful if you're facing significant referencing counting performance issues. Swift, like modern Objective-C, uses a system called Automatic Reference Counting (ARC), which keeps track of how many times an object is referenced. When that count reaches 0, the object is automatically destroyed.

Swift's structs are *not* reference counted, because they are always uniquely referenced. But if a struct contains an object as one of its properties, that object *is* reference counted. For something small this isn't a problem, but if your struct has lots of objects as properties – let's say 10 – then suddenly ARC has to do 10 reference increments and decrements each time your

struct is copied. In this situation, boxing your struct in a wrapper object simplifies things dramatically, because ARC would only need to manipulate the reference count for the box, not for all the individual properties.

Immutability

Value and reference types differ in the way they handle immutability, but I've kept it separate here because it's quite a subtle difference and it's easy to get confused.

Let me back track a little: one thing I love about Swift is that it's aggressively focused on immutability. That is, it's very easy to say "don't let this value ever change." Not only does this mean that you're encouraged to use **`let`** rather than **`var`** when writing your code, but the Swift compiler will scan your code and warn when it finds places where variables could be made into constants. This is very different from Objective-C, where mutability was enforced only as part of your class name – **`NSString`** had an **`NSMutableString`** counterpart, **`NSArray`** had an **`NSMutableArray`** counterpart, and so on.

Immutability and value types might seem to go hand in hand. After all, if a value is just a value, how can it change? Integers are value types in Swift, and you can't exactly say "hey, I'm changing the number 5 so that 5 is actually equal to 6 now." But when you start comparing classes and structs, immutability is more complicated, and I want to explain why.

Consider the following code:

```
struct PersonStruct {
    var name: String
    var age: Int
}

var taylor = PersonStruct(name: "Taylor Swift", age: 26)
taylor.name = "Justin Bieber"
```

When that code runs, the final name value of the **`taylor`** instance is "Justin Bieber," which is going to be a huge surprise for any Swifties attending a concert!

If we change just one line of code, the result is very different:

```
let taylor = PersonStruct(name: "Taylor Swift", age: 26)
taylor.name = "Justin Bieber"
```

With just one line different, the code won't even compile because changing the **`name`** property is disallowed. Even though the **`name`** and **`age`** properties are marked as variable, the **`taylor`** struct is marked as a constant so Swift will not allow any part of it to change.

Here's where things get a bit complicated, but please knuckle down and bear with me: this is really important. Consider the following code:

```
final class PersonClass {
    var name: String
    var age: Int

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

let taylor = PersonClass(name: "Taylor Swift", age: 26)
taylor.name = "Justin Bieber"
```

That example uses a **`PersonClass`** class rather than a struct, but it leaves the **`taylor`** instance as being a constant. Now the code compiles, which is very different from when it was a struct.

When **`taylor`** is a constant struct, you can't change its value or its properties. When it's a constant object, you can't change its value but you *can* change its properties, as long as they aren't marked individually as constant. This means using structs allows you to enforce immutability far more strongly than using classes, which can help simplify your code even further.

The code below shows off all possible options – I've commented out the code that won't work:

```
// variable struct: changing property and changing value OK
var taylor1 = PersonStruct(name: "Taylor Swift", age: 26)
taylor1.name = "Justin Bieber"
```

```
taylor1 = PersonStruct(name: "Justin Bieber", age: 22)

// constant struct: changing property or value not allowed
let taylor2 = PersonStruct(name: "Taylor Swift", age: 26)
//taylor2.name = "Justin Bieber"
//taylor2 = PersonStruct(name: "Justin Bieber", age: 22)

// variable object: changing property and reference OK
var taylor3 = PersonClass(name: "Taylor Swift", age: 26)
taylor3.name = "Justin Bieber"
taylor3 = PersonClass(name: "Justin Bieber", age: 22)

// constant object: changing property OK, changing reference
not allowed
let taylor4 = PersonClass(name: "Taylor Swift", age: 26)
taylor4.name = "Justin Bieber"
//taylor4 = PersonClass(name: "Justin Bieber", age: 22)
```

As you can see, the difference occurs where constants are used: a constant object cannot be changed to point to a new object but you can change any of its properties, whereas a constant struct is totally fixed.

The most common place where mutability is a concern is when dealing with collections, such as arrays. These are value types in Swift (hurray!) but they can also be mutable – you can add and remove elements freely if you declare them as **`var`**. If you are able to, declare them as immutable so that Swift can optimize their use and provide extra safety.

Chapter 4
Functions

Simon Gladman (@flexmonkey), author of Core Image for Swift

When writing code to interpolate between two numbers, it's so easy to default to a linear interpolation. However, it's often a lot nicer to smoothly transition between two values. So my tip is avoid the pedestrian and interpolate with a function such as **`smootherStep()`**:

```
func smootherStep(value: CGFloat) -> CGFloat {
    let x = value < 0 ? 0 : value > 1 ? 1 : value

    return ((x) * (x) * (x) * ((x) * ((x) * 6 - 15) + 10))
}
```

Variadic functions

Variadic functions are functions of indefinite arity, which is a fancy way of saying that they take as many parameters as you send. This is used in some basic functions – even **`print()`** – to make your code easier and safer to write.

Let's work with **`print()`**, because it's a function you know well. You're used to seeing code like this:

```
print("I'm Commander Shepard and this is my favorite book")
```

But **`print()`** is a variadic function, which means you can pass it any number of things to print:

```
print(1, 2, 3, 4, 5, 6)
```

That will produce different output to calling **`print()`** once for each number: using one call prints the number on a single line, whereas using multiple calls would print the numbers one per line.

The variadic nature of **`print()`** becomes more useful once you add in the optional extra parameters: **`separator`** and **`terminator`**. The first one places a string between each of the values you passed, and the second places a string once all values have been printed. For example, this will print "1, 2, 3, 4, 5, 6!":

```
print(1, 2, 3, 4, 5, 6, separator: ", ", terminator: "!")
```

So, that's how to *call* variadic functions. Let's now talk about making them, which I think you'll find is quite slick in Swift.

Consider the code below:

```
func add(numbers: [Int]) -> Int {
    var total = 0

    for number in numbers {
        total += number
```

```
    }

    return total
}

add(numbers: [1, 2, 3, 4, 5])
```

That function accepts an array of integers, then adds each number together to form a total. There are more efficient ways of doing this, but that's not the point of this chapter!

To make that function variadic – i.e., so that it accepts any number of individual integers rather than a single array, two changes are needed. First, rather than writing **`[Int]`** we need to write **`Int...`**. Second, rather than writing **`add(numbers: [1, 2, 3, 4, 5])`** we need to write **`add(numbers: 1, 2, 3, 4, 5)`**

And that's it. So, the final code is this:

```
func add(numbers: Int...) -> Int {
    var total = 0

    for number in numbers {
        total += number
    }

    return total
}

add(numbers: 1, 2, 3, 4, 5)
```

You can place your variadic parameter anywhere in your function's parameter list, but you're only ever allowed one per function.

Operator overloading

This is a topic that people seem to love and hate in equal measure. Operator overloading is the ability to implement your own operators or even adjust the existing ones, such as **+** or *****.

The main reason for using operator overloading is that it gives you very clear, natural, and expressive code. You already understand that 5 + 5 equals 10 because you understand basic maths, so it's a logical extension to allow **myShoppingList + yourShoppingList**, i.e. the addition of two custom structs.

There are several downsides to operator overloading. First, its meaning can be opaque: if I say **HenryTheEighth + AnneBoleyn**, is the result a happily (for a time!) married couple, a baby in the form of the future Queen Elizabeth, or some conjoined human with four arms and legs?

Second, it does nothing that methods can't do: **HenryTheEighth.marry(AnneBoleyn)** would have the same result, and is significantly clearer. Third, it hides complexity: 5 + 5 is a trivial operation, but perhaps **Person + Person** involves arranging a ceremony, finding a wedding dress, and so on.

Fourth, and perhaps most seriously, operator overloading can produce unexpected results, particularly because you can adjust the existing operators with impunity.

The basics of operators

To demonstrate how confusing operator overloading can be, I want to start by giving you a basic example of overloading the **==** operator. Consider the following code:

```
if 4 == 4 {
    print("Match!")
} else {
    print("No match!")
}
```

As you might, that will print "Match!" because 4 is always equal to 4. Or is it…?

Enter operator overloading. With just three lines of code, we can cause critical damage to

pretty much every app:

```
func ==(lhs: Int, rhs: Int) -> Bool {
    return false
}

if 4 == 4 {
    print("Match!")
} else {
    print("No match!")
}
```

When that code runs, it will print "No match!" because we've written the **==** operator so that it always returns false. As you can see, the name of the function is the operator itself, i.e. **func ==**, so it's very clear what you're modifying. You can also see that this function expects to receive two integers (**lhs** and **rhs**, for left-hand side and right-hand side), and will return a boolean that reports whether those two numbers are considered equal.

As well as a function that does the actual work, operators also have precedence and associativity, both of which affect the result of the operation. When multiple operators are used together without brackets, Swift uses the operator with the highest precedence first – you probably learned this as PEMDAS (Parentheses, Exponents, Multiply, Divide, Add, Subtract), BODMAS, or similar depending on where you were schooled. If precedence alone is not enough to decide the order of operations, associativity is used.

Swift lets you control both precedence and associativity. Let's try an experiment now: what value should the following operation result in?

```
let i = 5 * 10 + 1
```

Following PEMDAS, the multiplication should be executed first (5 * 10 = 50), then the addition (50 + 1 = 51), so the result is 51. This precedence is baked right into Swift – here's the exact code from the Swift standard library:

```
precedencegroup AdditionPrecedence {
```

```
    associativity: left
    higherThan: RangeFormationPrecedence
}

precedencegroup MultiplicationPrecedence {
    associativity: left
    higherThan: AdditionPrecedence
}

infix operator * : MultiplicationPrecedence
infix operator + : AdditionPrecedence
infix operator - : AdditionPrecedence
```

That declares two operator precedence groups – things that are evaluated in the same order - then declares the `*`, `+`, and `-` operators to be in those groups. You can see that `MultiplicationPrecedence` is marked as being higher than `AdditionPrecedence`, which is what makes `*` be evaluated before `+`.

The three operators are called "infix" because they are placed inside two operands, i.e. 5 + 5 rather than a prefix operator like `!` that goes before something, e.g. `!loggedIn`.

Swift lets us redefine the precedence for existing operators by assigning them to new groups. You can create your own precedence groups if you want, or just re-use an existing one. In the code above, you can see that the order is multiplication precedence (used for `*`, `/`, `%` and more) followed by addition precedence (used for `+`, `-`, `|`, and more), followed by range formation precedence (used for `...` and `..<`.)

In the case of our little arithmetic we could cause all sorts of curious behavior by rewriting the `*` operator like this:

```
infix operator * : RangeFormationPrecedence
```

That redefines `*` to have a lower precedence than `+`, which means this code will now return 55:

```
let i = 5 * 10 + 1
```

That's the same line of code as before, but now it will be executed with the addition first (10 + 1 = 11) followed by the multiplication (5 * 11) to give 55.

When two operators have the same precedence, associativity comes into play. For example, consider the following:

```
let i = 10 - 5 - 1
```

Look again how Swift's own code declares the **`AdditionPrecedence`** group, to which the **`-`** operator belongs:

```
precedencegroup AdditionPrecedence {
    associativity: left
    higherThan: RangeFormationPrecedence
}
```

As you can see, it's defined as having left associativity, which means **`10 - 5 - 1`** is executed as **`(10 - 5) - 1`** and *not* **`10 - (5 - 1)`**. The difference is subtle, but important: unless we change it, **`10 - 5 - 1`** will give the result 4. Of course, if you want to cause a little havoc, you can change it like this:

```
precedencegroup AdditionPrecedence {
    associativity: right
    higherThan: RangeFormationPrecedence
}

infix operator - : AdditionPrecedence

let i = 10 - 5 - 1
```

That modifies the existing **`AdditionPrecedence`** group, then updates the **`-`** with the change – now the sum will be interpreted as **`10 - (5 - 1)`**, which is 6.

Adding to an existing operator

Now that you understand how operators work, let's modify the `*` operator so that it can multiply arrays of integers like this:

```
let result = [1, 2, 3] * [1, 2, 3]
```

Once we're done, that will return a new array containing [1, 4, 9], which is 1x1, 2x2 and 3x3.

The `*` operator already exists, so we don't need to declare it. Instead, we just need to create a new `func *` that takes our new data types. This function will create a new array made of multiplying each item in the two arrays it is provided. Here's the code:

```
func *(lhs: [Int], rhs: [Int]) -> [Int] {
    guard lhs.count == rhs.count else { return lhs }

    var result = [Int]()

    for (index, int) in lhs.enumerated() {
        result.append(int * rhs[index])
    }

    return result
}
```

Note that I added a `guard` at the beginning to ensure both arrays contain the same number of items.

Because the `*` operator already exists, the important thing is the `lhs` and `rhs` parameters, both of which are integer arrays: these parameters are what ensure this new function is selected when two integer arrays are multiplied.

Adding a new operator

When you add a new operator, you need to give Swift enough information to use it. At the very least you need to specify the new operator's position (prefix, postfix, or infix), but if you don't

specify a precedence or associativity Swift will provide default values that make it a low-priority, non-associative operator.

Let's add a new operator, **`**`**, which returns one value raised to the power of another. That is, **`2 ** 4`** should equal 2 * 2 * 2 * 2, i.e. 16. We're going to use the **`pow()`** function for this, so you'll need to import Foundation:

```
import Foundation
```

Once that's done, we need to tell Swift that **`**`** will be an infix operator, because we'll have one operand on its left and another on its right:

```
infix operator **
```

That *doesn't* specify a precedence or associativity, so the defaults will be used.

Finally, the new **`**`** function itself. I've made it accept doubles for maximum flexibility, and Swift is smart enough to infer 2 and 4 as doubles when used with this operator:

```
func **(lhs: Double, rhs: Double) -> Double {
    return pow(lhs, rhs)
}
```

As you can see, the function itself is a cinch thanks to **`pow()`**. Try it yourself:

```
let result = 2 ** 4
```

So far, so good. However, expressions like this one won't work:

```
let result = 4 ** 3 ** 2
```

In fact, even something like this won't work either:

```
let result = 2 ** 3 + 2
```

This is because we're using default precedence and associativity. To fix that, we need to

decide where `**` should rank compared to other operators, and for that you can either go back to PEMDAS (it's the E!) or just look at what other languages do. Haskell, for example, places it before multiplication and division, following PEMDAS. Haskell also declares exponentiation to be right associativity, meaning that `4 ** 3 ** 2` would be parsed as `4 ** (3 ** 2)`.

We can make our own `**` operator behave the same way by modifying its declaration to this:

```
precedencegroup ExponentiationPrecedence {
    higherThan: MultiplicationPrecedence
    associativity: right
}

infix operator **: ExponentiationPrecedence
```

With that change, you can now use `**` twice in the same expression, and also combine it with other operators – much better!

Modifying an existing operator

Now for something more complicated: modifying an existing operator. I've chosen a slightly more complicated example because if you can watch me solve it here I hope it will help you solve your own operator overloading problems.

The operator I'm going to modify is `...`, which exists already as the closed range operator. So, you can write `1...10` and get back a range covering 1 up to 10 inclusive. By default this is an infix operator with the low end of the range on the left and the high end of the range on the right, but I'm going to modify it so that it also accepts a range on the left and another integer on the right, like this:

```
let range = 1...10...1
```

When that code is run, it will return an array containing the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 – it counts up then down. This is possible because the operator appears twice: the first time it will see `1...10`, which is the closed range operator, but the second

time it will see `CountableClosedRange<Int>...1`, which is going to be our new operation. In this function, the `CountableClosedRange<Int>` is the left-hand side operand and the `Int` 1 is the right-hand side operand.

The new `...` function needs to do two things:

1. Calculate a new range by going from the right-hand integer up to the highest point in the left-hand range, then reversing this range.
2. Append the left-hand range to the newly created downward range, and return that as the result of the function.

In code, it looks like this:

```
func ...(lhs: CountableClosedRange<Int>, rhs: Int) -> [Int] {
    let downwards = (rhs ..< lhs.upperBound).reversed()
    return Array(lhs) + downwards
}
```

If you try that code out, you'll see it *won't* work – at least not yet. To see why, take a look at Swift's definition of the `...` operator:

```
infix operator ... : RangeFormationPrecedence

precedencegroup RangeFormationPrecedence {
    higherThan: CastingPrecedence
}
```

And now take a look at our code again:

```
let range = 1...10...1
```

You can see we're using the `...` operator twice, which means Swift needs to know whether we mean `(1...10)...1` or `1...(10...1)`. As you can see above, Swift's definition of `...` doesn't mention its associativity, so Swift doesn't know what to do in this situation. So, as things stand our new operator will work only with code like this:

```
let range = (1...10)...1
```

If we want that same behavior without the user having to add the parentheses, we need to tell Swift that `...` has left associativity, like this:

```
precedencegroup RangeFormationPrecedence {
    associativity: left
    higherThan: CastingPrecedence
}

infix operator  ... : RangeFormationPrecedence
```

That's it: the code now works correctly without parentheses, and we have a useful new operator. Don't forget that in playgrounds the order of your code matters a lot – your final code should be this:

```
precedencegroup RangeFormationPrecedence {
    associativity: left
    higherThan: CastingPrecedence
}

infix operator  ... : RangeFormationPrecedence

func ...(lhs: CountableClosedRange<Int>, rhs: Int) -> [Int] {
    let downwards = (rhs ..< lhs.upperBound).reversed()
    return Array(lhs) + downwards
}

let range = 1...10...1
print(range)
```

Closures

Like tuples, closures are endemic in Swift: global functions are closures, nested functions are closures, functional methods like **`sort()`** and **`map()`** accept closures, lazy properties use closures, and that's just the tip of the iceberg. You *will* need to use closures during your Swift career, and if you want to advance to a senior development position then you'll need to be comfortable *creating* them too.

I know some people have unusual ideas of what closures are, so let's start with a simple definition: a closure is a block of code that can be passed around and stored like a variable, which also has the ability to capture any values it uses. This capturing is really what makes closures hard to understand, so we'll come back to it later.

Creating simple closures

Let's create a trivial closure to get things moving:

```
let greetPerson = {
    print("Hello there!")
}
```

That creates a closure called **`greetPerson`**, which can then be used like a function:

```
greetPerson()
```

Because closures are first-class data types – i.e., just like integers, strings, and others – you can copy them and use them as parameters to other functions. Here's copying in action:

```
let greetCopy = greetPerson
greetCopy()
```

When copying closures, remembering that closures are a reference type – both those "copies" in fact point to the same shared closure.

To pass a closure into a function as a parameter, specify the closure's own parameter list and return value as its data type. That is, rather than writing **`param: String`**, you would write

something like **`param: () -> Void`** to accept a closure that has no parameters and returns no value. Yes, the **`-> Void`** is required, otherwise **`param: ()`** would mean an empty tuple.

If we wanted to pass our **`greetPerson`** closure into a function and call it there, we'd use code like this:

```
func runSomeClosure(_ closure: () -> Void) {
    closure()
}

runSomeClosure(greetPerson)
```

Of course, there isn't much point to that code: why is a closure even needed? Well, in that example it isn't, but what if we wanted to call the closure after 5 seconds? Or if we wanted to call it only sometimes? Or if certain conditions are met? This is where closures become useful: they are pieces of functionality that your app can store away to be used later on if needed.

Where closures start to get confusing is when they accept their own parameters, partly because their parameter list is placed in an unusual location, but also because the type syntax for these closures can look quite messy!

First: how to make a closure accept parameters. To do this, write your parameter list *inside* the closure's braces, followed by the keyword **`in`**:

```
let greetPerson = { (name: String) in
    print("Hello, \(name)!")
}

greetPerson("Taylor")
```

This is also where you specify a capture list if you need one. This is most commonly used to avoid strong reference cycles with **`self`** by making it **`unowned`** like this:

```
let greetPerson = { (name: String) [unowned self] in
    print("Hello, \(name)!")
}
```

```
greetPerson("Taylor")
```

Now, how to pass parameter using closures to functions. This is complicated for two reasons: 1) it can start to look like a sea of colons and parentheses, and 2) the calling convention varies depending on what you're trying to do.

Let's go back to our **`runSomeClosure()`** function. To make it accept one parameter – a closure that itself accepts one parameter - we need to define it like this:

```
func runSomeClosure(_ closure: (String) -> Void)
```

So, **`closure`** is a function that accepts a string and returns nothing. Here is that new function in action:

```
let greetPerson = { (name: String) in
    print("Hello, \(name)!")
}

func runSomeClosure(_ closure: (String) -> Void) {
    closure("Taylor")
}

runSomeClosure(greetPerson)
```

Closure capturing

I've already discussed how closures are reference types, which has cunning implications for captured values: when two variables point at the same closure, they both use the same captured data.

Let’s start with the basics: when a closure references a value, it needs to ensure that value will still be around when the closure is run. This can seem like the closure is copying the data, but it's actually more subtle than that. The process is called capturing, and it allows a closure to refer to and modify values that it refers to, even when the original values no longer exist.

The distinction is important: if the closure *copied* its values, value type semantics would apply and any changes to a value type inside a closure would be on a unique copy that would not affect the original caller. Instead, closures *capture* data.

I realize this all sounds hypothetical, so let me give you a practical example:

```
func testCapture() -> () -> Void {
    var counter = 0

    return {
        counter += 1
        print("Counter is now \(counter)")
    }
}

let greetPerson = testCapture()
greetPerson()
greetPerson()
greetPerson()

let greetCopy = greetPerson
greetCopy()
greetPerson()
greetCopy()
```

That declares a function called **`testCapture()`**, which has the return value **`() -> Void`** – i.e., it returns a function that accepts no parameters and returns nothing. Inside **`testCapture()`** I created a new variable called **`counter`**, giving it the initial value of 0. However, nothing else happens to that variable inside the function. Instead, it returns a closure that adds one to **`counter`** and prints out its new value. It doesn't *call* that closure, it just returns it.

Where things get interesting is after the function: **`greetPerson`** is set to be the function returned by **`testCapture()`**, and it gets called three times. That closure references the

counter value that was created inside **testCapture()**, which is clearly out of scope now because that function finished. So, Swift *captured* the value: that closure now has its own independent reference to **counter** that can be used when ever it's called. Each time the **greetPerson()** function is called, you'll see **counter** increment by one.

Where things get *doubly* interesting is with **greetCopy**. This is what I was saying about closures being references and using the same captured data. When **greetCopy()** is called, it increments the *same* **counter** value as **greetPerson** because they are both pointing at the same captured data. This means the **counter** value will move from 1 to 6 as the closure is called again and again. I've talked about this quirk twice now, so if it hurts your brain don't worry: it's not covered again!

Closure shorthand syntax

Before we get onto the more advanced stuff, I want to quickly go over the closure shorthand syntax in full so that we're definitely on the same wavelength. When you pass an inline closure to a function, Swift has several techniques in place so you don't have to write as much code.

To give you a good example, I'm going to use the **filter()** method of arrays, which accepts a closure with one string parameter, and returns true if that string should be in a new array. The code below filters an array so that we end up with a new array containing everyone whose name starts with Michael:

```
let names = ["Michael Jackson", "Taylor Swift", "Michael
Caine", "Adele Adkins", "Michael Jordan"]

let result1 = names.filter({ (name: String) -> Bool in
    if name.hasPrefix("Michael") {
        return true
    } else {
        return false;
    }
})

print(result1.count)
```

From that you can see that **`filter()`** expects to receive a closure that accepts a string parameter named **`name`** and returns true or false. The closure then checks whether the name has the prefix "Michael" and returns a value.

Swift knows that the closure being passed to **`filter()`** must accept a string and return a boolean, so we can remove it and just use the name of a variable that will be used for each item to filter:

```
let result2 = names.filter({ name in
    if name.hasPrefix("Michael") {
        return true
    } else {
        return false;
    }
})
```

Next, we can return the result of **`hasPrefix()`** directly, like this:

```
let result3 = names.filter({ name in
    return name.hasPrefix("Michael")
})
```

Trailing closures allow us to remove one set of parentheses, which is always welcome:

```
let result4 = names.filter { name in
    return name.hasPrefix("Michael")
}
```

As our closure has only one expression – i.e., now that we've removed a lot of code it only does one thing – we don't even need the **`return`** keyword any more. Swift knows our closure must return a boolean, and because we have only one line of code Swift knows that must be the one that returns a value. The code now looks like this:

```
let result5 = names.filter { name in
```

```
    name.hasPrefix("Michael")
}
```

Many people stop here, and with good reason: the next step can be quite confusing at first. You see, when this closure is called, Swift automatically creates anonymous parameter names made up of a dollar sign then a number counting up from 0. So, **$0**, **$1**, **$2**, and so on. You aren't allowed to use names like that in your own code so these tend to stand out!

These shorthand parameter names map to the parameters that the closure accepts. In this case, that means **name** is available as **$0**. You can't mix explicit arguments and anonymous arguments: either you declare a list of the parameters coming in, or you use **$0** and friends. Both of these two do exactly the same thing:

```
let result6 = names.filter { name in
    name.hasPrefix("Michael")
}

let result7 = names.filter {
    $0.hasPrefix("Michael")
}
```

Notice how you have to remove the **name in** part when using the anonymous names? Yes, it means even less typing, but at the same time you give up a little bit of readability. I prefer to use shorthand names in my own code, but you should use them only if you want to.

If you do choose to go with shorthand names, it's common at this point to put the whole method call on a single line, like this:

```
let result8 = names.filter { $0.hasPrefix("Michael") }
```

When you compare that to the size of the original closure, you have to admit it's a big improvement!

Functions as closures

Swift really blurs the lines between functions, methods, operators, and closures, which is marvelous because it hides all the compiler complexity from you and leaves developers to do what we do best: make great apps. This blurry behavior can be hard to understand at first, and is even harder to use in day-to-day coding, but I want to show you two examples that I hope will demonstrate just how clever Swift is.

My first example is this: given an array of strings called **`words`**, how can you find out whether any of those words exist in a string named **`input`**? One possible solution is to break up **`input`** into its own array, then loop over both arrays looking for matches. But Swift gives us a better solution: if you import Foundation, strings get a method called **`contains()`** that accepts another string and returns a boolean. So, this code will return true:

```
let input = "My favorite album is Fearless"
input.contains("album")
```

Swift arrays also have two **`contains()`** methods: one where you specify an element directly (in our case a string), but another that accepts a closure using a **`where`** parameter. That closure needs to accept a string and return a bool, like this:

```
words.contains { (str) -> Bool in
    return true
}
```

The brilliant design of Swift's compiler lets us put these two things together: even though the string's **`contains()`** is a Foundation method that comes from **`NSString`**, we can pass it into the array's **`contains(where:)`** in place of a closure. So, the entire code becomes this:

```
import Foundation
let words = ["1989", "Fearless", "Red"]
let input = "My favorite album is Fearless"
words.contains(where: input.contains)
```

The last line is the key. **`contains(where:)`** will call its closure once for every element in the array until it finds one that returns true. Passing it **`input.contains`** means that Swift will call **`input.contains("1989")`** and get back false, then it will call

`input.contains("Fearless")` and get back true – then stop there. Because **`contains()`** has the exact same signature that **`contains(where:)`** expects (take a string and return a bool), this works like a charm.

My second example uses the **`reduce()`** method of arrays: you provide a starting value, then give it a function to apply to every item in an array. Each time the function is called you will be given two parameters: the previous value when the function was called (which will be the starting value initially), and the current value to work with.

To demonstrate this in action, here's an example of calling **`reduce()`** on an array of integers to calculate the sum of them all:

```
let numbers = [1, 3, 5, 7, 9]

numbers.reduce(0) { (int1, int2) -> Int in
    return int1 + int2
}
```

When that code runs, it will add the starting value and 1 to produce 1, then 1 and 3 (running total: 4), then 4 and 5 (9), then 9 and 7 (16), then 16 and 9, giving 25 in total.

This approach is perfectly fine, but Swift has a much simpler and more efficient solution:

```
let numbers = [1, 3, 5, 7, 9]
let result = numbers.reduce(0, +)
```

When you think about it, **`+`** is a function that accepts two integers and returns their sum, so we can remove our whole closure and replace it with a single operator.

Escaping closures

When you pass a closure into a function, Swift considers it non-escaping by default. This means that the closure must be used immediately inside the function, and cannot be stored away for later – the Swift compiler will refuse to build if you try to use the closure after the function returns, e.g. if you were to use GCD's **`asyncAfter()`** method to call it after a

delay.

This works great for many types of functions, such as **`sort()`**, where you know for sure the closure will be used in the method then never again. The **`sort()`** method accepts a non-escaping closure as its only parameter because **`sort()`** won't try to store a copy of that closure for use later – it uses the closure immediately, then is finished with it.

On the other hand, *escaping* closures are ones that *will* be called after the method has returned. These exist in lots of places that need the closure to be called asynchronously. For example, you might be given a closure that should only be called when the user has made a choice. You can store that closure away, prompt the user to decide, then call the closure once you have the user's choice ready.

The distinction between escaping and non-escaping closures might sound small, but it matters because closures are *reference types*. Once Swift knows the closure won't be used once the function is finished – that it's *non-escaping* – it won't need to worry about referencing counting, so it can save some work. As a result, *non-escaping* closures are faster, and are the default in Swift. That is, unless you specify otherwise, all closure parameters are considered to be non-escaping.

If you want to specify an *escaping* closure, you need to use the **`@escaping`** keyword. The best way to see this in action is to demonstrate a situation when it's required. Consider the code below:

```
var queuedClosures: [() -> Void] = []

func queueClosure(_ closure: () -> Void) {
    queuedClosures.append(closure)
}

queueClosure({ print("Running closure 1") })
queueClosure({ print("Running closure 2") })
queueClosure({ print("Running closure 3") })
```

That creates an array of closures to run and a function that accepts a closure to queue. The

function doesn't do anything other than append whatever closure it was given to the array of queued closures. Finally, it calls **`queueClosure()`** three times with three simple closures, each one printing a message.

To finish off that code, we just need to create a new method called **`executeQueuedClosures()`** that loops over the queue and executes each closure:

```
func executeQueuedClosures() {
    for closure in queuedClosures {
        closure()
    }
}

executeQueuedClosures()
```

Let's examine the **`queueClosure()`** method more closely:

```
func queueClosure(_ closure: () -> Void) {
    queuedClosures.append(closure)
}
```

It takes a single parameter, which is a closure with no parameters or return value. That closure is then appended to the **`queuedClosures`** array. What this means is that the closure we pass in can be used later, in this case when the **`executeQueuedClosures()`** function is called.

Because the closures can be called later, Swift considers them to be *escaping* closures, and so it will refuse to build this code. Remember, non-escaping closures are the default for performance reasons, so we need to explicitly add the **`@escaping`** keyword to make our intention clear:

```
func queueClosure(_ closure: @escaping () -> Void) {
    queuedClosures.append(closure)
}
```

So: if you write a function that calls a closure immediately then doesn't use it again, it will be

non-escaping by default and you can forget about it. But if you intend to store the closure for later use, **`@escaping`** is required.

@autoclosure

The **`@autoclosure`** attribute is similar to **`@escaping`** in that you apply it to a closure parameter for a function, but it's used much more rarely. Well, no, that's not strictly true: it's common to *call* functions that use **`@autoclosure`** but uncommon to *write* functions with it.

When you use this attribute, it automatically creates a closure from an expression you pass in. When you call a function that uses this attribute, the code you write *isn't* a closure, but it *becomes* a closure, which can be a bit confusing – even the official Swift reference guide warns that overusing autoclosures makes your code harder to understand.

To help you understand how it works, here's a trivial example:

```
func printTest(_ result: () -> Void) {
    print("Before")
    result()
    print("After")
}

printTest({ print("Hello") })
```

That code creates a **`printTest()`** method, which accepts a closure and calls it. As you can see, the **`print("Hello")`** is inside a closure that gets called between "Before" and "After", so the final output is "Before", "Hello", "After".

If we used **`@autoclosure`** instead, it would allow us to rewrite the code **`printTest()`** call so that it doesn't need braces, like this:

```
func printTest(_ result: @autoclosure () -> Void) {
    print("Before")
    result()
    print("After")
}
```

```
printTest(print("Hello"))
```

These two pieces of code produce identical results thanks to **`@autoclosure`**. In the second code example, the **`print("Hello")`** won't be executed immediately because it gets wrapped inside a closure for execution later.

This behavior might seem rather trivial: all this work just removes a pair of braces, and makes the code harder to understand. However, there is one specific place where you'll use them: **`assert()`**. This is a Swift function that checks whether a condition is true, and causes your app to halt if not.

That might sound awfully drastic: why would you *want* your app to crash? You wouldn't, obviously, but when you're *testing* your app, adding calls to **`assert()`** helps ensure your code is behaving as expected. What you really want is for your assertions to be active in debug mode but disabled in release mode, which is exactly how **`assert()`** works.

Take a look at the three examples below:

```
assert(1 == 1, "Maths failure!")
assert(1 == 2, "Maths failure!")
assert(myReallySlowMethod() == false, "The slow method returned
false!")
```

The first will return true, so nothing will happen. The second will return false, so the app will halt. The third is an example of **`assert()`**'s power: because it uses **`@autoclosure`** to wrap your code in a closure, the Swift compiler simply doesn't run the closure when it's in release mode. This means you get all the safety of assertions while debugging, with none of the performance cost in release mode.

You might be interested to know that autoclosures are also used to handle the **`&&`** and **`||`** operators. Here is the complete Swift source code for **`&&`**, as found in the official compiler:

```
public static func && (lhs: Bool, rhs: @autoclosure () throws -
> Bool) rethrows -> Bool {
```

```
    return lhs ? try rhs() : false
}
```

Yes, that has try/catch, throws and rethrows, operator overloading, the ternary operator, and **`@autoclosure`** all in one tiny function. Still, I hope you can see through all that cleverness to understand what the code actually does: if **`lhs`** is true it returns the result of **`rhs()`**, otherwise false. This is short-circuit evaluation in action: Swift doesn't need to run the **`rhs`** closure if the **`lhs`** code already returned false.

One last thing on **`@autoclosure`**: if you want to make an *escaping* autoclosure, you should combine the two attributes. For example, we could rewrite the **`queueClosure()`** function from earlier like this:

```
func queueClosure(_ closure: @autoclosure @escaping () -> Void)
{
    queuedClosures.append(closure)
}

queueClosure(print("Running closure 1"))
```

Reminder: use autoclosures with care. They can make your code harder to understand, so don't use them just because you want to avoid typing some braces.

The ~= operator

I know it sounds deeply strange to have a favorite operator, but I do, and it's **~=**. I love it because of its simplicity. I love it even though it isn't really even needed. I even love its shape – just look at **~=** and admire its beauty! – so I hope you'll forgive me for taking a couple of minutes to show it to you.

Enough of me drooling over two simple symbols: what does this actually do? I'm glad you asked! **~=** is the pattern match operator, and it lets you write code like this:

```
let range = 1...100
let i = 42

if range ~= i {
    print("Match!")
}
```

Like I said, the operator isn't needed because you can code this using the built-in **contains()** method of ranges. However, it does have a slight syntactic sugar advantage over **contains()** because it doesn't require an extra set of parentheses:

```
let test1 = (1...100).contains(42)
let test2 = 1...100 ~= 42
```

I think **~=** is a lovely example of using operator overloading to declutter everyday syntax.

Chapter 5
Errors

Chris Eidhof (@chriseidhof), author of Advanced Swift

You can extend collections to have safe subscripts that return **`nil`** when the value isn't present:

```
extension Array {
    subscript(safe idx: Int) -> Element? {
        return idx < endIndex ? self[idx] : nil
    }
}
```

Error fundamentals

Swift has a unique take on error handling that is extremely flexible as long as you understand the full extent of what's on offer. I'm going to start off relatively simply and work through the full range of error handling techniques. Apple's Swift reference guide says "the performance characteristics of a **throw** statement are comparable to those of a **return** statement," which means they are *fast* – we have no reason to avoid them.

Let's start off with a simple example. All errors you want to throw must be an enum that conforms to the **Error** protocol, which Swift can bridge to the **NSError** class from Objective-C. So, we define an error enum like this:

```
enum PasswordError: Error {
    case empty
    case short
}
```

This is a regular enum just like any other, so we can add an associated value like this:

```
enum PasswordError: Error {
    case empty
    case short
    case obvious(message: String)
}
```

To mark a function or method as having the potential to throw an error, add **throws** before its return type, like this:

```
func encrypt(_ str: String, with password: String) throws ->
String {
    // complicated encryption goes here
    let encrypted = password + str + password
    return String(encrypted.characters.reversed())
}
```

You then use a mix of **do**, **try**, and **catch** to run the risky code. At the absolute minimum,

calling our current code would look like this:

```
do {
    let encrypted = try encrypt("Secret!", with: "T4yl0r")
    print(encrypted)
} catch {
    print("Encryption failed")
}
```

That either prints the result of calling **`encrypt()`**, or an error message. Using **`catch`** by itself catches all possible errors, which is required in Swift. This is sometimes called Pokémon error handling because "you gotta catch 'em all." Note: this restriction does *not* apply to top-level code in a playground; if you're using a playground, you should put your **`do`** block inside a function for testing purposes:

```
func testCatch() {
    do {
        let encrypted = try encrypt("Secret!", with: "T4yl0r")
        print(encrypted)
    } catch {
        print("Encryption failed")
    }
}

testCatch()
```

Handling all errors in a single **`catch`** block might work for you sometimes, but more often you'll want to catch individual cases. To do this, list each case individually, ensuring the generic **`catch`** is last so that it will only be used when nothing else matches:

```
do {
    let encrypted = try encrypt("secret information!", with:
"T4ylorSw1ft")
    print(encrypted)
} catch PasswordError.empty {
```

```
    print("You must provide a password.")
} catch PasswordError.short {
    print("Your password is too short.")
} catch PasswordError.obvious {
    print("Your password is obvious")
} catch {
    print("Error")
}
```

To work with an associated value, you need bind it to a constant inside your **`catch`** block:

```
catch PasswordError.obvious(let message) {
    print("Your password is obvious: \(message)")
}
```

If you want to test that in action, amend the **`encrypt()`** method to this:

```
func encrypt(_ str: String, with password: String) throws ->
String {
    // complicated encryption goes here
    if password == "12345" {
        throw PasswordError.obvious(message: "I have the same
number on my luggage")
    }

    let encrypted = password + str + password
    return String(encrypted.characters.reversed())
}
```

One last thing before we move on: when working with associated values you can also use pattern matching. To do this, first use **`let`** to bind the associated value to a local constant, then use a **`where`** clause to filter. For example, we could modify our **`PasswordError.short`** case to return how many characters should have been provided:

```
case short(minChars: Int)
```

With that change, we can catch variations on **`short`** by filtering by the **`minChars`** associated value:

```
catch PasswordError.short(let minChars) where minChars < 5 {
   print("We have a lax security policy: passwords must be at
least \(minChars)")
} catch PasswordError.short(let minChars) where minChars < 8 {
   print("We have a moderate security policy: passwords must be
at least \(minChars)")
} catch PasswordError.short(let minChars) {
   print("We have a serious security policy: passwords must be
at least \(minChars)")
}
```

Error propagation

When you use **try** to call a function, Swift forces you to handle any errors. This is sometimes unhelpful behavior: if function A calls function B, and function B calls function C, who should handle errors thrown by C? If your answer is "B" then your existing error handling knowledge is enough for you.

If your answer is "A" – i.e., that one caller should handle some or all errors in any functions that it calls as well as any errors in functions that *those* functions call, and so on, you need to learn about *error propagation*.

Let's model the **A()**, **B()**, **C()** function calls along with a trimmed version of the **PasswordError** enum we already used:

```
enum PasswordError: Error {
    case empty
    case short
    case obvious
}

func functionA() {
    functionB()
}

func functionB() {
    functionC()
}

func functionC() {
    throw PasswordError.short
}
```

That code won't compile as-is because **functionC()** throws an error but isn't marked with **throws**. If we add that, the code is as follows:

```
func functionC() throws {
```

```
    throw PasswordError.short
}
```

But now the code still won't compile, because **functionB()** is calling a throwing function without using **try**. Now we are presented with several options, and I want to explore them individually.

The first option is to catch all errors inside **functionB()**. This is the option you'll use if you want **functionA()** to be oblivious to any errors that happen below it, and looks like this:

```
func functionA() {
    functionB()
}

func functionB() {
    do {
        try functionC()
    } catch {
        print("Error!")
    }
}

func functionC() throws {
    throw PasswordError.short
}
```

You could add individual **catch** blocks to **functionB()**, but the principle is still the same.

The second option is for **functionB()** to ignore the errors and let them bubble upwards to its own caller, which is called *error propagation*. To do this, we need to move the **do/catch** code out of **functionB()** and into **functionA()**. We then just need to mark **functionB()** with throws to have it work, like this:

```
func functionA() {
    do {
```

```
        try functionB()
    } catch {
        print("Error!")
    }
}

func functionB() throws {
    try functionC()
}

func functionC() throws {
    throw PasswordError.short
}
```

Before we look at the third option, I want you to take a close look at the current code for **`functionB()`**: it's a function that uses **`try`** *without* having a **`do/catch`** block around it. This is perfectly fine, as long the function itself is marked as **`throws`** so any errors can continue to propagate upwards.

The third option is to delegate parts of error handling to whichever function is most appropriate. For example, you might want **`functionB()`** to catch empty passwords, but **`functionA()`** to handle all other errors. Swift normally wants all **`try/catch`** blocks to be exhaustive, but if you're in a throwing function that requirement is waived – any errors you don't catch will just propagate upwards.

The code below has **`functionB()`** handle empty passwords, and **`functionA()`** handle everything else:

```
func functionA() {
    do {
        try functionB()
    } catch {
        print("Error!")
    }
}
```

```
func functionB() throws {
    do {
        try functionC()
    } catch PasswordError.empty {
        print("Empty password!")
    }
}

func functionC() throws {
    throw PasswordError.short
}
```

Ultimately all error cases must be caught, so at some point you need to have a generic catch all statement.

Throwing functions as parameters

I'm now going to walk through a really useful feature of Swift, and I'm saying this up front because if you find yourself questioning why it's useful I want to make sure you power through – trust me, it's worth it!

First, here's an important quote from the Swift reference guide: "nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in the same places as a throwing one."

Think about that for a moment. Let it sink in: nonthrowing functions are subtypes of throwing functions, so you can use them anywhere that expects a throwing function. You could even write code like the below if you wanted to, although you'll get a compiler warning because it's unnecessary:

```
func definitelyWontThrow() {
    print("Shiny!")
}

try definitelyWontThrow()
```

Where this starts to really matter is when you use a throwing function as a parameter, and I want to give you a working example so you can learn all this in a practical context.

Imagine an app that must fetch user data, either remotely or locally, then act upon it. There's a function to fetch remote user data, which might throw an error if there's a network problem. There's a second function to fetch local user data, which is guaranteed to work and so doesn't throw. Finally, there's a third function that calls one of those two fetch functions, then acts upon the result.

Putting that final function to one side, the initial code might look like this:

```
enum Failure: Error {
    case badNetwork(message: String)
    case broken
}
```

```
func fetchRemote() throws -> String {
    // complicated, failable work here
    throw Failure.badNetwork(message: "Firewall blocked port.")
}

func fetchLocal() -> String {
    // this won't throw
    return "Taylor"
}
```

The third function is where things get interesting: it needs to call either **`fetchRemote()`** or **`fetchLocal()`** and do something with the data that was fetched. Both those functions accept no parameters and return a string, however one is marked **`throws`** and the other is not.

Think back to what I wrote a few minutes ago: you can use nonthrowing functions anywhere that expects a throwing function. So, we can write a function like this: **`fetchUserData(using closure: () throws -> String)`**. Let's break that down:

- It's called **`fetchUserData()`**
- It accepts a closure as a parameter
- That closure must accept no parameters and return a string.

But the closure isn't *required* to throw: we've said that it *can*, but it doesn't *have* to.

With that in mind, a first pass at the **`fetchUserData()`** function might look like this:

```
func fetchUserData(using closure: () throws -> String) {
    do {
        let userData = try closure()
        print("User data received: \(userData)")
    } catch Failure.badNetwork(let message) {
        print(message)
    } catch {
```

```
        print("Fetch error")
    }
}

fetchUserData(using: fetchLocal)
```

As you can see, we can move from local fetching to remote fetching by changing only the last line:

```
fetchUserData(using: fetchRemote)
```

So, it doesn't matter if the closure we pass in does or does not throw, as long as we state that it *might* and handle it appropriately.

Where things get interesting – and by interesting I mean thoroughly awesome – is when you want to use error propagation with a throwing closure parameter. Stick with me – we're almost through!

A simple solution might declare **`fetchUserData()`** as throwing then catch errors at the caller, like this:

```
func fetchUserData(using closure: () throws -> String) throws {
    let userData = try closure()
    print("User data received: \(userData)")
}

do {
    try fetchUserData(using: fetchLocal)
} catch Failure.badNetwork(let message) {
    print(message)
} catch {
    print("Fetch error")
}
```

That works fine for this exact situation, but Swift has a much smarter solution. This

`fetchUserData()` could be called elsewhere in our app, perhaps more than once – it's going to get awfully messy having all that **`try/catch`** code in there, particularly the times we use **`fetchLocal()`** and know for a fact it won't throw.

Swift's solution is the **`rethrows`** keyword, which we can use to replace the regular **`throws`** in the **`fetchUser`** function like this:

```
func fetchUserData(using closure: () throws -> String) rethrows
{
    let userData = try closure()
    print("User data received: \(userData)")
}
```

So, the closure *throws*, but the **`fetchUserData()`** function *rethrows*. The difference might seem subtle, but this code will now create a warning in Xcode:

```
do {
    try fetchUserData(using: fetchLocal)
} catch Failure.badNetwork(let message) {
    print(message)
} catch {
    print("Fetch error")
}
```

If you replace **`try fetchUserData(using: fetchLocal)`** with **`try fetchUserData(using: fetchRemote)`** then the warning goes away. What's happening is that the Swift compiler is now checking each call to **`fetchUserData()`** individually, and now only requires **`try/catch`** to be used when the closure you pass in throws.

So, when you use **`fetchUserData(using: fetchLocal)`** the compiler can see **`try/catch`** isn't necessary, but when you use **`fetchUserData(using: fetchRemote)`** Swift will ensure you catch errors correctly.

So, with **`rethrows`** you get the best of both worlds: when you pass in a closure that will

throw you get all the safety you'd expect from Swift, but when you pass in a closure that won't throw you don't need to add pointless **try/catch** code.

Armed with this knowledge, take another look at the code for the short-circuit logical **&&** operator, taking from the Swift source code:

```
public static func && (lhs: Bool, rhs: @autoclosure () throws -
> Bool) rethrows -> Bool {
   return lhs ? try rhs() : false
}
```

You should now be able to break down exactly what that does: the right-hand side of the **&&** is an autoclosure so that it gets executed only if the left-hand side evaluates to true. **rhs** is marked as throwing even though it might not, and the whole function is marked as **rethrows** so that the caller is required to use **try/catch** only when necessary.

I hope you'll agree that Swift's error handling approach is fractally beautiful: the more you dig into it, the more you come to appreciate its subtle genius.

try vs try? vs try!

Swift's error handling comes in three forms, and all three have their uses. All are used when calling a function that is marked with **throws**, but have subtly different meaning:

1. When using **try** you must have a **catch** block to handle any errors that occur.
2. When using **try?** the function you call will automatically return **nil** if any errors are thrown. You don't need to catch them, but you do need to be aware that your return value becomes optional.
3. When using **try!** the function will crash your application if any errors are thrown.

I've numbered them because that's the order you should use them in: regular **try** is by far the most common and behaves as we've seen so far; **try?** is a safe and useful fallback that, when used wisely, will do a lot to improve your code's readability; **try!** means "throwing an error is so unlikely – or so undesirable – that I'm willing to accept a crash," and is uncommon.

Now, you might wonder why **try!** even exists: if you're certain it won't throw an error, why is the function marked with **throws** in the first place? Well, consider code that reads a file from your application bundle. Loading a file from the contents of a string can throw an error if the file doesn't exist or isn't readable, but if that situation arises your app is clearly in a very broken state – forcing a crash might well be a desirable outcome rather than allowing the user to continue with a corrupted app. The choice is yours.

Of the three, only the regular **try** requires a **do/catch** block, so if you're looking for concise code you are likely to want to use **try?** on occasion. I've already covered **try**, and **try!** is effectively identical to **try?** except rather than a nil return value you get a crash, so I'll focus on using **try?** here.

Using **try?** means safely unwrapping its optional return value, like this:

```
if let savedText = try? String(contentsOfFile: "saved.txt") {
    loadText(savedText)
} else {
    showFirstRunScreen()
}
```

You can also use the nil coalescing operator **??** to use a default value if **nil** is returned, thus removing optionality altogether:

```
let savedText = (try? String(contentsOfFile: "saved.txt")) ??
"Hello, world!"
```

On rare occasions, I use **try?** bit like UDP: try this thing, but I don't care if it fails. Two of the most important protocols behind the internet are called TCP and UDP. TCP guarantees that all packets will arrive, and will keep trying to re-send until a certain time has expired; it's used for downloads, for example, because if you're missing part of a zip file then you have nothing.

UDP sends all data packets once and hopes for the best: if they arrive, great; if not, just wait until the next one arrives. UDP is useful for things like video streaming, where you don't care if you lost a split second of a live stream – it's more important that new video keeps coming in.

So, **try?** can be used a bit like UDP: if you don't care what the return value is and you just want to make an attempt to do something, **try?** is for you:

```
_ = try? string.write(toFile: somePathHere, atomically: true,
encoding: String.Encoding.utf8)
```

Assertions

Assertions allow you to state that some condition must be true. The condition is down to you, and can be as complicated as you like, but if it evaluates to **`false`** your program will halt immediately. Assertions are cleverly designed in Swift, to the point where reading its source code is a valuable exercise.

When you write **`assert`** in Xcode, the code completion will give you two options:

```
assert(condition: Bool)
assert(condition: Bool, message: String)
```

In the first option you need to provide a condition to be tested; in the second, you also provide a message to display if the condition evaluates to false. In the code below, the first assertion will evaluate to true and so nothing will happen; in the second, it will fail because the condition is false, so a message will be printed:

```
assert(1 == 1)
assert(1 == 2, "Danger, Will Robinson: mathematics failure!")
```

Obviously asserting basic arithmetic isn't much use, so you'll usually write code like this:

```
let success = runImportantOperation()
assert(success == true, "Important operation failed!")
```

Assertions are extremely useful when writing complicated apps, because you can scatter them throughout your code to ensure that everything is as it should be. You can test as much as you want, as often as you want, and it means that any unexpected state in your app – when you say "how did **`someVar`** get set to *that*?" – gets caught early in development.

To explore how assertions work, take a look at the actual type signature from the Swift source code:

```
public func assert(_ condition: @autoclosure () -> Bool, _
message: @autoclosure () -> String = String(), file:
StaticString = #file, line: UInt = #line)
```

The last two parameters have defaults provided by the Swift compiler, and it's unlikely you'll want to change them: **`#file`** is replaced by the name of the current file, and **`#line`** is replaced by the line number of the code that triggered the assertion. These are passed in as parameters with default values (as opposed to being specified inside the function) so that Swift uses the filename and line number of the call site, not some line inside **`assert()`**.

The first two parameters are much more interesting: both the **`condition`** and **`message`** parameters use **`@autoclosure`**, which means they aren't immediately executed. This is important, because internally Swift will execute assertions only if it's in debug mode. This means you can assert hundreds or even thousands of times in your app, but all that work will only get done if you're debugging. When the Swift compiler runs in release mode, the work is skipped.

Here's the body of **`assert()`**, straight from the Swift source code:

```
if _isDebugAssertConfiguration() {
   if !_branchHint(condition(), expected: true) {
      _assertionFailed("assertion failed", message(), file,
line, flags: _fatalErrorFlags())
   }
}
```

The functions prefixed with an underscore are internal, but you ought to be able to guess what they do:

- **`_isDebugAssertConfiguration()`** returns false if we're not in debug mode. This is what causes the assertion to disappear when built for release.
- **`!_branchHint(condition(), expected: true)`** runs the condition closure that was created by **`@autoclosure`**. It tells the compiler to expect the condition to evaluate successfully (which ought to be the case most of the time), which helps optimize your code. This only affects debugging, but it helps your assertions run a little faster.
- If we're in debugging mode and calling **`condition()`** returned false, then **`_assertionFailed()`** is called to terminate your program. At this point, the

`message()` closure is called to print a useful error.

The use of **`@autoclosure`** is perfect for the condition, because this way it gets run only if we're in debug mode. But you might wonder why the **`message`** parameter is also an autoclosure – isn't it just a string? This is where **`assert()`** gets *really* clever: because **`message`** is a closure, you can run any other code you like before finally returning a string, and none of it will get called unless your assertion fails.

The most common use for this is when using a logging system: your **`message`** closure can write an error to your log, before returning that message back to **`assert()`**. Below is a simplified example that writes your message to a file before returning it – it's basically just a pass-through that adds some extra functionality:

```
func saveError(message: String, file: String = #file, line: Int
= #line) -> String {
    _ = try? message.write(toFile: pathToDebugFile, atomically:
true, encoding: String.Encoding.utf8)
    return message
}

assert(1 == 2, saveError(message: "Fatal error!"))
```

Preconditions

Assertions are checked only when your app is running in debug mode, which is useful when you're developing but automatically deactivated in release mode. If you wish to make assertions in release mode – bearing in mind that a failure causes your app to terminate immediately – you should use **`precondition()`** instead.

This takes identical parameters to **`assert()`** but is compiled differently: if you build with **`-Onone`** or **`-O`** (no optimization or standard optimization), a failed precondition will cause your app to terminate. If you build with **`-Ounchecked`** – the fastest optimization level – only then will preconditions be ignored. If you're using Xcode, this means the "Disable Safety Checks" build option is set to Yes.

Just like using **try!**, there is a good reason why you might want to crash your app in release mode: if something has gone fatally wrong in such a way that it suggests your app is in an unstable, unknown, or perhaps even dangerous state, it's better to bail out than continue and risk serious data loss.

In the chapter on operator overloading, I presented a modification to the * operator that allows us to multiply two arrays:

```
func *(lhs: [Int], rhs: [Int]) -> [Int] {
    guard lhs.count == rhs.count else { return lhs }

    var result = [Int]()

    for (index, int) in lhs.enumerated() {
        result.append(int * rhs[index])
    }

    return result
}
```

As you can see, the function kicks off with a **guard** to ensure both arrays are exactly the same size – if they aren't, we just return the left-hand side operand. In many cases this is safe programming, but it's also possible that if our program ended up with two arrays of different sizes then something serious has gone wrong and we should halt execution. In this situation, using **precondition()** might be preferable:

```
func *(lhs: [Int], rhs: [Int]) -> [Int] {
    precondition(lhs.count == rhs.count, "Arrays were not the
same size")

    var result = [Int]()

    for (index, int) in lhs.enumerated() {
        result.append(int * rhs[index])
    }
```

```
    return result
}

let a = [1, 2, 3]
let b = [4, 5]
let c = a * b
```

Remember, enabling **`-Ounchecked`** will effectively disable your preconditions, but it also disables other bounds checking too – that's why it's so fast!

Chapter 6
Functional programming

Veronica Ray (@nerdonica), software engineer at LinkedIn

If you need the index for each element along with its value, you can use the enumerated() method to iterate over the array:

```
for (index, element) in loggerContent.enumerated() {
    logfiles["logfile\(index).txt"] = element
}
```

What is functional programming?

Depending on where you're coming from, functional programming might be a perfectly normal way of writing code, it might be an academic tool that somehow snuck out of university campuses, or it might something you use to scare kids into staying in bed. The truth is that functional programming usage covers a spectrum, from "I'm basically reading algebra" to "I'm mostly using object-oriented code, with a few functional techniques where I want them."

I've read a lot of tutorials about functional programming that mean well, but probably end up doing more harm than good. You see, functional programming isn't too hard to pick up and use in a practical way, but you can also overwhelm people with talk of monads and functors. I aim to teach you the benefits of functional programming while also cutting out the parts you're less likely to benefit from. I make no apologies: this whole book is about teaching you hands-on techniques to improve your coding immediately, not explaining what referential transparency is.

I want to teach you some functional programming using what Andy Matuschak describes as "lightweight encounters." This means we'll focus on finding simple, small benefits that you can understand and use straight away. Andy is an experienced functional developer, but he's also pragmatic about learning approaches – I hope he won't frown too much on me simplifying (and even jettisoning) so much theory!

Before I jump into functional code, I want to give you a broad idea of why we might want to change the current way of working. Chances are that you have used object orientation extensively in the past: you've created classes, made subclasses from those classes, added methods and properties, and more. If you've already read the chapter on reference and value types, you'll also know that objects are reference types, which means a property could be changed by any of its multiple owners.

We consider object orientation easy because it's what we know. If I tell you that `Poodle` inherits from `Dog`, has a `barkVolume` property and a `biteStrength` property, as well as a `barkWorseThanBite()` method, you immediately understand what all that means. But it's not *simple*. It's complicated: that one "easy" class blends state, functionality, inheritance, and more, so you need to keep a lot in your head to follow along.

Functional programming – or at least the slightly simplified definition I'll be working with here

– can dramatically simplify your code. At first, this will cause problems because you're effectively fighting object-oriented muscle memory: your instinct to solve every problem by creating a new class needs to be put on hold, at least for the time being.

Instead, we're going to apply five principles that help deliver the benefits of functional programming without the algebra.

First, functions are first-class data types. That means they can be created, copied, and passed around like integers and strings. Second, because functions are first-class data types, they can be used as parameters to other functions. Third, in order to allow our functions to be re-used in various ways, they should always return the same output when given specific input, and not cause any side effects. Fourth, because functions always return the same output for some given input, we should prefer to use immutable data types rather than using functions to change mutable variables. Fifth and finally, because our functions don't cause side effects and variables are all immutable, we can reduce how much state we track in our program – and often eliminate it altogether.

I know that's a lot to take in all at once, so let me try to break down each of those into more detail.

You should already know that functions are first-class data types in Swift – after all, you can copy closures and pass them around. So, that's one down. Next up, passing functions as parameters to other functions is also something you maybe have done already, such as calling `sort()` with a closure. You might come across the name "higher-order function", which is the name given to a function that accepts another function as a parameter.

Where things get a bit more complicated – but a lot more interesting – is when we write functions that always produce the same output for a given input. This means if you write a function `lengthOf(strings:)` that accepts an array of strings and returns back an array of integers based on the length of each string, that function will return [6, 4, 5] when given ["Taylor", "Paul", "Adele"]. It doesn't matter what else has happened in your program, or how often the function has been called: the same input must return the same output.

A corollary is that functions ought not to create side effects that might affect other functions. For example, if we had another function `fetchSystemTime()` that returned the time, it

should not be the case that calling `fetchSystemTime()` could affect the result of `lengthOf(strings:)`. A function that always returns the same result for a given input without causing side effects is often called a *pure* function. I guess that makes all other functions *impure*, and you don't want to write dirty, dirty functions, do you?

One source of confusion about pure functions revolves around the meaning of "side effect." If a function does something like writing to disk, is that a side effect or in fact just the main point of the function? There's a lot of argument about this, and I'm not going to wade into it. Instead, I'll say that functional programmers should *aspire* to create pure functions, but when it comes to the crunch favor predictable output for known input over avoiding side effects. That is, if you want to write a function that writes some data to disk (a side effect? the actual effect? call it what you want!) then go for it, but at least make sure it writes exactly the same thing when given the same data.

I've already covered the importance of immutable data types and values rather than references, so I shan't go over them again other than to say that classes are about as welcome in functional code as a hedgehog in a hemophilia convention.

Finally, the lack of state can be tricky, because it's so deeply baked into object orientation. "State" is a series of values stored by your program, which isn't always a bad thing – it includes caching things to increase performance, and also important things like user settings. The problem comes when this state gets used inside a function, because it means the function is no longer predictable.

Using the `lengthOf(strings:)` example from earlier, consider what would happen if we had a boolean setting called `returnLengthsAsBinary` – the function that would always return [6, 4, 5] when given ["Taylor", "Paul, "Adele"] could now also return ['110', '10', '101'] depending on the value of something completely external. Being pragmatic, don't avoid state at all costs, but never let it pollute your functional code, and work to minimize its use more generally.

When all five of these principles combine, you get a number of immediate, valuable benefits. When you write functions that produce predictable output, you can write unit tests for them trivially. When you use immutable value types rather than reference types, you remove unexpected dependencies in your app and make your code easier to reason about. When you

build small, composable functions that can be combined with higher-order functions and re-used in any number of ways, you can build hugely powerful apps by putting together many small, simple pieces.

Note: I'll be referring to these five principles in subsequent chapters. Rather than repeat myself again and again, I'll just say the Five Functional Principles, and hopefully you'll remember 1) first-class data types, 2) higher-order functions, 3) pure functions, 4) immutability, and 5) reduce state.

OK, enough theory. I hope I've managed to convince you that functional programming has something to offer everyone, and even if you take only a few concepts from the following sections that's a big improvement.

map()

Let's start with the easiest way into functional programming: the **`map()`** method. This takes a value out of a container, applies a function to it, then puts the result of that function back into a new container that gets returned to you. Swift's arrays, dictionaries, and sets have **`map()`** built in, and it's commonly used to loop through every item in an array while applying a function to each value.

I already mentioned a **`lengthOf(strings:)`** function that accepts an array of strings and return an array of integers based on the size of the input strings. You might write it something like this:

```
func lengthOf(strings: [String]) -> [Int] {
    var result = [Int]()

    for string in strings {
        result.append(string.characters.count)
    }

    return result
}
```

That function takes an array of strings and returns an array of integers based on those strings. This is a perfect use for **`map()`**, and in fact we can replace all that code with this:

```
func lengthOf(strings: [String]) -> [Int] {
    return strings.map { $0.characters.count }
}
```

It's clear that the functional approach is shorter, but its benefit is not just about writing less code. Instead, the functional version conveys significantly more meaning to the compiler: it's now clear we want to apply some code to every item in an array, and it's down to Swift to make that happen efficiently. For all we know, Swift could parallelize your closure so that it gets applied to four items at a time, or it could run through the items in a more efficient order than beginning to end.

Using `map()` also makes clear our intention to other coders: it will go over every item in an array and apply a function to it. With the traditional `for` loop you might have a `break` in there half-way through – something that is not possible with `map()` – and the only way to find that out is to read all the code. If you're following the Five Functional Principles I already laid down (specifically using pure functions and avoiding state), a human reading your code can know immediately that a closure used with `map()` won't try to store global state along the way.

This simplification is important, and it's a change of focus. Javier Soto – well-known functional proponent and Swift hacker extraordinaire at Twitter – sums up the usefulness of `map()` like this: it "allows us to express what we want to achieve, rather than how this is implemented." That is, we just say "here's what we want to do with the items in this array" is easier to read, write, and maintain than hand-writing loops and creating arrays by hand.

One other thing: notice how the type signature hasn't changed. That means we write `func lengthOf(strings: [String]) -> [Int]` whether or not we're using a functional approach internally. This means you can change the internals of your functions to adopt functional approaches without affecting how they interact with the rest of your app – you can upgrade your code bit by bit rather than all at once.

Examples

To help you get more comfortable using `map()`, here are some examples.

This snippet converts strings to uppercase:

```
let fruits = ["Apple", "Cherry", "Orange", "Pineapple"]
let upperFruits = fruits.map { $0.uppercased() }
```

This snippet converts an array of integer scores into formatted strings:

```
let scores = [100, 80, 85]
let formatted = scores.map { "Your score was \($0)" }
```

These two snippets use the ternary operator to create an array of strings, matching each item

against specific criteria. The first checks that each score is above 85, and the second checks that each position is within a the range 45 to 55 inclusive:

```
let scores = [100, 80, 85]
let passOrFail = scores.map { $0 > 85 ? "Pass" : "Fail" }

let position = [50, 60, 40]
let averageResults = position.map { 45...55 ~= $0  ? "Within
average" : "Outside average" }
```

Finally, this example uses the **`sqrt()`** function to calculate the square roots of numbers:

```
import Foundation
let numbers: [Double] = [4, 9, 25, 36, 49]
let result = numbers.map(sqrt)
```

As you can see, **`map()`** has that name because it specifies the mapping from one array to another. That is, if you pass it an array **`[a, b, c]`** and function **`f()`**, Swift will give you the equivalent of **`[f(a), f(b), f(c)]`**.

Optional map

To repeat what I said earlier, **`map()`** "takes a value out of a container, applies a function to it, then puts the result of that function back into a new container that gets returned to you." We've been using an array so far, but if you think about it a value inside a container is exactly what optionals are. They are defined like this:

```
enum Optional<Wrapped> {
    case none
    case some(Wrapped)
}
```

Because they are just simple containers for a single value, we can use **`map()`** on optionals too. The principle is identical: take value out of container, apply function, then place value back in the container again.

Let's start with a simple example so you can see how optional map works:

```
let i: Int? = 10
let j = i.map { $0 * 2 }
print(j)
```

That will print **`Optional(20)`**: the value 10 was taken out of its optional container, multiplied by 2, then placed back into an optional. If **`i`** had been nil, **`map()`** would just return nil. This behavior makes **`map()`** useful when manipulating optional values, particularly when combined with the nil coalescing operator.

To give you an example, consider the following function:

```
func fetchUsername(id: Int) -> String? {
    if id == 1989 {
        return "Taylor Swift"
    } else{
        return nil
    }
}
```

That returns an optional string, so we'll either get back "Taylor Swift" or nil. If we wanted to print out a welcome message – but only if we got back a username – then optional map is perfect:

```
var username: String? = fetchUsername(id: 1989)
let formattedUsername = username.map { "Welcome, \($0)!" } ??
"Unknown user"
print(formattedUsername)
```

To write that in a non-functional approach, the alternative is much longer:

```
let username = fetchUsername(id: 1989)
let formattedUsername: String
```

```
if let username = username {
    formattedUsername = "Welcome, \(username)!"
} else {
    formattedUsername = "Unknown user"
}

print(formattedUsername)
```

There is a shorter alternative that we could use, but it involves a ternary operator as well as force unwrapping:

```
let username = fetchUsername(id: 1989)
let formattedUsername = username != nil ? "Welcome, \
(username!)!" : "Unknown user"
print(formattedUsername)
```

forEach

`map()` has a close relation called **`forEach()`**, which also loops over an array and executes a function on each item. The main difference lies in the return value: **`map()`** returns a new array of items, whereas **`forEach()`** returns nothing at all – it's just a functional way to loop over each item.

This gives more information to the compiler and to readers of your code: by using **`forEach()`** you're making it clear you're not manipulating the contents of the array, which allows the Swift optimizer to do a better job.

Other than the return value, **`forEach()`** is used the same as **`map()`**:

```
["Taylor", "Paul", "Adele"].forEach { print($0) }
```

There is one other difference between **`forEach()`** and **`map()`**, which is execution order: **`forEach()`** is guaranteed to go through an array's elements in its sequence, whereas **`map()`** is free to go in any order it pleases.

Behind the scenes, **`forEach()`** literally boils down to a regular **`for-in`** loop – there's nothing special about it. Here's the internal Swift source code for **`forEach()`**, taken straight from Apple:

```
public func forEach(_ body: (Iterator.Element) throws -> Void)
rethrows {
    for element in self {
        try body(element)
    }
}
```

flatMap()

I'm not going to lie to you, **`flatMap()`** can seem daunting at first. I've put it here, directly following the surprisingly-easy-and-useful-too **`map()`** function because the two are closely related, not because I want to shock you with **`flatMap()`** so early in your functional career!

As you saw, **`map()`** takes an item out of a container such as an array, applies a function to it, then puts it back in the container. The immediate use case for this is arrays, but optionals work just as well.

When an array contains arrays – i.e., an array of arrays – you get access to a version of the **`joined()`** method that converts that array of arrays into a single array, like this:

```
let numbers = [[1, 2], [3, 4], [5, 6]]
let joined = Array(numbers.joined())
// [1, 2, 3, 4, 5, 6]
```

So, **`joined()`** reduces array complexity by one level: a two-dimensional array becomes a single-dimensional array by concatenating items.

The **`flatMap()`** function is effectively the combination of using **`map()`** and **`joined()`** in a single call, in that order. It maps items in array A into array B using a function you provide, then joins the results using concatenation. This becomes valuable when you remember that arrays and optionals are both containers, so **`flatMap()`**'s ability to remove one level of containment is very welcome indeed.

First, let's look at our friend **`map()`**:

```
let albums: [String?] = ["Fearless", nil, "Speak Now", nil,
"Red"]
let result = albums.map { $0 }
print(result)
```

Mapping using **`$0`** just means "return the existing value," so that code will print the following:

```
[Optional("Fearless"), nil, Optional("Speak Now"), nil,
Optional("Red")]
```

That's a lot of optionals, with some **nil**s scattered through. Switching to **flatMap()** rather than **map()** can help:

```
let albums: [String?] = ["Fearless", nil, "Speak Now", nil,
"Red"]
let result = albums.flatMap { $0 }
print(result)
```

Just changing **map { $0 }** to **flatMap { $0 }** changes the result dramatically:

```
["Fearless", "Speak Now", "Red"]
```

The optionals are gone *and* the nils are removed – perfect!

The reason for this magic lies in **flatMap()**'s return value: whereas **map()** will retain the optionality of the items it processes, **flatMap()** will strip it out. So, in the code below, **mapResult** is of type **[String?]** and **flatMapResult** is of type **[String]**:

```
let mapResult = albums.map { $0 }
let flatMapResult = albums.flatMap { $0 }
```

Optional flat map

If the usefulness of **flatMap()** isn't quite clear yet, please stick with me! Let's look at this example again:

```
let albums: [String?] = ["Fearless", nil, "Speak Now", nil,
"Red"]
let result = albums.flatMap { $0 }
print(result)
```

The **albums** array is of type **[String?]**, so it holds an array of optional strings. Using **flatMap()** here strips out the optionality, but really that's just the effect of joining the optional containers – it doesn't do any sort of mapping transformation.

Now, here's where **`flatMap()`** becomes awesome: because the joining happens after the mapping, you can effectively say "do something interesting with these items (map), then remove any items that returned nil (join).”

To give you a practical example, imagine a school grade calculator: students are asked to enter their scores from various exams, and it will output their estimated grade. This needs to convert user-entered numbers into integers, which is problematic because they can make mistakes or enter nonsense. As a result, creating an integer from a string returns an optional integer – it's nil if the input was "Fish", but 100 if the input was "100".

This is a perfect problem for **`flatMap()`**: we can take an array like **`["100", "90", "Fish", "85"]`**, map over each value to convert it to an optional integer, then join the resulting array to remove the optionality and any invalid values:

```
let scores = ["100", "90", "Fish", "85"]
let flatMapScores = scores.flatMap { Int($0) }
print(flatMapScores)
```

That will output "[100, 90, 85]" – perfect!

This technique can be used whenever you need to differentiate between success and failure of code by the use of optionality. To put that a bit more simply: if you have an array of input items, and a transformation function that will return either a transformed item or nil if the transformation failed, this technique is ideal.

If you think this situation crops up rarely, remember that any throwing function can be used with **`try?`**, which translates it to a function that returns nil on failure – exactly what this approach needs.

Obviously I can't predict what's in your code, but here's an example to get you started:

```
let files = (1...10).flatMap { try? String(contentsOfFile:
"someFile-\($0).txt") }
print(files)
```

That will load into an array the contents of someFile-1.txt, someFile-2.txt, and so on. Any files

that don't exist will throw an exception, which **`try?`** converts into nil, and **`flatMap()`** will ignore – all with just one line of code!

So, **`try?`** effectively turns any throwing call into a candidate for **`flatMap()`** by introducing nil. A similar trick is possible using the **`as?`** typecast, which also returns nil when it fails. For example, if you wanted an array of all the UIKit labels that belong to the current view you could write this:

```
let labels = view.subviews.flatMap { $0 as? UILabel }
```

That will attempt to typecast each subview as a **`UILabel`**, and place any items that succeeded into the returning array. Even better, Swift is smart enough to infer that **`labels`** should be of type **`[UILabel]`**.

As you can see, **`flatMap()`** brings together **`joined()`**, **`map()`**, and **`filter()`** for nil values all in one, and I hope you can see how it's useful in your own code.

filter()

The **`filter()`** method loops over every item in a collection, and passes it into a function that you write. If your function returns true for that item it will be included in a new array, which is the return value for **`filter()`**.

For example, the code below creates an array of the first 10 numbers in the Fibonacci sequence, then filters that array to include only items that are even:

```
let fibonacciNumbers = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
let evenFibonacci = fibonacciNumbers.filter { $0 % 2 == 0 }
```

So, the code will run, and pass the number 1 into the closure, making **`1 % 2 == 0`**, which is false. As a result, 1 will not be included in the result array. When it reaches 2, that *will* be included because **`2 % 2 == 0`** is true.

As with **`map()`**, the order items are filtered is out of our control – Swift is free to apply any optimizations it wants to make filtering go faster.

Examples

Here are four examples that demonstrate using **`filter()`** in different ways, starting with two I've used elsewhere:

```
let names = ["Michael Jackson", "Taylor Swift", "Michael
Caine", "Adele Adkins", "Michael Jordan"]
let result = names.filter { $0.hasPrefix("Michael") }
```

That will create an array of strings using only names that begin with Michael.

This next example returns a subset of the **`words`** array based on which words exist in the string **`input`**:

```
import Foundation
let words = ["1989", "Fearless", "Red"]
let input = "My favorite album is Fearless"
let result = words.filter { input.contains($0) }
```

I've already demonstrated **`flatMap()`**'s extraordinary ability to strip nil values from an array, but it also removes optionality from the remaining elements. If you want to strip nil but retain optionality, use **`filter()`** like this:

```
let words: [String?] = ["1989", nil, "Fearless", nil, "Red"]
let result = words.filter { $0 != nil }
print(result)
```

Finally, **`filter()`** behaves curiously when presented with a dictionary: you get back an array of tuples containing the result, rather than another dictionary. So, you should access items using array subscripting to get an item, then **`.0`** and **`.1`** for the key and value respectively. For example:

```
let scores = ["Paul": 100, "Taylor": 95, "Adele": 90,
"Michael": 85, "Justin": 60]
let goodScores = scores.filter { $1 > 85 }
print(goodScores[0].1)
```

reduce()

The **reduce()** method condenses an array into a single value by applying a function to every item. Each time the function is called, you get passed the previous value from your function as well as the next item in your array. I've already used it a couple of times elsewhere in this book, using this example:

```
let scores = [100, 90, 95]
let sum = scores.reduce(0, +)
```

That will make **sum** equal to 285 by adding 0, 100, 90, and 95. The **0** is the initial value, which is used the first time your function – in this case **+** is called. Once you become more experienced with **reduce()** you'll start to realize why functional developers lean on it so heavily!

To make the behavior of **reduce()** a little clearer, let's experiment with other possible uses. Read the code below:

```
let result = scores.reduce("") { $0 + String($1) }
print(result)
```

What might that print? **reduce()** is a generic method, just like **map()** and **filter()**, which means it can work on a variety of types. With **reduce()** you can give it an array of type S (e.g. an array of integers) and get back an array of S (e.g. **[Int]** -> **[Int]**) or of type T (i.e., Some Other Type, like strings).

The two lines of code above operate on an array of integers (that's our type S), but it starts with an initial value of an empty string, which means our type T is **String**. Remember, the S and T are just arbitrary letters meaning "some type here" – we could have used T and U, or X and Y, or “first” and “second”.

It’s worth looking at how the structure of the **reduce()** method would look when it's used with an array of integers. Here it is, adapted from the Swift source code:

```
func reduce(_ initial: T, _ combine: (T, Int) throws -> T)
```

That first **`T`** is what sets the return type, so when we use **`""`** (an empty string) for the initial value, the type signature looks a little different. If this exact method were in the Swift source code rather than using generics, it would look like this:

```
func reduce(_ initial: String, _ combine: (String, Int) throws
-> String)
```

The **`combine`** part is interesting, because it mixes types: a **`String`** will be passed in to represent the cumulative value of the function so far, and an **`Int`** will be passed in as the next item from our **`scores`** array. We need to convert that integer into a string, as the function's return value must be a string.

The body of our **`reduce()`** closure is **`$0 + String($1)`**, which means "take the current string and add to it a string containing the contents of the current integer. When it first runs, the initial value (**`""`**) is used, and added to "100", to give "100". The second time "100" is added to "90" to make "10090", because string addition means "append". The third time it's "10090" + "95", to give "1009095", which is the final result.

Try it yourself

It's time to see how well you understand **`reduce()`**: try to use it to sum the number of characters used in all the names below:

```
let names = ["Taylor", "Paul", "Adele"]
// your code here
print(count)
```

That should print "15", which is the sum of the lengths of "Taylor", "Paul", and "Adele". This is a straight integer operation, but you can't just use **`+`** because you want to add the character count rather than appending strings.

Give it a try now; below is my solution:

```
let names = ["Taylor", "Paul", "Adele"]
let count = names.reduce(0) { $0 + $1.characters.count }
```

```
print(count)
```

For comparison, here's the non-functional variant:

```
let names = ["Taylor", "Paul", "Adele"]
var count = 0

for name in names {
    count += name.characters.count
}

print(count)
```

Again, it's not really about the difference in the number of lines of code. Instead, about it's about clarity of intent: with the non-functional variant we're manipulating each item directly, and you need to read the contents of the loop to see what happens to **`count`** – it's a variable now, so it could be changed in any number of ways.

Reducing to a boolean

A common programming task is to check whether all items in array evaluate to true for a given condition: "is everyone over 18?" or "are all addresses in Canada?" This is converting an array into a single value, which is perfect for **`reduce()`**. For example, we can check that our array of names contains strings that are over four characters in length:

```
let names = ["Taylor", "Paul", "Adele"]
let longEnough = names.reduce(true) { $0 && $1.characters.count
> 4 }
print(longEnough)
```

That function starts with an initial value of **`true`**, then uses **`&&`** to compare the existing value against a new one. As a reminder:

- false && false == false
- true && false == false

- false && true == false
- true && true == true

Each time the function runs, it checks whether **`$1.characters.count > 4`**, for example **`"Taylor".characters.count > 4`**. That will return true, and our initial value was true, so Swift will run **`true && true`** to make **`true`**. The next time it runs, **`"Paul".characters.count > 4`** will return false, and our previous value was true, so Swift will run **`true && false`** to make **`false`** Finally, **`"Adele".characters.count > 4`** will return true, and our previous value was false, so Swift will run **`false && true`** to make **`false`**.

So, once the code runs **`longEnough`** will be set to false: we've successfully checked that all strings are at least a certain length.

There's an obvious but unavoidable problem here, which is that if we're checking 1000 items and item 2 fails the test, we don't need to continue. With this functional approach all 1000 items will be checked, but with a regular **`for`** loop you can **`break`** as needed. This is a cost-benefit analysis you'll need to make for each situation that comes up, but I would suggest you prefer the functional approach by default and only switch when you think it's necessary.

Reducing to find a value

Reduce can also be used to find specific values in an array, but you need to keep in mind that there are often better solutions. For example, the code below finds the longest name in the array by comparing each name against the longest so far:

```
let names = ["Taylor", "Paul", "Adele"]
let longest = names.reduce("") { $1.characters.count >
$0.characters.count ? $1 : $0 }
```

That code certainly works, but you could use **`max()`** instead, like this:

```
let longest = names.max { $1.characters.count >
$0.characters.count }
```

In this example, **`reduce()`** will do one extra comparison because of the initial value of **`""`**, however because you're starting with *something* you're always guaranteed a value to work with. In comparison, **`max()`** returns an optional value that you then need to unwrap, so which you use will depend on your context.

Reducing a multi-dimensional array

I started this chapter by saying that "the **`reduce()`** method condenses an array into a single value by applying a function to every item." That's true, but it might mean a bit more than you realized: the input can be an array of *anything* and the output can be any kind of value, even another array. To demonstrate this, I want to demonstrate using **`reduce()`** to convert an array of arrays (a two-dimensional array) into a single array.

To make this work, we need to provide **`reduce()`** with an initial value of an empty array. Then each time it's called we'll add the new item (another array) to the existing one by using **`+`**. So, the code is this:

```
let numbers = [
    [1, 1, 2],
    [3, 5, 8],
    [13, 21, 34]
]

let flattened: [Int] = numbers.reduce([]) { $0 + $1 }
print(flattened)
```

Again, that code certainly works, but it could be more succinctly expressed by using **`flatMap()`** like this:

```
let flattened2 = numbers.flatMap { $0 }
```

You could even use **`joined()`**, like this:

```
let flattened3 = Array(numbers.joined())
```

The moral here is that as your functional arsenal grows, you'll discover several ways of solving the same problem – and that's perfectly fine!

sort()

Sorting an array is a common task, and one most people are accustomed to performing using a functional approach – after all, when was the last time you tried to sort an array using a regular **`for`** loop?

In fact, sorting is so common that you often don't need to write any special code to use it: if you have an array that holds a primitive data type such as strings or integers, Swift can sort it for you just by calling **`sorted()`** with no parameters. Things only get more complicated if you want a custom sort order or have a complex data type such as a struct.

Let's tackle custom sort order first using an earlier example: a student has entered their grades into your app for you to process. An initial implementation might look like this:

```
let scoresString = ["100", "95", "85", "90", "100"]
let sortedString = scoresString.sorted()
print(sortedString)
```

However, the results are unlikely to be what you want: that program will output **`["100", "100", "85", "90", "95"]`**. This is because the elements are strings, so they are sorted character by character where 1 comes before 9 even though 100 is greater than 90.

A second implementation might be to convert the strings to integers and sort them, like this:

```
let scoresInt = scoresString.flatMap { Int($0) }
let sortedInt = scoresInt.sorted()
print(sortedInt)
```

If you were wondering why I used **`flatMap()`** rather than regular **`map()`**, it's because creating an integer from a string returns **`Int?`**, which means **`scoresInt`** would end up being **`[Int?]`** rather than **`[Int]`**. This isn't necessarily a problem, but it would mean writing more code: **`[Int]`** has a **`sorted()`** method that works with no parameters, whereas **`[Int?]`** does not. We'll look at this more soon.

A third implementation is to write a custom sort function for our string array, which would convert each value to an integer and compare one with another. This is less efficient than using

`flatMap()` then parameter-less **`sorted()`** because each element would need to be converted to an integer multiple times: Swift would compare 95 to 100, then 85 to 100, then 85 to 95, and so on, with each comparison involving re-creating the integers.

Even though it's less efficient, it's at least worth looking at the code:

```
let sortedString = scoresString.sorted {
    if let first = Int($0), let second = Int($1) {
        return first < second
    } else {
        return false
    }
}

print(sortedString)
```

Finally, let's look at using **`map()`** rather than **`flatMap()`**, which means working with optional integers. This will result in a sorted array of optional integers:

```
let scoresInt = scoresString.map { Int($0) }
let sortedInt = scoresInt.sorted {
    if let unwrappedFirst = $0, let unwrappedSecond = $1 {
        return unwrappedFirst < unwrappedSecond
    } else {
        return false
    }
}
```

That will result in **`[Optional(85), Optional(90), Optional(95), Optional(100), Optional(100)]`** – almost certainly a less useful result than using **`flatMap()`**, despite using more code. Hopefully the difference between **`map()`** and **`flatMap()`** is becoming clearer!

Sorting complex data

When you have an array that contains a custom data type, e.g. a struct with various properties, you need to tell Swift what you want to use for sorting. You can do this in two ways, and I want to explain both so you can use whichever works best for you. Here is some code to provide something for us to work with:

```
struct Person {
    var name: String
    var age: Int
}

let taylor = Person(name: "Taylor", age: 26)
let paul = Person(name: "Paul", age: 36)
let justin = Person(name: "Justin", age: 22)
let adele = Person(name: "Adele", age: 27)

let people = [taylor, paul, justin, adele]
```

Because **`people`** doesn't store primitive data types, there's no parameter-less **`sorted()`** method. So, the first way to sort the **`people`** array is simply by writing a custom closure for the **`sorted()`** method, like this:

```
let sortedPeople = people.sorted { $0.name < $1.name }
```

Easy, right? And that solution is perfectly fine when you don't need to repeat the operation more than once. However, a smarter solution is to make your struct conform to the **`Comparable`** protocol so that you can use Swift's built-in **`sorted()`** method. It's smarter because it keeps your sort logic centralized, and I recommend you add **`Comparable`** to all your structs so you can work with them more efficiently in your apps.

Note: **`Comparable`** builds on **`Equatable`**, so you're solving two problems at once by using **`Comparable`**.

To make **`Person`** conform to **`Comparable`** you first need to add it to the list of protocols, like this:

```
struct Person: Comparable {
    var name: String
    var age: Int
}
```

You then need to add two functions inside the struct: **<** and **==**. These allow us to compare and sort items, so this is where you'll add your sorting logic. For this example, I want Swift to sort our structs using the **name** property, so I'll implement this function:

```
static func <(lhs: Person, rhs: Person) -> Bool {
    return lhs.name < rhs.name
}
```

To make the **Comparable** protocol complete, you also need to implement the **==** function like this:

```
static func ==(lhs: Person, rhs: Person) -> Bool {
    return lhs.name == rhs.name && lhs.age == rhs.age
}
```

Now that Swift knows how to sort an array of **Person** instances, you can call **sorted()** without any parameters again:

```
let sortedPeople = people.sorted()
```

Reverse sorting

Sorting items in reverse is more interesting than you might think. As you might imagine, reverse sorting is just a matter of changing **<** for **>** when writing your own code, like this:

```
let sortedString = scoresString.sorted {
    if let first = Int($0), let second = Int($1) {
        return first > second
    } else {
        return false
```

```
    }
}

print(sortedString)
```

So far, so easy. Where the interest comes is when you use the built-in **`sorted()`** method, regardless of whether you're storing primitive data or structs that conform to **`Comparable`**. Number four in my list of Five Functional Principles was immutable data types, which is something that Swift does in abundance: strings, integers, arrays, structs and more are all immutable, and so represent a fixed value.

Using value types allows Swift to do all sorts of optimization, and reverse sorting is a great example. Take a look at the code below – what do you think it will print?

```
let names = ["Taylor", "Paul", "Adele", "Justin"]
print(names)

let sorted = names.sorted().reversed()
print(sorted)
```

The first **`print()`** will output **`["Taylor", "Paul", "Adele", "Justin"]`**, which is exactly what you would expect: it's an array containing four strings. But the *second* call to **`print()`** is what's clever, because it outputs this:

```
ReverseRandomAccessCollection<Array<String>>(_base: ["Adele",
"Justin", "Paul", "Taylor"])
```

That is quite clearly something else entirely: that's a *lazy* array that stores the original array as its base value and a function (reversing) alongside. This is because the array is a value type – it can't be changed – so Swift doesn't need to do the actual work of reversing items, and will instead just return values as needed. This means the **`reversed()`** method runs in constant time – O(1), if you like Big O notation – which is extremely efficient.

If you want a "real" array rather than a **`ReverseRandomAccessCollection`** – for example if you want to access items individually – you can create an array like this:

```
let sortedArray = Array(names.sorted().reversed())
print(sortedArray)
```

Function composition

The second and third of my Five Functional Principles are to use higher-order functions and make your functions pure if possible. The combination of these two allows us to build new functions by composing – i.e., putting together – existing functions. This comes in two forms: simple chaining – i.e., running one pure function and piping its output into another – and true functional composition, which is creating new functions by assembling others like Lego bricks.

Each approach is commonly used, so I'll cover them both here. Chaining really is very simple, so we'll start with that: if you can use **`map()`**, **`flatMap()`**, **`filter()`**, and so on, then chaining them is just a matter of making one call then immediately making another.

Here's some example data for us to work with, which builds an array of tuples containing basic data about some cities:

```
let london = (name: "London", continent: "Europe", population:
8_539_000)
let paris = (name: "Paris", continent: "Europe", population:
2_244_000)
let lisbon = (name: "Lisbon", continent: "Europe", population:
530_000)
let rome = (name: "Rome", continent: "Europe", population:
2_627_000)
let tokyo = (name: "Tokyo", continent: "Asia", population:
13_350_000)

let cities = [london, paris, lisbon, rome, tokyo]
```

To sum the population of all cities, you'd use **`reduce()`** like this:

```
let totalPopulation = cities.reduce(0) { $0 + $1.population }
```

To sum the population of cities located in Europe, you need to run **`filter()`** first then chain it to **`reduce()`** like this:

```
let europePopulation = cities.filter { $0.continent ==
```

```
"Europe" }.reduce(0) { $0 + $1.population }
```

To print formatted information about the biggest cities we can use **`filter()`** to select only cities with high population, call **`sorted()`** on the result, use **`prefix()`** to select the first three items, use **`map()`** to convert the city tuples into a formatted array, then use **`joined()`** to create a single string:

```
let biggestCities = cities.filter { $0.population >
2_000_000 }.sorted { $0.population >
$1.population }.prefix(upTo: 3).map { "\($0.name) is a big
city, with a population of \
($0.population)" }.joined(separator: "\n")
```

Clearly there soon comes a point where chaining functions together becomes hard to read – probably anything more than three will cause more confusion than it clears up.

Functional building blocks

Once you have a collection of small, re-usable functions in your code, you can create new functions by assembling your existing ones in specific combinations. This is called function composition, and I hope it will encourage you to ensure your functions can be composed easily!

Swift does not have a compose operator – i.e., a special operator that makes function composition easy – so most people combine functions like this:

```
let foo = functionC(functionB(functionA()))
```

Some functional programmers create a compose operator, **`>>>`**, that allows you to rewrite that code a little more clearly:

```
let foo = functionC >>> functionB >>> functionA
```

I don't find that particularly pleasant to read, because you still need to read right to left. Instead, I prefer to create a custom operator that lets you write code like this:

```
let foo = functionA >>> functionB >>> functionC
```

This isn't hard to do using operator overloading. I'll show you the code then explain how it works:

```
precedencegroup CompositionPrecedence {
   associativity: left
}

infix operator >>>: CompositionPrecedence

func >>> <T, U, V>(lhs: @escaping (T) -> U, rhs: @escaping (U)
-> V) -> (T) -> V {
   return { rhs(lhs($0)) }
}
```

That declares a new operator, **>>>**, which uses three generic data types: T, U, and V. It works with two operands, **lhs** and **rhs**, which are two functions: **lhs** accepts a parameter of type T and returns a value of type U, and **rhs** accepts a parameter of type U and returns a value of type V.

The whole **>>>** function returns a function that accepts a **T** and returns a **V** – it just merges the two steps by calling **rhs(lhs($0))**. Both functions must be declared **@escaping** because we're creating and returning a new closure that will call them later.

To make things a bit clearer, imagine our first function accepts an integer and returns a string, and the second function accepts a string and returns an array of strings. That would make **>>>** look like this:

```
func >>> (lhs: @escaping (Int) -> String, rhs: @escaping
(String) -> [String]) -> (Int) -> [String] {
   return { rhs(lhs($0)) }
}
```

Hopefully that makes its behavior a bit clearer: by combining the two functions together, the

resulting combined function accepts an integer (the input for the first function) and returns an array of strings (the output of the second function).

This operator is just syntactic sugar, but let me give you an example that demonstrates how useful it is. Consider the following code:

```
func generateRandomNumber(max: Int) -> Int {
    let number = Int(arc4random_uniform(UInt32(max)))
    print("Using number: \(number)")
    return number
}

func calculateFactors(number: Int) -> [Int] {
    return (1...number).filter { number % $0 == 0 }
}

func reduceToString(numbers: [Int]) -> String {
    return numbers.reduce("Factors: ") { $0 + String($1) + " " }
}
```

When called in a sequence, that will generate a random number, calculate its factors, then convert that factor array into a single string. To call that in code you would normally write this:

```
let result = reduceToString(numbers: calculateFactors(number:
generateRandomNumber(max: 100)))
print(result)
```

That needs to be read from right to left: **`generateRandomNumber()`** is called first, then its return value is passed to **`calculateFactors()`**, then *its* return value is passed to **`reduceToString()`**. And if you need to write this code more than once, you need to make sure you keep the order correct even when making changes in the future.

Using our new compose operator, there's a better solution: we can create a new function that combines all three of those, then re-use that function however we need. For example:

```
let combined = generateRandomNumber >>> calculateFactors >>>
reduceToString
print(combined(100))
```

There are several reasons I love this approach. First, it's read naturally: generate a random number, calculate its factors, then reduce to a string. Second, it lets you save the combined function for use as many times as you need. Third, you can compose your functions even further: **`combined`** could be used with **`>>>`** to make an even bigger function. And fourth, the combined function automatically accepts the same parameter as the first function that was composed, so we use **`combined(100)`** rather than **`combined()`**.

I hope this technique gives you another good reason to write small, modular functions: you can write as many as you want, each performing one small task, then create larger functions that combine those together in useful ways.

Lazy functions

I've covered lazy sequences already, but now that you know a bit more about functional programming I want to go over one particular complexity you need to be aware of. As I said previously, lazy sequences *can* save you time, but you need to be careful because they can generate huge amounts of work.

To demonstrate this, try the code below in a playground:

```
let numbers = Array(1...10000)
let lazyFilter = numbers.lazy.filter { $0 % 2 == 0 }
let lazyMap = numbers.lazy.map { $0 * 2 }

print(lazyFilter.count)
print(lazyFilter.count)
print(lazyMap[5000])
print(lazyMap[5000])
```

When you run that, you'll see Swift runs the **`$0 % 2 == 0`** closure 10,000 times to calculate the first **`print()`** statement, then another 10,000 times to calculate the second **`print()`** statement. This is because the result of **`filter()`** can't be known until all items have been processed – Swift has no idea how many items are in the resulting array until everything has been processed. On the other hand, the third and fourth **`print()`** calls execute immediately, because Swift can run the **`map()`** closure on a single element and ignore the rest.

This is particularly problematic when you chain functions together, because if **`filter()`** comes first in your chain you will end up having to repeat vast amounts of work.

Functors and monads

Now that you understand some functional programming, I have some good news: you can start dropping words like "functor" and "monad" into your casual office conversations – something I'm sure you're desperate to do.

First up: a functor is a container that implements `map()`. It doesn't need to be *called* `map()`, but it does need to do the same thing: convert A into B by applying a transformation. We've been using lots of arrays and optionals, both of which are functors. `map()` is *not* the functor – array is the functor, because it implements `map()`.

A monad is a functor that also implements `flatMap()`. Again, arrays and optionals are monads. The formal definition for monads in Haskell – commonly seen as the definitive reference – adds a few other rules, but when you're working with Swift just think about `flatMap()` and you're there.

So: a functor is a data type that can be mapped over, and a monad is a functor that also can be flatmapped over. That's it – go forth and dazzle your friends with your new-found vocabulary! (Actually, please don't: although the terms give us a shared vocabulary to use when discussing data types, they can also scare people who are new to functional programming.)

Chapter 7
Patterns

Wayne Bishop (@waynewbishop), author of Swift Algorithms and Data Structures

This extension has been a long time favorite:

```
extension Int {
   // iterates the closure body a specified number of times
   func times(closure: (Int) -> Void) {
      for i in 0 ..< self {
         closure(i)
      }
   }
}
```

What I like is its readability and usefulness. It's a neat alternative to default looping formats as it can also capture its surrounding state.

Object-oriented programming

Although some people wish it were not the case, object-oriented programming (OOP) is the de facto standard way of building software. It's not *everywhere*: a lot of software is still built in C (which pre-dates OOP), and as you'll see in the following chapter Swift can be written using a similar-but-different approach called protocol-oriented programming (POP).

Even when using POP, Swift still relies on the OOP underbelly of Cocoa Touch, which was written for Objective-C – a language so aggressively object-oriented that they baked "object" right into its name. If you need to write apps for iOS or OS X, you need to understand a little OOP in order to interact with the system. If you intend to interact with other developers, more often than not they will use OOP. If you intend to find examples online, more often than not they will use OOP too – it's unavoidable no matter how much you might love POP.

In this chapter I'm going to start by giving you a lightning fast introduction to OOP, including examples in Swift. If you already understand terms like "polymorphism", "encapsulation", and "static" then you're welcome to move on.

Nouns first

OOP attempts to model programs in terms of entities that have real-world equivalents. So, if you were making an app to track hospital bookings, you'd have an entity for patients, an entity for wards, an entity for the building, an entity for staff, an entity for bookings, an entity for operations, and so on. You can be as fine-grained or otherwise as you want, but the goal is to make your entities – usually classes in OOP – reflect actual things that exist in the real world.

This makes OOP easy to pick up: you can just write down a list of the nouns you care about – "patient", "doctor", "ward", and so on – then make a class for each of them. In theory this breaks up your program into small, re-usable parts that each have specific pieces of responsibility for part of your program, but it also means you make your big architectural decisions very early in your development, and sometimes these can hinder your flexibility later.

Once you have decided your nouns – i.e., created all the classes for things you care about – you add data to them in the form of properties, and add verbs to them in the form of methods.

Methods are effectively identical to functions, but they carry an implicit **`self`** parameter. For example, look at the code below:

```
class Meh {
    func doStuff(number: Int) { }
}
```

The **`doStuff()`** function belongs to the **`Meh`** class, so we call it a *method* instead. And behind the scenes, it actually looks like this:

```
func doStuff(self: Meh, number: Int) { }
```

Objects are *reference types* in Swift, which means they can have multiple simultaneous owners, any of which can manipulate the object. This creates implicit dependencies across your program, which can add complexity and fragility.

Encapsulation

Because classes have their own properties and methods, they can be said to encapsulate functionality: a **`Doctor`** class can have all the functionality required to work as a doctor in a hospital, and the rest of your program can just call your doctor's methods to make things happen.

Encapsulation is a fundamental concept in OOP, and so access modifiers exist to enforce a set of rules. Continuing with the doctor example, consider this class:

```
class Doctor {
    var name = "Gregory House"
    var currentPatient = "Taylor Swift"
}
```

Note: I'm using default values for the properties to avoid having to write pointless initializers.

That **`Doctor`** class has a **`currentPatient`** property that holds the name of the patient they are currently attending to. Now something has gone wrong elsewhere, and Dr House is

required to dash off to see another patient as a matter of priority – how should that change happen?

One option is to create a **`Hospital`** class that can force Dr House to move, something like this:

```
class Hospital {
    func assign(doctor: Doctor, to patient: String) {
        doctor.currentPatient = patient
    }
}
```

However, this is the antithesis of encapsulation: the **`Doctor`** class should contain all its own functionality, and having the **`Hospital`** class poke around in a doctor's properties will create spaghetti code. What if the doctor is in the middle of a check up? Or is on vacation?

By placing functionality inside the **`Doctor`** class you centralize your logic in one place: you can refuse to move if the current task is important, you can refuse to move if you're on vacation, and so on. This means treating objects like black boxes: you don't touch their properties directly.

So, the smarter way to assign a doctor to a new patient is like this:

```
class Doctor {
    var name = "Gregory House"
    var currentPatient = "Taylor Swift"

    func assignToPatient(_ name: String) {
        currentPatient = name
    }
}
```

You can then go ahead and add any validation directly into **`assignToPatient()`**.

When I was much younger, I wrote a small strategy game that was similar in intention – if not completion! – to the game *Civilization*. I was fairly new to programming back then, so I didn't

use encapsulation at all. When a turn ended and I wanted to figure out how much gold each city generated, I added a loop to my `Game` class that went through every city, pulled out its population, then multiplied that by the tax rate. I had to copy and paste that code into the city view so that the player could click on a city and see exactly how much money it generated, but that was OK.

When I let the player create buildings in cities to increase tax revenue, I had to modify my turn end loop: get population, multiply by tax rate, then modify by building improvements. Of course I had to copy that code into my city view calculation too, but that was only a few extra lines. Then I let players improve the squares around their cities, and some squares brought in more gold than others, so I added that to my turn end loop… but completely forgot to add it to the city view.

You should be able to see why my approach was bad, and why encapsulation would have made it go away. Duplicating code is always bad, particularly because you need to remember all the places you've made duplication so you can update them all when you make a change – something you will inevitably get wrong. Once I realized my mistake, I switched tax calculation into a single `calculateTax()` method on my `City` class, and all the logic to calculate a city's gold was contained in just one place.

Going back to the `Doctor` class, just creating an `assignToPatient()` method is insufficient to guarantee encapsulation, because other parts of your code are free to ignore the method and adjust the properties directly. Remember, that `assignToPatient()` method might contain all sorts of logic to validate whether the doctor could move, as well as other things such as updating their diary – we really need people to use it, rather than adjust `currentPatient` directly.

This is where access modifies come in: you can tell Swift that only certain parts of your code can read or write properties or call methods. There are four access modifiers you can use:

- `private` means "this can only be used by code in the same class or struct."
- `fileprivate` means "this can only be used by code in the same Swift file."
- `internal` means "this can only be used by code in my module", which is usually your app.
- `public` means "anyone can use this."

Because playgrounds are just one big source file, **`fileprivate`** is effectively useless in them. **`internal`** is mainly important when you're writing a framework, because it allows any of your code to use a property or method but any external code will be denied access.

When working with private properties, you can make a property fully private just by using the **`private`** keyword, like this:

```
private var currentPatient = "Taylor Swift"
```

If you want everyone to be able to read the property but make writing it private, you should use **`private(set)`** instead:

```
private(set) var currentPatient = "Taylor Swift"
```

Once your properties are private, you need to introduce methods to read and write them, commonly called getters and setters. The rest of your app then manipulates your objects using these getters and setters, thus ensuring everyone uses the logic you set down in your methods.

Inheritance

The classes you define in your OOP app will frequently have common properties and methods, but naturally you don't want to duplicate code so instead you can use a technique called inheritance. This lets you create a new class based on an existing one, and it imports all the properties and methods as it goes. When you create a new class this way it's called a subclass, and the class you inherited from is called the super class. You'll also hear subclasses referred to as "child" classes, which makes the class you inherited from the "parent" class. You can inherit as many times as you want, so you can have grandparent classes and even great-grandparent classes, although those names are rarely used!

Inheritance is another fundamental technique inside OOP, and it's endemic in Apple's systems. If you create a new iOS Single View Application in Xcode, you'll get a class called **`ViewController`** that inherits from **`UIViewController`**, but **`UIViewController`** itself inherits from **`UIResponder`**, which inherits from **`NSObject`**, which is the root class in Objective-C – the thing that all other classes must start with.

Inheritance is useful because it lets you import huge amounts of functionality then add to it. For example, nowhere in the **`ViewController`** class does it say how to behave when the screen is rotated, or how to load a layout from a storyboard, but all that functionality is automatically brought in by inheriting from **`UIViewController`**.

When you create class **`Surgeon`** by inheriting from class **`Doctor`**, you bring with it all of **`Doctor`**'s properties and methods, which might in turn bring in properties and methods from **`Employee`**, and so on. You can then add your own specific properties and methods to **`Surgeon`** to do things that doctors don't do, but you can also *override* existing behavior that came with the **`Doctor`** class – you can say "don't do *that*, but do *this* instead."

Overriding behavior is explicit in Swift, which is not the case in many other languages. To demonstrate this, let's make the **`Doctor`** and **`Surgeon`** classes:

```
class Doctor {
    var name = "Gregory House"
    var currentPatient = "Taylor Swift"
}

class Surgeon: Doctor { }
```

The **`Doctor`** class defines a couple of basic properties, and the **`Surgeon`** class inherits from **`Doctor`** so it gets those properties. This makes sense: everyone has a name regardless of whether they are doctors or surgeons. If we wanted to allow our surgeons to perform surgery, we could add a **`performSurgery()`** method to the surgeon class like this:

```
class Surgeon: Doctor {
    func performSurgery(isEmergency emergency: Bool) {
        if emergency {
            print("OK, let's go!")
        } else {
            print("I'll do it next")
        }
    }
}
```

So far, so easy. But if we need to allow doctors to perform surgery too, things get more complicated. Just adding **`performSurgery()`** method to the **`Doctor`** class is easy enough:

```
class Doctor {
    var name = "Gregory House"
    var currentPatient = "Taylor Swift"

    func performSurgery(isEmergency emergency: Bool) {
        if emergency {
            print("OK, let's go!")
        } else {
            print("Please find a surgeon")
        }
    }
}
```

However, now **`Doctor`** has a **`performSurgery()`** method which **`Surgeon`** inherits, then **`Surgeon`** declares its own **`performSurgery()`** method – Swift doesn't know which one to use, so it will refuse to compile.

To fix the problem, you need to make your intention clear: either delete the surgeon's **`performSurgery()`** method (so that Swift will use the doctor's **`performSurgery()`** method), change the surgeon's method so that it has a different signature (e.g. **`performSurgery()`** rather than **`performSurgery(isEmergency emergency: Bool)`**), or tell Swift that the surgeon's method overrides the existing method from **`Doctor`**, like this:

```
class Surgeon: Doctor {
    override func performSurgery(isEmergency emergency: Bool) {
        if emergency {
            print("OK, let's go!")
        } else {
            print("I'll do it next")
        }
```

```
    }
}
```

Anything you don't override will be inherited from the parent class.

Super methods

As you've seen, when you inherit one class from another, you either use the parent class's methods or you override the ones you want to change in the child. There is a third way, which lets you use the existing functionality of a parent's method and add some custom tweaks in the child.

Let's work up an example: our doctors have an **`assignToPatient()`** method like this:

```
func assignToPatient(_ name: String) {
    currentPatient = name
}
```

By default, that will be inherited by the **`Surgeon`** subclass, but we want to make surgeons do something special: when we call **`assignToPatient()`** on **`Surgeon`** we first want them to ask some junior doctors for a diagnosis so they have a chance to learn, *then* we want to call the existing **`assignToPatient()`** method on **`Doctor`**.

Swift makes this easy using the **`super`** keyword, which works like **`self`** but for parent classes. If the **`Surgeon`** class overrides **`assignToPatient()`** to add its own functionality, it can at any time call **`super.assignToPatient()`** to have the code in the **`Doctor`** class execute. This means you don't need to duplicate any code: **`Surgeon`** handles the parts it cares about, then lets **`Doctor`** do the rest. In code, it looks like this:

```
class Surgeon: Doctor {
    override func assignToPatient(_ name: String) {
        print(getJuniorDoctorDiagnosis())

        super.assignToPatient(name)
    }
```

```
    func getJuniorDoctorDiagnosis() -> String {
        return "He's dead, Jim"
    }
}
```

The only complication is when working with **`init()`**, which has some strict rules. Specifically:

- If you create a class using inheritance, you must call **`super.init()`** in the child class to allow the parent to construct itself.
- Your subclass must have initialized its own properties fully before calling **`super.init()`**.
- You can't touch properties from your parent class until you have called **`super.init()`**.
- You can't call any other methods until you have initialized all your properties, including calling **`super.init()`**.

To illustrate the correct layout for initializing a subclass, here's a modified **`Doctor`** class with a simple initializer:

```
class Doctor {
    var name: String
    var currentPatient: String

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

If **`Surgeon`** doesn't introduce any new properties of its own, it will use the existing **`Doctor`** initializer. To make things more interesting, let's allow surgeons to have their own title. A curiosity in the UK is that most doctors are referred to using the title "Doctor", but the most

senior ones – consultants – prefer to be referred to as Mr, Mrs, Miss, or Ms even though they are also fully qualified doctors. So if I were a regular doctor I'd be "Dr Hudson," but if I were a consultant I would be "Mr Hudson."

Let's modify our **`Surgeon`** class to allow a custom title to be used, as they are senior doctors. Here's the code:

```
class Surgeon: Doctor {
    var title: String

    init(name: String, title: String, currentPatient: String) {
        self.title = title
        super.init(name: name, currentPatient: currentPatient)
    }
}
```

Notice that **`self.title`** gets initialized *before* **`super.init()`** gets called. If you try to put these two lines the other way around, Swift will refuse to build your code. If you try to adjust **`self.name`** before calling **`super.init()`**, Swift will refuse to build your code. If you remove **`super.init()`** or try to call any other methods before using **`super.init()`**, Swift will refuse to build your code – it's extremely strict!

Polymorphism

Polymorphism is a word pulled from two Greek words that mean "many shapes", and in OOP it means an object can be used like its own class or any of the classes it inherits from. It's best explained using an example, so below is an array of **`Doctor`** objects that lists which doctors are available for work right now:

```
var availableDoctors = [Doctor]()
availableDoctors.append(Doctor())
availableDoctors.append(Doctor())
availableDoctors.append(Doctor())
```

In that code, the **`availableDoctors`** variable is an array containing **`Doctor`** objects. But

Surgeon inherits from **Doctor**, so as far as Swift is concerned it is a **Doctor** too. This means we can add surgeons to the array directly, like this:

```
var availableDoctors = [Doctor]()
availableDoctors.append(Doctor())
availableDoctors.append(Doctor())
availableDoctors.append(Surgeon())
availableDoctors.append(Doctor())
```

Swift knows this is safe, because **Surgeon** inherits from **Doctor** and so either inherits its properties and methods or overrides them. For example, we can loop over the array and call **performSurgery()** on each item, and Swift will call the correct version of **performSurgery()** depending on each object it finds:

```
for doctor in availableDoctors {
    doctor.performSurgery(isEmergency: false)
}
```

That will output the following:

```
Please find a surgeon
Please find a surgeon
I'll do it next
Please find a surgeon
```

Polymorphism allows the **Surgeon** object to work simultaneously like a **Doctor** and like a **Surgeon**: it can added to an array of doctors, but use the **Surgeon** version of **performSurgery()** thanks to a technique called dynamic dispatch. This would *not* work if we declared **availableDoctors** to have the data type **[Surgeon]** because you can't add a parent class (**Doctor**) to an array of the child class (**Surgeon**).

Final classes

It can be really helpful to inherit from a class and modify the way it works, particularly because it lets you build complex objects quickly. However, some classes need to work a

specific way, and it might be confusing and/or risky to modify that behavior. For example, if you created a banking framework that others could integrate into their apps, would you really want to let them subclass your types and modify the way they worked? Probably not!

Swift lets you mark classes using the **`final`** keyword, which means "don't let anyone subclass this." So, this code will not compile:

```
final class Doctor { }
class Surgeon: Doctor { }
```

You can also mark individual properties and methods if you want, which allows someone to inherit from your class but not modify specific parts of it.

Because of the way Swift is built, there is a potential performance benefit to declaring things as **`final`**. Unless you enable whole module optimization, Swift needs to perform extra checks before making a method call because it needs to check whether it has been overridden somewhere else. Marking properties, methods, or even whole classes as **`final`** eliminates this check, so your code will run a little faster.

Unless you have specifically designed your class to be inheritable, you should mark it **`final`**.

Class properties and methods

Sometimes you will want to create properties and methods that belong to a class rather than an object. The difference might seem subtle, but it's important: a regular property or method belongs to an instance of the **`Doctor`** class, e.g. Gregory House, whereas a class property or method is shared across all doctors.

For example, you might create a method that recites the Hippocratic oath – a medical promise that many physicians take when they graduate medical school. This oath is the same for all doctors, so instead we can create a single method that can be called directly on the **`Doctor`** class. To do this, just write **`static`** before your method, like this:

```
class Doctor {
    static func quoteHippocraticOath() {
        print("I will prevent disease whenever I can, for
```

```
prevention is preferable to cure.")
    }
}

Doctor.quoteHippocraticOath()
```

When you use **`static func`**, you're making a class method that also happens to be **`final`**. If you want to allow subclasses to override the method, you can use **`class func`** instead of **`static func`**.

Static properties are also possible, and again just require the **`static`** keyword:

```
static let latinTitle = "Medicinae Doctor"
```

is-a vs has-a

Protocol-oriented programmers sometimes like to think they have a monopoly on composition, which isn't true: many OOP developers have been encouraging composition rather than inheritance for some time, although it's fair to say the uptake has been patchy.

In OOP, the contrast between inheritance and composition is often called "is-a vs has-a". A surgeon *is a* doctor, so it makes sense to inherit **`Surgeon`** from **`Doctor`**. A surgeon *has a* scalpel, which means the **`Surgeon`** class could have a property called **`scalpel`**, but it would be strange if **`Surgeon`** inherited from **`Knife`**.

In the unlikely event you find yourself unsure whether to create a new class using inheritance or by adding a property, try using the "is-a / has-a" comparison and see if it helps.

Protocol-oriented programming

Swift 2.0 introduced some major changes to the way protocols work, and in doing so introduced a new way to build software dubbed protocol-oriented programming (POP). It is not entirely new – far from it! – and indeed builds upon Swift techniques you should already be familiar with: protocols, extensions, value types, and more. This means Swift blends OOP, POP, and functional approaches all in a single language, so it has something for everyone.

POP uses nearly all the same tools you would use with OOP, so you should already have a sound understanding of OOP to continue. It follows the same principles of encapsulation and polymorphism from OOP, but it does away with inheritance. Instead, POP encourages you to build your app architecture horizontally rather than vertically: you add functionality by adopting protocols rather than inheriting from parent classes. A large part of the approach comes through the protocol extension syntax introduced in Swift 2.0, which lets you add functionality in smart, focused ways.

Protocol extensions combine two previous technologies, which are – surprise! – protocols and extensions. A protocol is a promise of functionality: the compiler will ensure that anyone who wants to conform to the protocol must implement all the methods you specify. For example, we could create a **`Payable`** protocol that requires a **`calculateWages()`** method to be implemented in any conforming data type:

```
protocol Payable {
    func calculateWages() -> Int
}
```

When an object adopts a protocol like **`Payable`**, it means you can call **`calculateWages()`** on it and expect to get an integer back. What the actual method does internally is an implementation detail, but that's not a bad thing: it means you can use any data type that conforms to **`Payable`** without wondering about what their code does, as long as they stick to the API contract – as long as they implement **`calculateWages()`** with the specified type signature.

Extensions allow you to add functionality to specific types. For example, you can add a **`squared()`** method to integers like this:

```
extension Int {
    func squared() -> Int {
        return self * self
    }
}
```

That will then apply to all **`Int`** values, so you can call it like this:

```
let number: Int = 5
let result = number.squared()
```

Protocols have the disadvantage that they contain only declarations of methods; if you try to include any functionality in a protocol, the compiler will refuse to build your code. Extensions *can* add functionality, but they only apply to specific data types – we just extended **`Int`**, but not any of the other integer types. So code like this won't work because it uses a **`UInt`** rather than an **`Int`**:

```
let number: UInt = 5
let result = number.squared()
```

Protocol extensions combine the two: they let you provide complete default implementations for functions in the same way as extensions, but apply at the protocol level rather than the individual data type level. So, we could extend *all* integers to add a **`squared()`** method like this:

```
extension Integer {
    func squared() -> Self {
        return self * self
    }
}
```

These implementations are *default* implementations, which means they exist as standard but can be overridden if you need. Note that **`squared()`** returns **`Self`** with a capital S: this means "return whatever data type I'm being used with."

Protocol extensions in detail

Protocol extensions let us add functionality to whole classes of data types, either by creating new protocols and extensions, or just by adding a protocol to an existing type. But it does have a few complexities that you must be aware of before starting, so let's get them out of the way now.

First, if you're creating a new protocol you need to separate creating the protocol and adding functionality. For example, if we wanted to create an **`Payable`** protocol that had a default implementation of the **`calculateWages()`** method, we would need to create the protocol then separately extend it, like this:

```
protocol Payable {
    func calculateWages() -> Int
}

extension Payable {
    func calculateWages() -> Int {
        return 10000
    }
}
```

Second, Objective-C is not aware of any protocol extensions you build, which means you can't create extensions for UIKit protocols like **`UITableViewDataSource`** and expect them to work. To be more specific, you're welcome to *create* the extensions, but UIKit won't be able to see them – they'll have no practical effect.

It also means that if you create any **`@objc`** protocols your extensions will be ignored by Swift too. For example, in the code below the extension is ignored:

```
@objc protocol Payable {
    func calculateWages() -> Int
}

extension Payable {
```

```
    func calculateWages() -> Int {
        return 10000
    }
}
```

Third, this separation of declaration (listing the type signature in the protocol) and the definition (writing the actual code in the extension) has meaning: if you do things slightly differently, your code will behave differently and perhaps even unexpectedly. Consider the code below, which creates an **`Payable`** protocol and extends it to provide a default implementation of the **`calculateWages()`** method:

```
protocol Payable {
    func calculateWages() -> Int
}

extension Payable {
    func calculateWages() -> Int {
        return 10000
    }
}
```

To test that out, we can create an empty **`Surgeon`** struct, then extend it so that it uses our **`Payable`** protocol, like this:

```
struct Surgeon { }

extension Surgeon: Payable { }
```

Just doing that is enough for surgeons to have the **`calculateWages()`** method – we retroactively declare that the **`Surgeon`** struct builds on the **`Payable`** protocol, and don't need to add any custom code to get all the **`Payable`** functionality for free. So this will print out "10000":

```
let gregory = Surgeon()
gregory.calculateWages()
```

Remember, the **`calculateWages()`** method we added to **`Payable`** is just a default implementation, so any conforming data type can write its own version that does something different, like this:

```
extension Surgeon: Payable {
    func calculateWages() -> Int {
        return 20000
    }
}
```

Here is where things get complicated: when you list a method in your protocol, you're making it available as a customization point for people who want to write their own implementations. If you *don't* list a method in your protocol, but *do* provide a default implementation, that implementation might be used even if it has been overridden later on. What matters is the data type you're using. Yes, this is confusing, but I hope the code below will explain:

```
protocol Payable {
    // func calculateWages() -> Int
}

extension Payable {
    func calculateWages() -> Int {
        return 10000
    }
}

struct Surgeon { }

extension Surgeon: Payable {
    func calculateWages() -> Int {
        return 20000
    }
}
```

```
let gregory = Surgeon()
gregory.calculateWages()

let doogie: Payable = Surgeon()
doogie.calculateWages()
```

The protocol is empty because I commented out its declaration of **`calculateWages()`**. There's a default implementation provided for this in the **`Payable`** extension, and another implementation inside the **`Surgeon`** extension. They both return different values: regular employees get paid 10000, but surgeons get paid 20000.

Take a look at the last four lines, because they print different things: even though both **`gregory`** and **`doogie`** are instances of the **`Surgeon`** struct, **`doogie`** is being stored as an **`Payable`**. This is polymorphism, but it *matters*: when there is no protocol declaration of a method, Swift will decide which method to call based on the data type it sees. In this case, **`gregory`** will use the **`calculateWages()`** method of **`Surgeon`**, whereas **`doogie`** will use the method of **`Payable`**. If you uncomment the **`calculateWages()`** declaration in the **`Payable`** protocol, both **`gregory`** and **`doogie`** will get paid the same.

So, including a method in a protocol declaration is the signal to Swift that we want to allow any default implementation to be overridden in the situation when we refer to an instance by one of its protocols rather than its specific type.

My advice: until you're comfortable with this distinction, always declare your methods in your protocols.

Thinking horizontally

Now that you understand how protocol extensions work, let's look at protocol-oriented programming itself. I said that "POP encourages you to build your app architecture horizontally rather than vertically," and this is really the fundamental difference between OOP and POP. "Vertical architecture" is inheritance: creating a base class, then adding functionality through subclassing until you get to your most specific data types. "Horizontal architecture" is using whichever protocols make sense to add specific pieces of functionality.

POP still allows you to use inheritance if you want to, but it's likely you won't need it. As a result, POP developers usually rely on structs rather than classes, which brings its own set of benefits. When working with inheritance, you can inherit from exactly one superclass, and doing so brings with it all the properties and methods of that superclass whether or not you need them. When you switch to protocols, you can adopt as many protocols as you need, and each one can add one or more methods that provide particular functionality.

This is a huge difference: rather than your superclasses dictating what methods their child classes will receive – a process that could easily result in complex, fragile hierarchies – the *child* classes dictate what functionality they want to bring in by selectively adding the protocols they want. Does your `Surgeon` class need to inherit from `Doctor`, which itself inherits from `Payable`, just to get paid? With POP, all you need to do is add a `Payable` protocol that `Surgeon` conforms to, and you get all the functionality immediately. In Swift, the advantage of POP is extended even further, because protocols can be adopted by enums as well as classes and structs, so you can share functionality wherever it's needed.

If you're an experienced developer, POP might sound like multiple inheritance – a technique that let you create classes by inheriting from more than one thing at a time, thus creating monsters like `Pegasus` by inheriting from the classes `Horse` and `Bird` at the same time. There is a subtle difference: protocols let you add only methods and computed properties, not stored properties. This means they can never add more *state* to your data types, so they keep complexity low.

Splitting up your functionality into distinct logical components conveys two further benefits. First, your code becomes significantly more testable: when each protocol handles one specific behavior (e.g. validating user input), you can write tests for that behavior and be confident that you have 100% coverage. Second, you can easily change your mind about your architecture long after you designed it just by adjusting the protocols you're using – it's significantly easier than trying to change class `C` when you know that `D`, `E`, `F`, and `G` all inherit from it.

POP in practice

At this point you should understand the core tenets of OOP – most of which apply in POP – as well as how protocol definitions work and why horizontal architecture is important. Now it's

time to look at a specific example of POP in action; I'm going to define six protocols with accompanying extensions. To keep things simple, I'm only going to give each protocol a single method. Here's the code:

```
protocol Payable {
    func calculateWages() -> Int
}

extension Payable {
    func calculateWages() -> Int {
        return 10000
    }
}

protocol ProvidesTreatment {
    func treat(name: String)
}

extension ProvidesTreatment {
    func treat(name: String) {
        print("I have treated \(name)")
    }
}

protocol ProvidesDiagnosis {
    func diagnose() -> String
}

extension ProvidesDiagnosis {
    func diagnose() -> String {
        return "He's dead, Jim"
    }
}
```

```
protocol ConductsSurgery {
    func performSurgery()
}

extension ConductsSurgery {
    func performSurgery() {
        print("Success!")
    }
}

protocol HasRestTime {
    func takeBreak()
}

extension HasRestTime {
    func takeBreak() {
        print("Time to watch TV")
    }
}

protocol NeedsTraining {
    func study()
}

extension NeedsTraining {
    func study() {
        print("I'm reading a book")
    }
}
```

Those six protocols give us enough data to start modeling some staff members in a hospital. Specifically, we're going to define four roles: receptionist, nurse, doctor, and surgeon. In a real hospital the lines between these roles aren't black and white – roles like nurse practitioner exist that can diagnose and prescribe treatment, for example – but I'm trying to keep it simple.

So, with the protocols in place, we can create four structs:

```
struct Receptionist { }
struct Nurse { }
struct Doctor { }
struct Surgeon { }
```

And now for the best bit: to give those roles the functionality we defined in our protocols, we just need to decide who should get what. We don't need to create an inheritance tree – we just need to pick which protocols each role needs. For example, the **`Receptionist`** role should probably adopt the protocols **`Payable`**, **`HasRestTime`**, and **`NeedsTraining`**, like this:

```
extension Receptionist: Payable, HasRestTime, NeedsTraining {}
```

The **`Nurse`** role needs the same roles as **`Receptionist`** as well as **`ProvidesTreatment`**:

```
extension Nurse: Payable, HasRestTime, NeedsTraining,
ProvidesTreatment {}
```

The **`Doctor`** role needs the same roles as **`Nurse`** with the addition of **`ProvidesDiagnosis`** because they can diagnose as well as treat:

```
extension Doctor: Payable, HasRestTime, NeedsTraining,
ProvidesTreatment, ProvidesDiagnosis {}
```

The **`Surgeon`** role is a little different: the surgeons in our example hospital won't provide treatment (they have junior doctors to do that), but they can provide diagnosis as well as conduct surgery. So, they look like this:

```
extension Surgeon: Payable, HasRestTime, NeedsTraining,
ProvidesDiagnosis, ConductsSurgery {}
```

That completes our line up. Notice how we can be really specific about what each role needs, rather than try to craft a hierarchy? If we had made **`Surgeon`** inherit from **`Doctor`** it would

have been given the ability to provide treatment as part of that inheritance, and we would need to override that functionality to try to opt out.

However, our current solution is a bit messy: all staff members adopt the protocols **`Payable`**, **`HasRestTime`**, and **`NeedsTraining`**, so there's a lot of duplication right now. To make things easier to read, we can create a new **`Employee`** protocol that groups together **`Payable`**, **`HasRestTime`**, and **`NeedsTraining`**, then make our four roles adopt **`Employee`** rather than the individual protocols. In code, it looks like this:

```
protocol Employee: Payable, HasRestTime, NeedsTraining {}

extension Receptionist: Employee {}
extension Nurse: Employee, ProvidesTreatment {}
extension Doctor: Employee, ProvidesDiagnosis,
ProvidesTreatment {}
extension Surgeon: Employee, ProvidesDiagnosis, ConductsSurgery
{}
```

Apple's advice for POP is this: "don't start with a class, start with a protocol." And that's exactly what we've done here: we started by defining the various behaviors we wanted to represent in our app, then created structs that adopted those behaviors to bring it all to life.

Constrained extensions

We've already seen how we can extend a protocol to add new functionality. For example, we could add a **`checkInsurance()`** method to all roles that adopt the **`Employee`** protocol, like this:

```
extension Employee {
    func checkInsurance() {
        print("Yup, I'm totally insured")
    }
}
```

But that's a pretty big change to make, and it brings back the same problem that inheritance

gave us: everyone gets this method whether they want it or not. Sometimes this just adds extra unnecessary complexity, but other times it will stop your code from building because the method you want to add doesn't work on every **`Employee`**.

Swift's solution to this problem is called constrained extensions, and allows us to specify that an extension applies only to certain kinds of protocols. In our current instance, we only need staff to have insurance if they provide treatment, so we can rewrite the above code like this:

```
extension Employee where Self: ProvidesTreatment {
    func checkInsurance() {
        print("Yup, I'm totally insured")
    }
}
```

Using this approach, employees that don't provide treatment, or people who provide treatment that aren't employees (e.g. a doctor who is on loan from another hospital to cover vacation time), won't be given the **`checkInsurance()`** method.

Things get more complicated when you want to handle sub-types of data, for example if you want to target any kind of collection that stores strings. To make this work, you need to specify a **`where`** clause to filter by **`Iterator.Element`** where the element type matches something specific. You can use **`Iterator.Element`** inside the method as a placeholder for the matching data type, or you can use specific data types if you prefer.

For example, the extension below targets collections that store integers, and adds the ability to count how many odd and even numbers are in the collection:

```
extension Collection where Iterator.Element: Integer {
    func countOddEven() -> (odd: Int, even: Int) {
        var even = 0
        var odd = 0

        for val in self {
            if val % 2 == 0 {
                even += 1
```

```
            } else {
                odd += 1
            }
        }

        return (odd: odd, even: even)
    }
}
```

Example extensions

As you've seen Swift's extension system is remarkably powerful: you can extend a specific type or whole groups of types all at once, plus you can build constrained extensions that work on specific subsets.

In the event that you create two methods of the same signature and try to give them both to a single data type, Swift will automatically choose the one that is the most constrained – i.e., the most specific. So, an extension for **`Int`** would be considered more specific than an extension for all **`Integer`** data types.

I want to demonstrate Swift extensions by building up some examples, starting with one you've seen already. I've written ten examples in all, built around the following four protocols:

- The **`Equatable`** protocol covers things that can be compared using **`==`** and **`!=`**.
- The **`Comparable`** protocol covers things that can be compared using **`<`** and **`>`**.
- The **`Collection`** protocol covers things that store multiple values, like arrays.
- The **`Integer`** protocol covers things that hold round numbers like 1, 2, and 128.

I'll walk you through all ten examples, but if you want to get the maximum benefit I'd like you to to attempt the even-numbered examples yourself before reading my answers. I'll be using the odd-numbered examples to introduce new concepts, and the even-numbered examples to help you try things out for yourself.

Example one: Squaring integers

Let's start by creating an extension for the Int type that will calculate the value squared. This is a regular extension that we've looked at previously, so this should just be a refresher:

```
extension Int {
    func squared() -> Int {
        return self * self
    }
}

let i: Int = 8
print(i.squared())
```

Because we extended the **`Int`** type, that extension won't be visible in other similar data types. For example, this won't compile:

```
let j: UInt = 8
print(j.squared())
```

To make that work, we need to extend the **`Integer`** protocol, which will affect all types of integer – **`Int`**, **`UInt`**, **`Int64`**, etc:

```
extension Integer {
    func squared() -> Self {
        return self * self
    }
}
```

Notice the Self with a capital S – it means "whatever data type we're being used with," which is different to "self" with a lowercase s. So, **`Self`** means **`Int`** when working with that data type, or **`Int64`** when working with that instead, whereas **`self`** (lowercase S!) means the actual number being stored.

Example two: Clamping integers

Now it's time for you to try creating an extension for yourself. Don't worry – I'll provide the

answer below. But I do want you to try solving it yourself, to make sure you understand how extensions work.

Here's your goal: create an extension for all integer types that returns the value clamped between two values. So, if you feed it the number 15 and ask it to clamp between 1 and 10, you'll get back 10. Here's some example input and output so you can test your code:

- Minimum 5, maximum 10, input 8 should output 8.
- Minimum 5, maximum 10, input 3 should output 5.
- Minimum 5, maximum 100, input 800 should output 100.

Remember, you're extending **`Integer`**, so your input number will be available inside **`self`**. You just need to write the method that accepts low and high values for clamping.

Please try now, then check my answer below to see how you got on:

```
extension Integer {
    func clamp(low: Self, high: Self) -> Self {
        return min(max(self, low), high)
    }
}
```

Don't worry if your answer was different to mine – there are lots of ways of solving this problem, and as long as yours is easy to read and efficient to run it's a good solution.

Example three: Matching value types

Swift uses value types almost everywhere, which means you can compare most things of the same type using a straight **`==`**. This isn't magic; instead, it's the **`Equatable`** protocol providing that **`==`** functionality: everything that conforms to this protocol can be checked for equality with another item of the same type.

We're going to add an extension to **`Equatable`** so that one value can be checked against an array of values of the same type in one method. So, **`2.matches(array: [2, 2, 2, 2])`** will return true, but **`2.matches(array: [2, 2, 2, 3])`** will return false.

Here's the code:

```
extension Equatable {
    func matches(array items: [Self]) -> Bool {
        for item in items {
            if item != self {
                return false
            }
        }

        // if we made it this far, all items match!
        return true
    }
}
```

Example four: Comparing arrays

You've seen how to create an extension for **`Equatable`** that works on data types that can be checked for equality using **`==`**. Now I'd like you to try your hand at writing an extension for **`Comparable`**, which works on data types that can be ordered using **`<`** and **`>`**.

You're probably thinking of numbers right now, and it's true that they conform to **`Comparable`**. But so do strings: "alpha" is ordered before "omega" because they can be sorted lexicographically.

Your job is to create a method that returns true if a value is less than all other values in an array. Here are some example calls and their result:

- Running **`5.lessThan(array: [6, 7, 8])`** should return true.
- Running **`5.lessThan(array: [5, 6, 7])`** should return false.
- Running **`"cat".lessThan(array: ["dog", "fish", "gorilla"])`** should return true.

This is an even-numbered example, so I would like you to try it yourself before reading my example. You need to use the **`Comparable`** protocol, but other than that my code from

example three should be enough to get you through.

All done? Here's my solution:

```
extension Comparable {
    func lessThan(array items: [Self]) -> Bool {
        for item in items {
            if item <= self {
                return false
            }
        }

        return true
    }
}
```

Example five: Rewriting contains()

You've been extending specific protocols so far by using **`Integer`**, **`Equatable`**, and **`Comparable`**, but I want to take things a step further now: I want to create an extension for *collections* that hold specific data types.

More specifically, we're going to recreate the **`contains()`** method that exists in Swift's standard collections, and I think you'll find it's straightforward. The method needs to loop over every item in an array until it finds one that matches – i.e., returns true for **`==`** - and input object. This means using the **`Equatable`** protocol again, which is what guarantees us support for **`==`**.

This time we'll be extending the **`Collection`** protocol, but we need to constrain it so that it applies only to collections that contain equatable elements. In Swift this is done by adding a constraint to **`Iterator.Element`** – the iterator is the thing that returns one element at a time when you move through a sequence, so what we're saying is "the type of object that the iterator stores must be equatable."

Here's the code:

```
extension Collection where Iterator.Element: Equatable {
    func myContains(element: Iterator.Element) -> Bool {
        for item in self {
            if item == element {
                return true
            }
        }

        return false
    }
}
```

Example six: Fuzzy array matching

Now that you've seen how to constrain protocols based on the element type that a collection holds, your next challenge is to create a method that returns true if two arrays contain the same elements, regardless of their order.

So:

- Using `[1, 2, 3].fuzzyMatches([1, 2, 3])` will return true
- Using `[1, 2, 3].fuzzyMatches([3, 2, 1])` will return true
- Using `[1, 2, 3].fuzzyMatches([1, 2])` will return false
- Using `[1, 2, 3].fuzzyMatches([1, 2, 3, 1])` will return false

Again, please give it a try yourself.

Done? Here's my solution:

```
extension Collection where Iterator.Element: Comparable {
    func fuzzyMatches(_ other: Self) -> Bool {
        let usSorted = self.sorted()
        let otherSorted = other.sorted()

        return usSorted == otherSorted
```

```
    }
}
```

As you can see, I'm taking advantage of the fact that Swift's own **==** operator can directly compare two arrays, so all we need to do is ensure they are in the same order before doing the comparison!

Note: I wrote **self.sorted()** rather than just **sorted()** for the sake of clarity. The **self.** isn't needed, but it makes the intent a bit clearer.

Example seven: average string length

The previous two examples extended protocols where their items *also* conformed to a protocol. We can also extend protocols where items are a specific data type by using **==** for the constraint rather than **:**.

To demonstrate this, we're going to extend the **Collection** type for collections that hold strings (a specific data type) by writing a method that returns the average length of all the strings. This highlights a curiosity in the way Swift's collections work, so pay attention!

Here's the code:

```
extension Collection where Iterator.Element == String {
    func averageLength() -> Double {
        var sum = 0
        var count = 0

        self.forEach {
            sum += $0.characters.count
            count += 1
        }

        return Double(sum) / Double(count)
    }
}
```

The algorithm itself is just a mean average, which is straightforward enough. But notice how I keep track of a `count` variable inside the `forEach()` call – how come?

Swift's collections *do* have a built-in `count` property, but it's not as useful as you might hope. You see, it's easy to think of collections as being things like `[1, 2, 3, 4, 5]` or dictionaries of strings and so on, but in practice they can be a far wider range of things.

In fact, as long as the iterator keeps producing values, a collection can effectively be infinite. For example, you could write an iterator that outputs the Fibonacci sequence without having to store the entire sequence – it just needs to store a couple of previous values and have an iterator that puts them together. It doesn't have an end because it can always generate larger numbers, so it doesn't have a `count` equal to the number of elements it contains. So, in our code we keep track of our own `count` variable to work around this.

Example eight: counting integers

Now you've seen how to constrain collections to a specific element type, it's time for another problem you need to solve: create an extension that accepts an array of integers and returns how many times the digit 5 appears in any number.

For example:

- The array `[5, 3, 5, 1, 5]` should return 3.
- The array `[5, 15]` should return 2.
- The array `[5, 15, 55, 555]` should return 7.
- The array `[555, 555, 555]` should return 9.

So, you're counting how many times the digit 5 appears in any number, i.e. 5, 55, 515, and so on.

This is an even-numbered example, so I would like you to try it yourself. Remember, you need to use `==` to constrain to a specific data type, in this case `Int`.

All done? Here's my solution:

```
extension Collection where Iterator.Element == Int {
    func numberOf5s() -> Int {
        var count = 0

        for item in self {
            let str = String(item)

            for letter in str.characters {
                if letter == "5" {
                    count += 1
                }
            }
        }

        return count
    }
}
```

Example nine: De-duping an array

You've seen how to extend the **`Collection`** protocol where its elements are a specific type, but you can also do the opposite: extend a *fixed* collection type where its elements conform to a particular protocol.

For example, we could create a method that de-dupes an array. We already implemented our own **`myContains()`** method and in doing so learned that **`Equatable`** is what provides the **`==`** operator for a given data type. So, in this example we can require that our array holds elements that conform to **`Equatable`**, and in doing so we'll automatically get the built-in **`contains()`** method.

Time for some code, and again the algorithm isn't too hard: create an empty array that will hold our return value, then loop through every item in **`self`** – i.e., the current collection. If the result array doesn't already contain the item we should add it, otherwise just move on. Finally, return the result array.

Here's the code:

```
extension Array where Element: Equatable {
    func uniqueValues() -> [Element] {
        var result = [Element]()

        for item in self {
            if !result.contains(item) {
                result.append(item)
            }
        }

        return result
    }
}
```

There are two things to note there. First, we're extending the `Array` type specifically, so there's no more `Iterator` – it's just a group of values. Second, I declared the result array to be of type `[Element]`, which means it will match whatever the input array held.

Example ten: Array is sorted

At this point you should be fairly comfortable with extensions, so here's a simple challenge to ensure you're able to extend a specific data type where elements conform to a protocol. Your final mission: write a method that returns true if an array is sorted.

So:

- The array `[1, 3, 5]` should return true.
- The array `[5, 3, 1]` should return false.

This shouldn't trouble you – go ahead and try now before reading my solution below.

Finished? Here you go:

```
extension Array where Element: Comparable {
```

```
    func isSorted() -> Bool {
        return self == self.sorted()
    }
}
```

Again, that takes advantage of Swift's built in **==** operator for **Comparable** arrays – this stuff really is powerful, I think you'll agree.

That's the last of my ten examples, and I hope you've managed to complete the even-numbered ones yourself without too much pain. Don't be worried if your solution was more complex than mine, or even if yours was much simpler – there are lots of ways of solving each of these, and as long as your code is clear and efficient then it's a good solution.

POP vs OOP

As I've said a couple of times now, POP is a fresh take on OOP rather than a wholly new way of writing software. This makes it easy to learn and easy to transition to, because you do it bit by bit rather than having to undertake a massive rewrite. You can also carry on using industry-standard techniques such as encapsulation and polymorphism, and the extra emphasis on structs rather than classes gives you yet another reason to move to value types.

Honestly, I think it's almost no contest: once you understand how POP works, and you get started with protocol extensions, you'll quickly see benefits in code simplicity, testability, reusability, and lots of other -ities that will help you deliver software more efficiently.

The one time where the rump of OOP remains - objects and inheritance – is likely dealing with UIKit, where subclassing things like **UIViewController** and **UIView** is the standard, and this is unlikely to change in the near future.

Remember, though, that classes can adopt protocol extensions too, so you could easily create a protocol for input validation, a protocol for network requests, a protocol for loading and saving settings, and more – even moving a few small chunks out of your view controller and into protocols will immediately help reduce the size of your view controllers.

MVC

MVC – Model, View, Controller – is the standard approach for building software on iOS. Although it has its fair share of drawbacks, it is easy to learn and so endemic to iOS development that you're never far from a dozen or more code examples for any given topic.

In MVC, every component in your app is one of three things: a Model (something that describes your data), a View (something that describes your layout), or a Controller (the glue that brings model and view together into a working app). The concept might seem simple enough, but it's made complicated by the fact that iOS blends views and controllers into "view controllers", and also because some things – e.g. code to access the network – don't really fit into M, V, or C.

Let's keep it simple for now, and explore each component of MVC in more detail.

Models

Models provide storage for your data, as well as any business logic such as validation. Sometimes it can handle transforming its data into something more useful, for example translating a date into a specific format. It should *not* be responsible for rendering that to the screen, so if you see code like this you know someone needs to brush up on their MVC:

```
struct Person {
    var name = "Taylor Swift"
    var age = 26

    func render() {
        // drawing code here
    }
}
```

In the ideal world, models communicate only with controllers.

Views

A component is considered a View if it is responsible for rendering your user interface, so

that's buttons, images, table views, and more. UIKit labels some of these with the suffix "View", but it's not consistent: **`UIView`**, **`UITableView`**, **`UITextView`** are all good, but **`UIButton`**, **`UISwitch`**, and **`UILabel`** all skip the "View" moniker despite all being views.

Views are designed to be dumb renderers of content, which is why button taps and text view events are usually handled by a controller. But in practice you'll often find views do far more – I've seen networking code, input validation, and even connections straight to model data stored inside views. In the ideal world, views communicate only with controllers.

Controllers

Because models are mostly static data and views are mostly dumb renderers, it falls to controllers to hold everything together in a coherent application. This puts a lot of pressure on one component in the system, so it's common to have apps where the overwhelming majority of code lies with controllers.

On iOS, controllers and views get partly blended into one super-sized component called view controllers, which are responsible for large amounts of system events above and beyond whatever code you might want to write yourself.

To give you an idea of just how big view controllers are, consider this: they handle all view lifecycle events (**`viewDidLoad()`**, **`viewWillAppear()`**, **`viewDidAppear()`**, **`viewWillDisappear()`**, **`viewDidDisappear()`**, **`viewWillLayoutSubviews()`**), memory warnings (**`didReceiveMemoryWarning()`**), moving to and from other view controllers (**`performSegue(withIdentifier:)`**, **`prepare(for segue:)`**, **`present()`**, **`dismiss(animated:)`**, **`modalTransitionStyle`**), layout adjustments (**`edgesForExtendedLayout`**, **`automaticallyAdjustsScrollViewInsets`**, **`preferredStatusBarStyle`**, **`supportedInterfaceOrientations`**), child view controllers (**`addChildViewController`**), restoration (**`restorationIdentifier`**), and even keyboard shortcuts (**`addKeyCommand()`**) – and that's before you write any of your own code!

So, it's no surprise that some people believe MVC stands for "Massive View Controller" – a topic I'll get onto soon.

Unhelpfully, some developers get carried away with adding code to their controllers that could really be in their models or views.

Controllers are designed to pull data from models and render it into views, then read user input on their views and save changes back to the model. As a result, they need access to the views and the models.

Fat controllers

Just looking at the long list of methods you can override on view controllers, as well as remembering they are there to bring models and views together, it's no wonder that many view controllers end up becoming Fat Controllers: over-sized, disorganized, untestable mishmashes of layout, logic, and lifecycle.

I don't claim to be innocent of this crime, and have written my own share of Fat Controllers in the past, but that's OK. After all, the first step to solving a problem is admitting it exists!

(If you were wondering, the name "Fat Controller" comes from the Thomas the Tank Engine books, although I believe the character's official name was "Sir Topham Hatt".)

If you think you're suffering from Fat Controller Syndrome, here are my top tips for helping you refactor your code into something more sensible. Trust me, it's worth it: your code will be easier to understand, easier to maintain, and easier to test.

Create delegate objects

Any non-trivial app will need to work with a range of protocols: `UITableViewDelegate` and `UITableViewDataSource` are the most common, but `UICollectionViewDataSource`, `UIScrollViewDelegate`, `UIImagePickerControllerDelegate` and more are also popular. These all have vital parts to play in app development, but all too often they get thrust into view controllers so that they all get handled simultaneously. A telltale sign that you suffer from this problem is if you write `someObject.delegate = self` entirely with muscle memory: why should `self` be the delegate for pretty much everything?

It doesn't need to be this way. Your table view data source - the bit that must provides `numberOfRowsInSection` and so on – doesn't need to be inside your view controller. Instead, you can create a custom class that provides these methods based on a data source you provide. Your controller holds a reference to the data source object, then assigns it to be the delegate for your table view. This can easily move a hundred or more lines out of your view controller and into a separate class – a class that you can now write tests for, because it has decoupled your business logic from your presentation logic.

Push rendering to your views

If you set up lots of Auto Layout rules in `viewDidLoad()`, if you do custom work in `viewDidLayoutSubviews()`, if your `cellForRowAt` method creates custom cells, then you can spin all that off into custom `UIView` subclasses to handle specific pieces of work. I'm not suggesting you push it all into storyboards because I realize some people have a pathological hatred of Interface Builder, but you can still do your UI work in code without storing it all in your controller.

Push logic and formatting into your model

If your controller has lots of code to format data from your model ready for display, you could easily push that into your model where it makes more sense. If your model has specific data requirements, validation methods for those requirements can be right there in your model rather than in your controller. Users of Core Data will be familiar with this approach, and it can help encapsulate your validation when done judiciously.

For the sake of balance, I should add that this one is really easy to screw up: any logic you add to your model should be ruthlessly focused on the data inside that model. Try to avoid adding code that produces effects – your model ought to be inert if possible.

Switch to MVVM

This is the most drastic of all changes here, and it's a chance you need to weigh up carefully because MVVM doesn't benefit everyone. That being said, if you answer "none" to the question "how many tests have you written for your view controllers?" then I think it's time for radical action. MVVM is covered in more detail in the next chapter, but the TL;DR version is

this: MVVM keeps your model and views slim by pushing validation and formatting out from your view controller into a new object, the View Model.

Is MVC dying?

No. Really, no. It's true that MVVM is picking up traction, but that's just a specialized form of MVC and many would argue that it really needs something like ReactiveCocoa to live up to its full potential. If you use my tips above to their fullest, you should be able to tame your fat controllers and in doing so give MVC a fresh lease of life for your app. Trust me on this: for as long as Apple recommends using MVC – which they still do – it will continue to be the standard.

MVVM

No one denies that MVC – Model, View, Controller – is the standard approach for developing apps using Swift, but that doesn't mean it's perfect. Far from it: MVC stands for "Massive View Controller" if you ask some people, largely because you end up dumping large amounts of code into your view controllers that really ought to go somewhere else. But where?

MVVM – Model, View, ViewModel – is an alternative approach to MVC, but it's actually quite similar. In fact, the best way to think about MVVM is like this: it's just like MVC except you create a new class, the ViewModel, that is responsible for converting data from your model into formatted values for your views.

MVVM was designed by Microsoft for a platform that has data bindings in its user interface, which means the UI can talk directly to the ViewModel to display data and render updates as they come in. OS X has similar functionality, but iOS does not, so MVVM is an imperfect fit in its purest sense. Instead, we have something like "Model, View, View Controller, ViewModel", which is made even more confusing by the fact that iOS blurs the lines between View and View Controller. The end result is that view controllers still exist in MVVM for Swift developers, but in a much simplified form.

If you're not sure whether this chapter is worth reading, try a little experiment: pick one of your most important view controllers, and see how many protocols it conforms to. If the answer is "more than two," chances are you're the victim of Fat Controllers and are likely to benefit from MVVM. What you're looking for is orthogonal code, which is when one piece of code does two vastly different things. For example, if one controller handles reading and writing from the database as well as manipulating views, those two things could and perhaps even *should* be pulled apart.

Note: when I've taught MVVM previously, I sometimes see two extreme reactions. First, "this just changes a view controller into a view model, where's the point?" and "this is going to solve all my fat controller problems." Neither of those is true: MVVM is one facet of a larger strategy to tame view controllers, specifically handling presentation logic. You should still consider creating other data types to handle things like delegates and data sources.

Once you understand MVVM you'll see that it's really just an enhanced form of MVC, where presentation logic is decomposed into a new object. That's it – it's not too complicated. Yes,

there is some complexity around data bindings, but we'll come onto that soon.

View models

Models and views in MVVM are identical to those in MVC, so aren't worth covering in detail. Instead, let's focus on the two aspects that people find confusing about MVVM: what are view models, and where do view controllers fit in?

First, view models. This is a wholly new role in your application, and is responsible for reading data from your model and translating it into a form that can be used by your views. For example, your model might store a date in `NSDate` format, but your user interface wants to receive that in a formatted string, so it's the job of the view model to translate that date into a string. Similarly, you might allow users to change the date for an event, so you might add a method to your view model that accepts a string date, converts it to an `NSDate`, then updates the model.

This is all code that traditionally lives in the view controller, but view controllers are notoriously hard to test. Once you transfer this logic into view models, there's no UI attached to it: there is just data in and data out, so the unit tests almost write themselves. This approach also limits who can access your model, because in MVVM only the view model can read and write to the model – the views talk only to the view model, and the view controllers (still a necessity!) must also use the view model.

Having all your data manipulation and formatting inside a view model is a big win for encapsulation and testing, but it's also helpful for flexibility: if you want to change the way your UI looks across devices, or across languages, or across user settings, you can have multiple view models that draw on the same model. As long as the API is the same – i.e., the methods and properties you surface to consumers of your view model – no other code needs to change.

Second, view controllers. MVVM was built for .NET, which has built-in data bindings for its user interface. That means you can tie the contents of a label into a specific data value, and when the data value changes the label updates. This has no equivalent on iOS, but we do use similar things: delegation, key-value observing (KVO) and even `NSNotificationCenter` all attempt to solve the problem, albeit each with their own drawbacks. It is no coincidence that

the biggest fans of MVVM also happen to use ReactiveCocoa, but that's a separate book entirely!

When you switch to MVVM, VCs are effectively left with the job of displaying the user interface and responding to user interaction. Animations, event handlers, and outlets from Interface Builder are all sensible things for view controllers.

The role of view controllers

View controllers are a mish-mash of two things – the clue is right there in the name! – but more often than not they become a dumping ground for all sorts of code in your apps. When you switch to MVVM, you're carving a huge chunk out of view controller, and moving it into the view model. This does inevitably mean a slight increase in total amount of code, but also means your project's structure is simpler.

View controllers still have an important part to play in MVVM, at least on iOS. However, they become much more true to their name: they are there to control views, and nothing more. That means they interact with UIKit: pushing and popping things with a navigation controller, responding to button presses, showing alerts, and more, should form the vast majority of what your view controllers do. The remainder is setting up your `@IBOutlet`s and mapping data from the model view to user interface components.

What this means is that all business logic – the hard, "what state is my data in?" work – must *not* be stored in your view controllers. Instead, your view controller sends and receives data from the view model, and updates its UI appropriately. You have to admit, the promise of being able to test all your business logic without fighting with view controllers is appealing!

Benefits of MVVM

The biggest benefit of MVVM is that it decomposes your user interface into two parts: preparing your model data for presentation, and updating your UI with that data. Even with the latest, greatest XCTest tools to hand, UI testing can be a really unpleasant business. By taking all the presentation logic out of the view controller, MVVM dramatically increases testability: you can send it example model data and ensure you get the correct transformed data back, ready to be placed into a label. And because view controllers end up being fairly slim, the view

controller testing you need to do has become substantially less.

Separating layout and logic also carries with it increased flexibility for user interface design. With a traditional MVC approach, the layout of your data – e.g. updating a label – and the presentation logic for that data – e.g. making it an uppercase string that combines a date and subtitle from a model object – is so intertwined that it becomes non-trivial to redesign your user interface. Lengthy `cellForRowAt` methods are usually a code smell.

As I said already, you can also create more than one view model for each model depending on your needs. For example, you could create a "simple" view model that shows only the most important highlights, and a "detailed" view model that adds more data for users to read – as far as your model, view, and view controller is concerned, nothing needs to change.

The combination of the above makes MVVM a valuable – if somewhat invasive – approach to reducing the complexity of fat controllers. The nature of MVVM as a more refined form of MVC makes it something you can migrate to slowly: it's completely compatible with MVC, so you can start by converting one controller at a time, rather than diving into a month of refactoring.

Disadvantages of MVVM

Treating MVVM like a silver bullet for fat controller syndrome is likely to cause you to repeat your mistakes: moving 90% of the complexity from your view controllers into a view model doesn't help anyone. Instead, I would argue that MVVM gives you a template for decomposing presentation logic that you can then re-use when it comes to networking logic, persistence logic, theming logic, and more. If you treat your view model as a dumping ground for miscellaneous code that doesn't fit anywhere else, you haven't gained a great deal.

Another disadvantage of MVVM is that it forces you to write more code. This is unavoidably true, and a side effect of creating an extra layer of abstraction. To be fair, the more code you have in your project, the less the code increase caused by MVVM will matter – to the point where it's effectively a rounding error. In the fact of dramatically improved testability and a clearer separation of concerns, code size ought not to be an issue. For simpler projects, though, it's a different story: the creator of MVVM said "for simple UI, MVVM can be overkill," so I would suggest weighing up the benefits as you go.

On a more personal note, I think MVVM users can get themselves into a mess when they start to add behavior to their view models. I already said that pushing complexity from a view controller to a view model won't gain you much, but this is a step further: as soon as you start making your view models do more than transform data from your model, you're just adding complexity where it doesn't belong. Taking another quote from the creator of MVVM, "the view model, though it sounds View-ish is really more Model-ish" – treat your view model as a transformer of data, rather than something that acts upon that data, and you'll be fine.

MVC vs MVVM

I'm putting this chapter title here not because it's important, but because it's what some people will expect to see. If you're thinking to yourself, "this was too long, just tell me which I should use," then I'm afraid I have some bad news: MVC vs MVVM doesn't really exist, at least not to me. MVVM is a natural evolution of MVC: it's the kind of MVC you should already have been writing, rather than a replacement. It forces you to decompose orthogonal code into individual objects, which is smart behavior regardless of what you call it.

But MVVM is *not* the end of the process. Instead, it's the beginning: I recommend everyone gives MVVM a try at least once, but once you've mastered the approach you'll realize that the next logical step is to decompose your view model further so that more pieces of functionality are isolated. This is *not* me arguing for ravioli code, but simple separation of concerns – if your view model handles network requests, I'd be very suspicious indeed.

Note: spaghetti code is the name given to confused source code where various parts are hopelessly entangled; ravioli code is the name given to producing lots of small objects that contain their own data and functionality independent from other objects. Both terms are frequently used negatively: if someone says your code is ravioli code, they usually mean you've split up your program into so many small parts that it's hard to understand how it fits together.

Command-line Swift

If you're targeting a platform that sports a terminal, such as OS X or Linux, you can leverage your Swift knowledge to build powerful command-line apps with only a small amount of learning.

To demonstrate this, we're going to build a simple command-line Hangman game that reads user input and writes the current state of play. Once you get the hang of it, you'll be able to write your own quick Swift programs that can be chained to other terminal commands just like anything else – it's quite liberating!

Note: if you're using Linux and don't already have Swift installed, please start by following Apple's official guidelines: https://swift.org/getting-started/.

Before we dive into Hangman, I said you need to learn some things first, so let's start with reading user input. This is done using the readLine() function, which returns an optional string: either the user entered something, or input ended. That "something" could be an empty string if the user hit return without typing anything – you will only get back nil if there is no more input. This is usually triggered by pressing Ctrl+D on the command line.

So, to read user input you write code like this:

```
if let input = readLine() {
    print("You typed \(input)\n")
}

print("Done!\n")
```

Because Hangman will keep requesting user input until they have either guessed the word or quit, we'll put that into a loop like this:

```
while var input = readLine() {
    // do stuff
}
```

Next, Swift's support on Linux is a bit shaky in places. Yes, it's evolving quickly and I'm sure

we'll get to basic parity sooner or later, but you'll find many things not implemented. Broadly speaking, all the Swift standard library should work perfectly on OS X and Linux, but if you're using things from Foundation – e.g. the **`contentsOfFile:usedEncoding`** initializer for strings – you are likely to come across warnings about missing features.

Third, your code is parsed from top to bottom sort of like playgrounds. This means using a function before it has been declared will trigger an error. You can use classes, structs, enums, and so on if you want to, but to begin with you'll probably treat your command-line Swift like a compiled playground.

Fourth, the **`exit()`** function is available to you as a way to force your app to terminate. This is generally considered A Bad Thing in iOS, but in command-line apps it makes sense because you want to halt execution when you're done.

Finally, although Xcode has a built-in template for OS X terminal apps, there really isn't any need for it. Yes, go ahead and edit your Swift file in Xcode so you can benefit from code completion and syntax highlighting, but there are three much nicer ways to compile and test:

1. The **`swiftc`** command converts a Swift file into an executable.
2. The **`swiftc -O`** command does the same, but adds optimization – you'll want to use this for your finished product.
3. The **`swift`** command converts a Swift file into a temporary executable and runs it straight away, as if it were a scripting language. This is great for testing.

OK, it's time to put all that into a practical example. Please create a folder on your desktop called "hangman", then create a new Swift file in there called hangman.swift. You can open it for editing in Xcode if you want, or in something like Vim or Emacs if you're using Linux.

Give the file this initial code:

```
import Foundation

let word = "RHYTHM"
var usedLetters = [Character]()
```

```
print("Welcome to Hangman!")
print("Press a letter to guess, or Ctrl+D to quit.")
```

That imports Foundation, sets up some data we'll be using for the game, and prints out a welcome message. The word is hard-coded to "RHYTHM", but you're welcome to be more creative. The **`usedLetters`** variable will track the letters the player has guessed – we'll let them make eight guesses before they lose.

To test that everything is working OK, please open a terminal window and change to that folder. If you're on OS X and new to using the command line, here are some basic instructions:

1. Press Cmd+Space to show Spotlight.
2. Type "terminal" then press return to launch the Terminal app.
3. Type "cd Desktop/hangman" then return to change to the "hangman" folder on your desktop.

Once you have a terminal window open at your hangman folder, run "swift hangman.swift" to build and run your code. After a moment of thinking, you should see the following message appear on your screen:

```
Welcome to Hangman!
Press a letter to guess, or Ctrl+D to quit.
```

You'll be returned to the command prompt once that message has been shown, because no user input is requested just yet. However, at least we know your Swift compiler is set up correctly!

The next thing we're going to do is define a function called **`printWord()`** that will print the word the player needs to guess. To do that, it will loop through every character in the **`word`** constant and either print the character or print an underscore depending on whether the user has guessed that letter already. This will be done using a plain old **`print()`** call, but we'll using its **`terminator`** parameter so that it doesn't add line breaks unless we need one.

Here's the code for the **`printWord()`** function; please add it to the end of the hangman.swift file:

```
func printWord() {
    print("\nWord: ", terminator: "")
    var missingLetters = false

    for letter in word.characters {
        if usedLetters.contains(letter) {
            print(letter, terminator: "")
        } else {
            print("_", terminator: "")
            missingLetters = true
        }
    }

    print("\nGuesses: \(usedLetters.count)/8")

    if missingLetters == false {
        print("It looks like you live on… for now.")
        exit(0)
    } else {
        if usedLetters.count == 8 {
            print("Oops – you died! The word was \(word).")
            exit(0)
        } else {
            print("Enter a guess: ", terminator: "")
        }
    }
}
```

Make sure you look closely at the way I've used **`print()`**: sometimes I use the **`terminator`** parameter and sometimes I don't. This is used so that printing each character from the word is done on a single line even though there are multiple **`print()`** calls, and also on the last line where users are asked to enter a guess - this means their guess will be typed on the same line as the prompt.

We're almost finished with the game already, but there are three more things to do: print out the starting game status, read the user's input until they either win or lose, and print a message if they quit using Ctrl+D. Each time we read the user's input we need to check whether they have used that letter before, then print out the newest game status.

So, the game starts by calling **`printWord()`** once to show the user the basic instructions. Add this to the bottom of the file:

```
printWord()
```

Now we need to call **`readLine()`** repeatedly, add the new character to the **`usedLetters`** array if it hasn't been used before, then call **`printWord()`** again. Add this below the previous line:

```
while var input = readLine() {
    if let letter = input.uppercased().characters.first {
        if usedLetters.contains(letter) {
            print("You used that letter already!")
        } else {
            usedLetters.append(letter)
        }
    }

    printWord()
}
```

Note that I use **`if let`** with **`characters.first`** to safely unwrap the first letter in the user's input. This means that if the user presses return without typing anything they'll just see the game status printed again. It also means that if they try to write two or more characters, we only ever use the first.

Finally, we need to print a goodbye message, like this:

```
print("Thanks for playing!")
```

We're using `exit()` to terminate the program at the right point, so you'll need to place that `print()` call before `exit()` to ensure it gets printed.

That's it! Go ahead and run `swift hangman.swift` to try the game for yourself. If you're using the Swift package manager, run the command `touch Package.swift` (with a capital P), then use `swift build` to build the binary.