CHAPTER 5
Flow Control and More

This chapter is a miscellany. I'll start by describing Swift's flow control constructs for branching, looping, and jumping. Then, I'll summarize Swift's privacy and introspection features, and talk about how to override operators and how to create your own operators. Next, I'll explain some specialized aspects of Swift memory management. Finally, I'll survey some relatively recent Swift language features: synthesized protocol implementations, key paths, instance as function, dynamic members, property wrappers, custom string interpolation, reverse generics, result builders, and Result.

Flow Control

A computer program has a *path of execution* through its code statements. Normally, this path follows a simple rule: execute each statement in succession. But there is another possibility. *Flow control* can be used to make the path of execution skip some statements, or go back and repeat some statements.

Flow control is what makes a computer program "intelligent." By testing in real time the truth value of a *condition* — an expression that evaluates to a Bool and is thus `true` or `false` — the program decides *at that moment* how to proceed. Flow control based on testing a condition may be divided into two general types:

Branching
: The code is divided into alternative chunks, like roads that diverge in a wood, and the program is presented with a choice of possible ways to go; the truth of a condition is used to determine which chunk will actually be executed.

Looping
: A chunk of code is marked off for possible repetition; the truth of a condition is used to determine whether the chunk should be executed, and then whether it should be executed again. Each repetition is called an *iteration*.

The chunks of code in flow control, which I refer to as *blocks*, are demarcated by curly braces. These curly braces constitute a scope (Chapter 1). New local variables can be declared here, and go out of existence automatically when the path of execution exits the curly braces (Chapter 3). For a loop, this means that local variables come into existence and go out of existence on each iteration. As with any scope, code inside the curly braces can see the surrounding higher scope.

Swift flow control is fairly simple, and by and large is similar to flow control in C and related languages. There are two fundamental syntactic differences between Swift and C, both of which make Swift simpler and clearer:

- A condition *does not have to be wrapped in parentheses* in Swift.
- The curly braces *can never be omitted* in Swift.

Moreover, Swift adds some specialized flow control features to help you grapple more conveniently with Optionals, and boasts a particularly powerful form of switch statement.

Branching

Swift has two forms of branching: the if construct, and the switch statement. I'll also discuss conditional evaluation, a compact form of if construct.

If construct

The Swift branching construct with `if` is similar to C. Many examples of if constructs have appeared already in this book. The construct may be formally summarized as shown in Example 5-1.

Example 5-1. The Swift if construct

```
if condition {
    statements
}

if condition {
    statements
} else {
    statements
}

if condition {
    statements
} else if condition {
    statements
} else {
    statements
}
```

The third form, containing `else if`, can have as many `else if` blocks as needed, and the final `else` block may be omitted.

Here's a real-life if construct that lies at the heart of one of my apps:

```
// okay, we've tapped a tile; there are three cases
if self.selectedTile == nil { // no selected tile: select and play this tile
    self.select(tile:tile)
    self.play(tile:tile)
} else if self.selectedTile == tile { // selected tile tapped: deselect it
    self.deselectAll()
    self.player?.pause()
} else { // there was a selected tile, another tile was tapped: swap them
    self.swap(self.selectedTile, with:tile, check:true, fence:true)
}
```

Conditional binding

In Swift, `if` can be followed immediately by a variable declaration and assignment — that is, by `let` or `var` and a new local variable name, possibly followed by a colon and a type declaration, then an equal sign and a value:

```
if let variable = value {
    // the block
}
```

This syntax, called a *conditional binding*, is actually a shorthand for *conditionally unwrapping an Optional.* The assigned value (*`value`*) is expected to be an Optional — the compiler will stop you if it isn't — and this is what happens:

- If the Optional (*`value`*) is `nil`, the condition fails and the block is skipped, with execution resuming after the block.
- If the Optional is *not* `nil`, then:
    1. The Optional is unwrapped.
    2. The unwrapped value is assigned to the declared local variable (*`variable`*).
    3. The block is executed with the local variable in scope. The local variable is *not* in scope outside the block.

So a conditional binding safely passes an unwrapped Optional into a block. The Optional is unwrapped, and the block is executed, only if the Optional *can* be unwrapped.

It is perfectly reasonable for the local variable in a conditional binding to have the same name as an existing variable in the surrounding scope. It can even have the same name as the Optional being unwrapped! There is then no need to make up a new name, and inside the block the unwrapped value of the Optional neatly overshadows the original Optional so that the latter can't be accessed accidentally.

Recall this code from Chapter 4, where I optionally unwrap a Notification's `userInfo` dictionary, attempt to fetch a value from the dictionary using the `"progress"` key, and proceed only if that value turns out to be an NSNumber that can be cast down to a Double:

```
let prog = n.userInfo?["progress"] as? Double
if prog != nil {
    self.progress = prog!
}
```

We can rewrite that code more elegantly and compactly as a conditional binding:

```
if let prog = n.userInfo?["progress"] as? Double {
    self.progress = prog
}
```

It is also possible to nest conditional bindings. To illustrate, I'll rewrite the previous example to use a separate conditional binding for each Optional in the chain:

```
if let ui = n.userInfo {
    if let prog = ui["progress"] as? Double {
        self.progress = prog
    }
}
```

The result, if the chain involves many optional unwrappings, can be somewhat verbose and the nest can become deeply indented; Swift programmers like to call this the "pyramid of doom." To help avoid the indentation, nested conditions can be expressed as a *condition list*. Imagine that each left curly brace, together with the following `if`, is replaced by a comma:

```
if let ui = n.userInfo, let prog = ui["progress"] as? Double {
    self.progress = prog
}
```

The conditions are evaluated in left-to-right order. Each condition depends upon the preceding ones; a binding introduced in an earlier condition is in scope in a later one, and if a condition fails the entire list fails immediately. In the example code, the assignment to `prog` won't even be attempted if `n.userInfo` is `nil`; the assignment to `ui` fails and that's the end.

Condition lists do not have to consist solely of conditional bindings. They can include ordinary conditions. It would be possible (though not as elegant) to rewrite the previous example like this:

```
if let ui = n.userInfo, let prog = ui["progress"], prog is Double {
    self.progress = prog as! Double
}
```

If the list is too long for a single line, it can be conveniently broken up after each comma:

```
if let ui = n.userInfo,
    let prog = ui["progress"],
    prog is Double {
        self.progress = prog as! Double
}
```

Personally, I am not fond of this kind of extended condition list. I actually prefer the pyramid of doom, where the structure reflects perfectly the successive stages of testing. If I want to avoid the pyramid of doom, I can usually use a sequence of `guard` statements ("Guard" on page 303):

```
guard let ui = n.userInfo else {return}
guard let prog = ui["progress"] as? Double else {return}
self.progress = prog
```

Switch statement

A switch statement is a neater way of writing an extended `if...else if...else` construct. In C (and Objective-C), a switch statement contains hidden traps; Swift eliminates those traps, and adds power and flexibility. As a result, switch statements are commonly used in Swift (whereas they are rare in my Objective-C code).

In a switch statement, the condition involves the comparison of different possible values, called *cases*, against a single value, called the *tag*. The case comparisons are performed *successively in order*. As soon as a case comparison succeeds, that case's code is executed and the entire switch statement is exited. The schema is shown in Example 5-2; there can be as many cases as needed, and the `default` case can be omitted (subject to restrictions that I'll explain in a moment).

Example 5-2. The Swift switch statement

```
switch tag {
case pattern1:
    statements
case pattern2:
    statements
default:
    statements
}
```

Here's an actual example:

```
switch i {
case 1:
    print("You have 1 thingy!")
case 2:
    print("You have 2 thingies!")
default:
    print("You have \(i) thingies!")
}
```

In that code, an Int variable `i` functions as the tag. The value of `i` is first compared to the value `1`. If it *is* `1`, that case's code is executed and that's all. If it is *not* `1`, it is compared to the value `2`. If it *is* `2`, *that* case's code is executed and that's all. If the value of `i` matches neither of those, the `default` case's code is executed.

In Swift, a switch statement must be *exhaustive*. This means that *every* possible value of the tag must be covered by a case. The compiler will stop you if you try to violate this rule. If you don't want to write every case explicitly, you must add a "mop-up" case that covers all other cases; a common way to do that is to add a `default` case. It's easy to write an exhaustive switch when the tag is an enum with a small number of cases, but when the tag is an Int, there is an infinite number of possible cases, so a "mop-up" case *must* appear.

Each case's code can consist of multiple statements; it doesn't have to be a single statement, like the cases in the preceding example. However, it must consist of *at least* a single statement; it is illegal for a Swift switch case to be empty. It is legal for the first (or only) statement of a case's code to appear on the same line as the case, after the colon; I could have written the preceding example like this:

```
switch i {
case 1: print("You have 1 thingy!")
case 2: print("You have 2 thingies!")
default: print("You have \(i) thingies!")
}
```

The minimum single statement of case code is the keyword `break`; used in this way, `break` acts as a placeholder meaning, "Do nothing." It is very common for a switch statement to include a `default` (or other "mop-up" case) consisting of nothing but the keyword `break`; in this way, you exhaust all possible values of the tag, but if the value is one that no case explicitly covers, you do nothing.

Now let's focus on the comparison between the tag value and the case value. In the preceding example, it works like an equality comparison (==); but that isn't the only possibility. In Swift, a case value is actually a special expression called a *pattern*, and the pattern is compared to the tag value using a "secret" pattern-matching operator, ~=. The more you know about the syntax for constructing a pattern, the more powerful your case values and your switch statements will be.

A pattern can include an underscore (_) to absorb all values without using them. An underscore case is thus an alternative form of "mop-up" case:

```
switch i {
case 1:
    print("You have 1 thingy!")
case _:
    print("You have many thingies!")
}
```

A pattern can include a declaration of a local variable name (an unconditional binding) to absorb all values and use the actual value. This is yet another alternative form of “mop-up” case:

```
switch i {
case 1:
    print("You have 1 thingy!")
case let n:
    print("You have \(n) thingies!")
}
```

When the tag is a Comparable, a case can include a Range; the test involves sending the Range the `contains` message:

```
switch i {
case 1:
    print("You have 1 thingy!")
case 2...10:
    print("You have \(i) thingies!")
default:
    print("You have more thingies than I can count!")
}
```

A Range pattern can be an opportunity to use partial range syntax:

```
switch i {
case ..<0:
    print("i is negative, namely \(i)")
case 1...:
    print("i is positive, namely \(i)")
case 0:
    print("i is 0")
default:break
}
```

When the tag is an Optional, a case can test it against `nil`. Moreover, appending `?` to a case pattern safely unwraps an Optional tag. Presume that `i` is an Optional wrapping an Int:

```
switch i {
case 1?:
    print("You have 1 thingy!")
case let n?:
    print("You have \(n) thingies!")
case nil: break
}
```

When the tag is a Bool, a case can test it against a condition. Thus, by a clever perversion, you can use the cases to test *any* conditions you like by using `true` as the tag; a switch statement becomes a genuine substitute for an extended `if...else if` construct. In this example from my own code, I could have used `if...else if`, but a switch statement seems cleaner:

```
func position(for bar: UIBarPositioning) -> UIBarPosition {
    switch true {
    case bar === self.navbar:  return .topAttached
    case bar === self.toolbar: return .bottom
    default:                   return .any
    }
}
```

A pattern can include a where clause adding a condition to limit the truth value of the case. This is often, though not necessarily, used in combination with a binding; the condition can refer to the variable declared in the binding:

```
switch i {
case let j where j < 0:
    print("i is negative, namely \(j)")
case let j where j > 0:
    print("i is positive, namely \(j)")
case 0:
    print("i is 0")
default:break
}
```

A pattern can include the `is` operator to test the tag's type. In this example, we have a Dog class and its NoisyDog subclass, and `d` is typed as Dog:

```
switch d {
case is NoisyDog:
    print("You have a noisy dog!")
case _:
    print("You have a dog")
}
```

A pattern can include a cast with the `as` (not `as?`) operator. Typically, you'll combine this with a binding that declares a local variable; despite the use of unconditional `as`, the cast is conditional, and if it succeeds, the local variable carries the cast value into the case code. Again, `d` is typed as Dog, which has a NoisyDog subclass; assume that Dog implements `bark` and that NoisyDog implements `beQuiet`:

```
switch d {
case let nd as NoisyDog:
    nd.beQuiet()
case let d:
    d.bark()
}
```

You can also use `as` (not `as?`) to cast down the tag (and possibly unwrap it) conditionally as part of a test against a specific match. In this example, `i` might be an Any or an Optional wrapping an Any:

```
switch i {
case 0 as Int:
    print("It is 0")
default:break
}
```

You can perform multiple tests at once by expressing the tag as a tuple and wrapping the corresponding tests in a tuple. The case passes only if every test in the case tuple succeeds against the corresponding member of the tag tuple. In this example, we start with a dictionary `d` typed as `[String:Any]`. Using a tuple, we can safely attempt to fetch and cast two values at once:

```
switch (d["size"], d["desc"]) {
case let (size as Int, desc as String):
    print("You have size \(size) and it is \(desc)")
default:break
}
```

When a tag is an enum, the cases can be cases of the enum. A switch statement is thus an excellent way to handle an enum. Here's the Filter enum from Chapter 4:

```
enum Filter {
    case albums
    case playlists
    case podcasts
    case books
}
```

And here's a switch statement, where the tag, `type`, is a Filter; no mop-up is needed, because I've exhausted the cases:

```
switch type {
case .albums:
    print("Albums")
case .playlists:
    print("Playlists")
case .podcasts:
    print("Podcasts")
case .books:
    print("Books")
}
```

If an enum comes from Objective-C (or C) or the Swift standard library, an exhaustive switch over it might get you a warning from the compiler that the enum "may have additional unknown values." I'll explain what that means, and what to do about it, in the Appendix.

A switch statement provides a way to extract an associated value from an enum case. Recall this enum from Chapter 4:

```
enum MyError {
    case number(Int)
    case message(String)
    case fatal
}
```

If a case of the enum has an associated value, a tuple of patterns after the matched case name is applied to the associated value. If a pattern is a binding variable, it captures the associated value. The `let` (or `var`) can appear inside the parentheses or after the `case` keyword; this code illustrates both alternatives:

```
switch err {
case .number(let theNumber):
    print("It is a number: \(theNumber)")
case let .message(theMessage):
    print("It is a message: \(theMessage)")
case .fatal:
    print("It is fatal")
}
```

If the `let` (or `var`) appears after the `case` keyword, I can add a where clause:

```
switch err {
case let .number(n) where n > 0:
    print("It's a positive error number \(n)")
case let .number(n) where n < 0:
    print("It's a negative error number \(n)")
case .number(0):
    print("It's a zero error number")
default:break
}
```

If I don't want to extract the error number but just want to match against it, I can use some other pattern inside the parentheses:

```
switch err {
case .number(1...):
    print("It's a positive error number")
case .number(..<0):
    print("It's a negative error number")
case .number(0):
    print("It's a zero error number")
default:break
}
```

This same pattern also gives us yet another way to deal with an Optional tag. An Optional, as I explained in Chapter 4, is in fact an enum. It has two cases, `.none` and `.some`, where the wrapped value is the `.some` case's associated value. But now we know how to extract the associated value! We can rewrite yet again the earlier example where `i` is an Optional wrapping an Int:

```
switch i {
case .none: break
case .some(1):
    print("You have 1 thingy!")
case .some(let n):
    print("You have \(n) thingies!")
}
```

To combine switch case tests (with an implicit logical-or), separate them with a comma:

```
switch i {
case 1,3,5,7,9:
    print("You have a small odd number of thingies")
case 2,4,6,8,10:
    print("You have a small even number of thingies")
default:
    print("You have too many thingies for me to count")
}
```

In this example, `i` is declared as an Any:

```
switch i {
case is Int, is Double:
    print("It's some kind of number")
default:
    print("I don't know what it is")
}
```

A comma can even combine patterns that declare binding variables, provided they declare the same variable of the same type (`err` is our MyError once again):

```
switch err {
case let .number(n) where n > 0, let .number(n) where n < 0:
    print("It's a nonzero error number \(n)")
case .number(0):
    print("It's a zero error number")
default:break
}
```

Another way of combining cases is to jump from one case to the next by using a `fallthrough` statement. When a `fallthrough` statement is encountered, the current case code is aborted immediately and the next case code runs *unconditionally*. The test of the next case is not performed, so the next case can't declare any binding variables, because they would never be set. It is not uncommon for a case to consist entirely of a `fallthrough` statement:

```
switch pep {
case "Manny": fallthrough
case "Moe": fallthrough
case "Jack":
    print("\(pep) is a Pep boy")
default:
    print("I don't know who \(pep) is")
}
```

If case

When all you want to do is extract an associated value from one enum case, a full switch statement may seem a bit heavy-handed. The lightweight `if case` construct lets you use in a condition the same sort of pattern syntax you'd use in a case of a switch statement. The structural difference is that, whereas a switch case pattern is compared against a previously stated tag, an `if case` pattern is followed by an equal sign and then the tag. In this code, `err` is our MyError enum once again:

```
if case let .number(n) = err {
    print("The error number is \(n)")
}
```

The condition starting with `case` can be part of a longer comma-separated condition list:

```
if case let .number(n) = err, n < 0 {
    print("The negative error number is \(n)")
}
```

Conditional evaluation

An interesting problem arises when you'd like to decide on the fly what value to use — for example, what value to assign to a variable. This seems like a good use of a branching construct. You can, of course, declare the variable first without initializing it, and then set it from within a subsequent branching construct. It would be nice, however, to use a branching construct *as* the variable's value. Here, I try (and fail) to write a variable assignment where the equal sign is followed directly by a branching construct:

```
let title = switch type { // compile error
case .albums:
    "Albums"
case .playlists:
    "Playlists"
case .podcasts:
    "Podcasts"
case .books:
    "Books"
}
```

There are languages that let you talk that way, but Swift is not one of them. However, an easy workaround does exist — use a define-and-call anonymous function, as I suggested in Chapter 2:

```
let title : String = {
    switch type {
    case .albums:
        return "Albums"
    case .playlists:
        return "Playlists"
    case .podcasts:
        return "Podcasts"
    case .books:
        return "Books"
    }
}()
```

In the special case where a value can be decided by a two-pronged condition, Swift provides the C ternary operator (`?:`). Its scheme is:

```
condition ? exp1 : exp2
```

If the condition is `true`, the expression *`exp1`* is evaluated and the result is used; otherwise, the expression *`exp2`* is evaluated and the result is used. You can use the ternary operator while performing an assignment, using this schema:

```
let myVariable = condition ? exp1 : exp2
```

What `myVariable` gets initialized to depends on the truth value of the condition.

I use the ternary operator heavily in my own code. Here's an example:

```
cell.accessoryType =
    ix.row == self.currow ? .checkmark : .disclosureIndicator
```

The context needn't be an assignment; here, we're deciding what value to pass as a function argument:

```
context.setFillColor(self.hilite ? purple.cgColor : beige.cgColor)
```

The ternary operator can also be used to determine the receiver of a message. In this example, one of two UIViews will have its background color set:

```
(self.firstRed ? v1 : v2).backgroundColor = .red
```

In Objective-C, there's a collapsed form of the ternary operator that allows you to test a value against `nil`. If it is `nil`, you get to supply a substitute value. If it *isn't* `nil`, the tested value itself is used. In Swift, the analogous operation would involve testing an Optional: if the tested Optional is `nil`, use the substitute value; if it *isn't* `nil`, *unwrap* the Optional and use the unwrapped value. Swift has such an operator — the `??` operator (called the *nil-coalescing* operator).

Here's a real-life example from my own code:

```
func tableView(_ tv: UITableView, numberOfRowsInSection sec: Int) -> Int {
    return self.titles?.count ?? 0
}
```

In that example, self.titles is of type [String]?. If it's not nil, I want to unwrap the array and return its count. But if it *is* nil, there is no data and no table to display — but I *must* return *some* number, so clearly I want to return zero. The nil-coalescing operator lets me express all that very neatly.

The nil-coalescing operator together with the Optional map(_:) method neatly solves a class of problem where your goal is to *process* the wrapped value of an Optional or, if it is nil, to assign some default value. Suppose our goal is to produce a string expressing the index of target within arr if it is present, or "NOT FOUND" if it is not. This works, but it's ugly:

```
let arr = ["Manny", "Moe", "Jack"]
let target = // some string
let pos = arr.firstIndex(of:target)
let s = pos != nil ? String(pos!) : "NOT FOUND"
```

Here's a more elegant way:

```
let arr = ["Manny", "Moe", "Jack"]
let target = // some string
let s = arr.firstIndex(of:target).map {String($0)} ?? "NOT FOUND"
```

Expressions using ?? can be chained:

```
let someNumber = i1 as? Int ?? i2 as? Int ?? 0
```

That code tries to cast i1 to an Int and use that Int. If that fails, it tries to cast i2 to an Int and use *that* Int. If *that* fails, it gives up and uses 0.

Loops

The usual purpose of a loop is to repeat a block of code with some simple difference on each iteration. This difference will typically serve also as a signal for when to stop the loop. Swift provides two basic loop structures: while loops and for loops.

While loops

A while loop comes in two forms, schematized in Example 5-3.

Example 5-3. The Swift while loop

```
while condition {
    statements
}

repeat {
    statements
} while condition
```

The chief difference between the two forms is the timing of the test. In the second form, the condition is tested after the block has executed — meaning that the block will be executed at least once.

Usually, the code inside the block will change something that alters the environment and hence the value of the condition, eventually bringing the loop to an end. Here's a typical example from my own code (`movenda` is an array):

```
while self.movenda.count > 0 {
    let p = self.movenda.removeLast()
    // ...
}
```

Each iteration removes an element from `movenda`, so eventually its `count`, evaluated in the condition, falls to `0` and the loop is no longer executed; execution then proceeds to the next line after the closing curly braces.

In its first form, a while loop's condition can involve a conditional binding of an Optional. This provides a compact way of safely unwrapping an Optional and looping until the Optional is `nil`; the local variable containing the unwrapped Optional is in scope inside the curly braces. My previous code can be rewritten more compactly:

```
while let p = self.movenda.popLast() {
    // ...
}
```

Here's an example of `repeat...while` from my own code. In my LinkSame app, if there are no legal moves, we pick up all the cards, shuffle them, and redeal them into the same layout. It's possible that when we do that, there will *still* be no legal moves. So we do it again until there *is* a legal move:

```
repeat {
    var deck = self.gatherUpCards()
    deck.shuffle()
    self.redeal(deck)
} while self.legalPath() == nil
```

Similar to the `if case` construct, `while case` lets you use a switch case pattern. In this rather artificial example, we have an array of various MyError enums:

```
let arr : [MyError] = [
    .message("ouch"), .message("yipes"), .number(10), .number(-1), .fatal
]
```

With `while case`, we can extract the `.message` associated string values from the start of the array:

```
var i = 0
while case let .message(message) = arr[i]  {
    print(message) // "ouch", then "yipes"; then the loop stops
    i += 1
}
```

For loops

The Swift for loop is schematized in Example 5-4.

Example 5-4. The Swift for loop

```
for variable in sequence {
    statements
}
```

With a for loop, you cycle through (*enumerate*) a sequence. The sequence must be an instance of a type that adopts the Sequence protocol. An Array is a Sequence. A Dictionary is a Sequence. A Set is a Sequence. A String is a Sequence. A Range is a Sequence (as long as it is a range of something that comes in discrete steps, like Int). Those, and sequences derived from them, are the things to which you will regularly apply `for...in`.

On each iteration, a successive element of the sequence is used to initialize the variable. The variable is local to the block; it is in scope inside the curly braces, and is not visible outside them. The variable is implicitly declared with `let`; it is immutable by default. If you need to assign to or mutate the variable within the block, write `for var`.

A common use of for loops is to iterate through successive numbers. This is easy in Swift, because you can readily create a sequence of numbers on the fly — a Range:

```
for i in 1...5 {
    print(i) // 1, 2, 3, 4, 5
}
```

A for loop is secretly a while loop. Sequence has a `makeIterator` method that yields an iterator object adopting IteratorProtocol. According to this protocol, the iterator has a mutating `next` method that returns the next object in the sequence wrapped in an Optional, or `nil` if there is no next object. Under the hood, `for...in` is repeatedly calling the `next` method in a while loop. The previous code actually works like this:

```
var iterator = (1...5).makeIterator()
while let i = iterator.next() {
    print(i) // 1, 2, 3, 4, 5
}
```

Sometimes you may find that writing out the while loop explicitly in that way makes the loop easier to control and to customize.

When you cycle through a sequence with `for...in`, what you're actually cycling through is a *copy* of the sequence. That means it's safe to mutate the sequence while you're cycling through it:

```
var s : Set = [1,2,3,4,5]
for i in s {
    if i.isMultiple(of:2) {
        s.remove(i)
    }
} // s is now [1,3,5]
```

That may not be the most elegant way to remove all even numbers from the Set `s`, but it's not illegal or dangerous; removing an element of the Set doesn't mess up the subsequent iterations of the loop.

Not only is the *sequence* a copy; if the variable type is a value type, such as a struct, the *variable* is a copy. So even if you mutate the variable (which is legal if you say `for var`), the original sequence's elements are unaffected. Here's a Dog that's a struct:

```
struct Dog {
    var name : String
    init(_ n:String) {
        self.name = n
    }
}
```

We cycle through an array of Dogs, uppercasing their names:

```
var dogs : [Dog] = [Dog("rover"), Dog("fido")]
for var dog in dogs {
    dog.name = dog.name.uppercased()
}
```

But nothing useful happens; `dogs` still consists of Dogs named `"rover"` and `"fido"`. If the Sequence is also a Collection, one common workaround is to cycle through its `indices` instead, so that you can assign back into the original array:

```
var dogs : [Dog] = [Dog("rover"), Dog("fido")]
for ix in dogs.indices {
    dogs[ix].name = dogs[ix].name.uppercased()
}
```

Now `dogs` consists of Dogs named `"ROVER"` and `"FIDO"`.

The sequence `enumerated` method yields a succession of tuples in which each element of the original sequence is preceded by a count (see "Array enumeration and transformation" on page 244). In this example from my real code, `tiles` is an array of UIViews and `centers` is an array of CGPoints saying where those views are to be positioned:

```
for (i,v) in self.tiles.enumerated() {
    v.center = self.centers[i]
}
```

A `for...in` construct can take a where clause, allowing you to skip some values of the sequence:

```
for i in 0...10 where i.isMultiple(of:2) {
    print(i) // 0, 2, 4, 6, 8, 10
}
```

Like `if case` and `while case`, there's also `for case`, permitting a switch case pattern to be used in a for loop. The tag is each successive value of the sequence, so no assignment operator is used. To illustrate, let's start again with an array of MyError enums:

```
let arr : [MyError] = [
    .message("ouch"), .message("yipes"), .number(10), .number(-1), .fatal
]
```

Here we cycle through the whole array, extracting only the `.number` associated values:

```
for case let .number(i) in arr {
    print(i) // 10, -1
}
```

Another common use of `for case` is to cast down conditionally, picking out only those members of the sequence that can be cast down safely. In this real-life code, I want to hide all subviews that happen to be buttons:

```
for case let b as UIButton in self.boardView.subviews {
    b.isHidden = true
}
```

A sequence also has instance methods, such as `map(_:)`, `filter(_:)`, and `reversed`; you can apply these to hone the sequence through which we will cycle. In this example, I count backward by even numbers:

```
let range = (0...10).reversed().filter {$0.isMultiple(of:2)}
for i in range {
    print(i) // 10, 8, 6, 4, 2, 0
}
```

Yet another approach is to generate the sequence by calling global functions such as `stride(from:through:by)` or `stride(from:to:by:)`. These are applicable to adopters of the Strideable protocol, such as numeric types and anything else that can be incremented and decremented. Which form you use depends on whether you want the sequence to include the final value. The `by:` argument can be negative:

```
for i in stride(from: 10, through: 0, by: -2) {
    print(i) // 10, 8, 6, 4, 2, 0
}
```

The Swift Algorithms package supplements that with a `striding(by:)` method on an array. This lets you stride through the array itself rather than striding through index numbers and indexing into the array:

```
let planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter"]
for planet in planets.striding(by:2) {
    print(planet) // Mercury, Earth, Jupiter
}
```

For maximum flexibility, you can use the global `sequence` function to generate your sequence by rule. It takes two parameters — an initial value, and a generation function that returns the next value based on what has gone before. In theory, the sequence generated by the `sequence` function can be infinite in length — though this is not a problem, because the resulting sequence is "lazy," meaning that an element isn't generated until you ask for it. In reality, you'll use one of two techniques to limit the result:

Return `nil`

The generation function can limit the sequence by returning `nil` to signal that the end has been reached:

```
let seq = sequence(first:1) {$0 >= 10 ? nil : $0 + 1}
for i in seq {
    print(i) // 1,2,3,4,5,6,7,8,9,10
}
```

Stop requesting elements

You can request just a piece of the infinite sequence — for example, by cycling through the sequence for a while and then stopping, or by taking a finite `prefix`:

```
let seq = sequence(first:1) {$0 + 1}
for i in seq.prefix(5) {
    print(i) // 1,2,3,4,5
}
```

The `sequence` function comes in two forms:

`sequence(first:next:)`

Initially hands `first` into the `next:` function and subsequently hands the previous result of the `next:` function into the `next:` function, as illustrated in the preceding examples.

`sequence(state:next:)`

This form is more general: it repeatedly hands `state` into the `next:` function as an `inout` parameter; the `next:` function is expected to set that parameter, using it as a scratchpad, in addition to returning the next value in the sequence.

An obvious illustration of the second form is the Fibonacci series:

```
let fib = sequence(state:(0,1)) { (pair: inout (Int,Int)) -> Int in
    let n = pair.0 + pair.1
    pair = (pair.1,n)
    return n
}
for i in fib.prefix(10) {
    print(i) // 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
}
```

Trailing Closures and Flow Control Blocks

Trailing closures do not play well with subsequent flow control blocks; therefore parentheses are sometimes needed where generally they would not be. The closing curly brace of the trailing closure, followed by the opening curly brace of the block, upsets the compiler:

```
for i in arr.map {$0*2} { // warning
    print(i)
}
```

To silence the warning, wrap the trailing closure in parentheses:

```
for i in arr.map ({$0*2}) {
    print(i)
}
```

In this example, the curly braces are not adjacent, but you get the warning anyway:

```
if arr.map {$0*2}.first == 4 { // warning
```

Again, parentheses are the simplest solution:

```
if arr.map ({$0*2}).first == 4 {
```

Any sequence can be made "lazy" by asking for its `lazy` property. This can be a source of efficiency if you're going to be looping through the sequence (explicitly or implicitly) and potentially short-circuiting the loop; there's no point generating more elements of the sequence than the loop will actually process, and that is what `lazy` prevents. Importantly, laziness propagates through a chain of sequence operations. I'll give an example in the next section.

Jumping

Although branching and looping constitute the bulk of the decision-making flow of code execution, sometimes they are insufficient to express the logic of what needs to happen next. It can be useful to interrupt your code's progress completely and *jump* to a different place within it. In this section, I'll list Swift's modes of jumping. These are all controlled forms of *early exit* from the current flow of code.

Return

You already know one form of early exit: the return statement. One function calls another, which may call another, and so on, forming a call stack. When a return statement is encountered, execution of this function is aborted and the path of execution jumps to the point where the call was made in the function one level up the call stack.

Short-circuiting and labels

Swift has several ways of short-circuiting the flow of branch and loop constructs:

`fallthrough`
: A `fallthrough` statement in a switch case aborts execution of the current case code and immediately begins executing the code of the next case.

`continue`
: A `continue` statement in a loop construct aborts execution of the current iteration and proceeds to the next iteration:

    - In a while loop, `continue` performs the conditional test immediately.
    - In a for loop, `continue` proceeds immediately to the next iteration if there is one (and aborts the loop there isn't).

`break`
: A `break` statement aborts the current construct and proceeds after the end of the construct:

    - In a loop, `break` aborts the loop.
    - In a switch case, `break` aborts the entire switch construct.

When constructs are nested, you may need to specify *which* construct you mean when you say `continue` or `break`. Therefore, Swift permits you to put a *label* before the start of an if construct, a switch statement, a while loop, or a for loop (or a do block, which I'll describe later). The label is an arbitrary name followed by a colon. You can then use that label name (without the colon) in a continue statement or a break statement within the labeled construct at any depth, to specify that this is the construct you are referring to.

To illustrate, here's a simple struct for generating prime numbers:

```
struct Primes {
    static var primes = [2]
    static func appendNextPrime() {
        next: for i in (primes.last!+1)... {
            let sqrt = Int(Double(i).squareRoot())
            for prime in primes.lazy.prefix(while: {$0 <= sqrt}) {
                if i.isMultiple(of: prime) {
                    continue next
                }
            }
            primes.append(i)
            return
        }
    }
}
```

The algorithm could be improved, but it's effective and straightforward. The struct maintains a list of the primes we've found so far, and appendNextPrime basically just looks at each successive larger integer i to see whether any of the primes we've already found (prime) divides it. If so, i is not a prime, so we want to go on to the next i. But there are two nested for loops; if we merely say continue, we'll jump to the next prime (the inner for loop), not to the next i (the outer for loop). The label solves the problem.

That example also demonstrates lazy. We want to keep prefix(while:_) from working harder than it has to; there's no point extracting *all* the primes less than the square root of i in advance, because the loop might be short-circuited. So we make primes lazy, which makes prefix(while:_) lazy, and so a prime is tested as a divisor of i only if it has to be.

Throwing and catching errors

Sometimes a situation arises where further coherent progress is impossible: the entire operation in which we are engaged has failed. It may then be desirable to abort the current scope, and possibly the current function, and possibly even the function that called that function, and so on, exiting to some point where we can acknowledge this failure and proceed in good order in some other way.

For this purpose, Swift provides a mechanism for *throwing and catching errors*. In keeping with its usual insistence on safety and clarity, Swift imposes strict conditions on the use of this mechanism, and the compiler will ensure that you adhere to them.

An *error*, in this sense, is a kind of message, presumably indicating what went wrong. This message is passed up the nest of scopes and function calls as part of the error-handling process, and the code that recovers from the failure can read it. In Swift, an error must be an object of a type that adopts the Error protocol, which has just two requirements: a String _domain property and an Int _code property. The purpose of those properties is to help errors cross the bridge between Swift and Objective-C; in real life, they are hidden and you will be unaware of them. The Error object will be one of the following:

A Swift type that adopts Error
: As soon as a Swift type formally declares adoption of the Error protocol, it is ready to be used as an error object; the protocol requirements are magically fulfilled for you behind the scenes. Typically, this type will be an enum, with different cases distinguishing different kinds of possible failure, perhaps with raw values or associated types to carry further information.

NSError
: NSError is Cocoa's class for communicating the nature of a problem; Swift extends NSError to adopt Error and bridges them to one another. If your call to a

Cocoa method generates a failure, Cocoa will send you an NSError instance typed as an Error.

There are two stages of the error mechanism to consider:

Throwing an error

Throwing an error aborts the current path of execution and hands an error object to the error mechanism.

Catching an error

Catching an error receives that error object from the error mechanism and responds in good order, with the path of execution resuming after the point of catching. In effect, we have *jumped* from the throwing point to the catching point.

To *throw an error*, use the keyword `throw` followed by an error object. That's all it takes! The current block of code is immediately aborted, and the error mechanism takes over. However, to ensure that the `throw` command is used coherently, Swift imposes a rule that you can say `throw` *only in a context where the error will be caught*. What is such a context?

The primary context for throwing and catching an error is the `do...catch` construct. This consists of a do block and one or more catch blocks. It is legal to throw in the do block; an accompanying catch block can then be fed any errors thrown from within the do block. The `do...catch` construct's schema looks like Example 5-5.

Example 5-5. The Swift `do...catch` construct

```
do {
    statements // a throw can happen here
} catch errortype {
    statements
} catch {
    statements
}
```

A single do block can be accompanied by multiple catch blocks. Catch blocks are like the cases of a switch statement, and will usually have the same logic: first, you might have specialized catch blocks, each of which is designed to handle some limited set of possible errors; finally, you might (and usually will) have a general catch block that acts as the default, mopping up any errors that were not caught by any of the specialized catch blocks.

In fact, the *syntax* used by a catch block to specify what sorts of error it catches *is* the pattern syntax used by a case in a switch statement! Imagine that this *is* a switch statement, and that the tag is the error object. Then the matching of that error object to a particular catch block is performed just as if you had written `case` instead of catch. Typically, when the Error is an enum, a specialized catch block will state at least the enum that it catches, and possibly also the case of that enum; it can have a binding, to capture the enum or its associated type; and it can have a where clause to limit the possibilities still further.

To illustrate, I'll start by defining a couple of errors:

```
enum MyFirstError : Error {
    case firstMinorMistake
    case firstMajorMistake
    case firstFatalMistake
}
enum MySecondError : Error {
    case secondMinorMistake(i:Int)
    case secondMajorMistake(s:String)
    case secondFatalMistake
}
```

And here's a do...catch construct designed to demonstrate some of the different ways we can catch different errors in different catch blocks:

```
do {
    // throw can happen here
} catch MyFirstError.firstMinorMistake, MyFirstError.firstMajorMistake {
    // catches MyFirstError.firstMinorMistake
    // also catches MyFirstError.firstMajorMistake
} catch let err as MyFirstError {
    // catches other case(s) of MyFirstError
} catch MySecondError.secondMinorMistake(let i) where i < 0 {
    // catches e.g. MySecondError.secondMinorMistake(i:-3)
} catch {
    // catches everything else
}
```

Now let's talk about how the error object makes its way into each of the catch blocks:

Catch block with pattern
In a catch block with an accompanying pattern, it is up to you to capture in the pattern any desired information about the error. If you want the error itself to travel as a variable into the catch block, you'll need a binding in the pattern.

Catch block with "mop-up" binding
A catch block whose pattern is *only* a binding catches *any* error under that name; catch let mistake is a "mop-up" catch block that catches any error as a variable called mistake.

Bare catch block
In a "mop-up" catch block with *no* accompanying pattern (that is, the bare word catch and no more), the error arrives into the block *automatically* as a variable called error.

Let's look again at the previous example, but this time we'll examine what arrives into each catch block:

```
do {
    // throw can happen here
} catch MyFirstError.firstMinorMistake, MyFirstError.firstMajorMistake {
    // no error object
    // but we know it's either MyFirstError.firstMinorMistake
    // or MyFirstError.firstMajorMistake
} catch let err as MyFirstError {
    // MyFirstError.firstFatalMistake arrives as err
} catch MySecondError.secondMinorMistake(let i) where i < 0 {
    // only i arrives, but we know it's MySecondError.secondMinorMistake
} catch {
    // error object arrives as error
}
```

So much for the `do...catch` construct. But there's something else that can happen to a thrown error; instead of being caught directly, it can percolate up the call stack, leaving the current function and arriving at the point where this function was called. In this situation, the error won't be caught here, at the point of throwing; it needs to be caught further up the call stack. This can happen in one of two ways:

Throw without a corresponding catch
: A `do...catch` construct might lack a "mop-up" catch block. Then a throw inside the do block might *not* be caught here.

Throw outside a do block
: A `throw` might occur outside of any immediate `do...catch` construct.

However, a thrown error *must* be caught *somehow*. We therefore need a way to say to the compiler: "Look, I understand that it looks like this throw is not happening in a context where it will be caught, but that's only because you're not looking far enough up the call stack. If you do look up far enough, you'll see that a throw at this point *is* eventually caught." And there is a way to say that! Use the `throws` keyword in a function declaration.

If you mark a function with the `throws` keyword, then its *entire body* becomes a legal place for throwing. The syntax for declaring a `throws` function is that the keyword `throws` appears immediately after the parameter list (and before the arrow operator, if there is one):

```
enum NotLongEnough : Error {
    case iSaidLongIMeantLong
}
func giveMeALongString(_ s:String) throws {
    if s.count < 5 {
        throw NotLongEnough.iSaidLongIMeantLong
    }
    print("thanks for the string")
}
```

The addition of throws to a function declaration creates a distinct function type. The type of giveMeALongString is not (String) -> (), but rather (String) throws -> (). If a function receives as parameter a function that can throw, that parameter's type needs to be specified accordingly:

```
    func receiveThrower(_ f:(String) throws -> ()) {
        // ...
    }
```

That function can now be called with giveMeALongString as argument:

```
    func callReceiveThrower() {
        receiveThrower(giveMeALongString)
    }
```

An anonymous function, if necessary, can include the keyword throws in its in expression, in the same place where it would appear in a normal function declaration. But this is not necessary if, as is usually the case, the anonymous function's type is known by inference:

```
    func receiveThrower(_ f:(String) throws -> ()) {
        // ...
    }
    func callReceiveThrower() {
        receiveThrower {
            s in // can say "s throws in", but not required
            if s.count < 5 {
                throw NotLongEnough.iSaidLongIMeantLong
            }
            print("thanks for the string")
        }
    }
```

New in Swift 5.5, the getter of a computed property or a subscript can be marked throws, provided there is no setter (in other words, this must be a read-only computed property or subscript). You have to write get explicitly so that there is a place to put the throws keyword. The getter becomes effectively a throws function.

To illustrate, this is a Dog with a private Optional name that starts life as nil; the public interface is a setName(_:) method along with a name property that throws if the private name is still nil:

```
    class Dog {
        private var _name : String?
        func setName(_ newName: String) {
            self._name = newName
        }
        var name : String {
            get throws {
                enum UndefinedError : Error {
                    case noName
                }
                guard let name = _name else {
                    throw UndefinedError.noName
                }
                return name
            }
        }
    }
```

When a `throws` function does throw, it aborts prematurely. This means that if the function is supposed to return a value, it *doesn't* return that value. And that's fine; throwing is an alternate form of legal exit from the function, and the compiler understands that fact.

So now we know that `throws` functions exist. But there's more. Swift imposes some requirements on the *caller* of a `throws` function:

- The caller of a `throws` function must precede the function call with the keyword `try`. This keyword acknowledges, to the programmer and to the compiler, that this function can throw.
- The function call must be made in a place where throwing is legal! A function called with `try` can throw, so saying `try` is just like saying `throw`: you must say it either in the do block of a `do...catch` construct or in the body of a `throws` function.

Swift also provides two variants of `try` that you will often use as a shorthand:

`try!`

If you are very sure that a `throws` function will in fact *not* throw, then you can call it with the keyword `try!` instead of `try`. This relieves you of all further responsibility: you can say `try!` *anywhere*, without catching the possible throw.

But be warned: if you're wrong, and this function *does* throw when your program runs, your program can crash at that moment, because you have allowed an error to percolate, uncaught, all the way up to the top of the call stack.

`try?`

Like `try!`, you can use `try?` anywhere; but, like a `do...catch` construct, `try?` catches the throw if there is one, without crashing. The downside is that if there *is* a throw, *you don't receive any error information*, as you would with a `do...catch` construct. But `try?` does *report* that there was a throw, by returning `nil`.

`try?` is useful particularly in situations where you're calling a `throws` function that returns a value and all you care about is the value. If there's a throw, `try?` returns `nil`. If there's no throw, `try?` wraps the returned value in an Optional. Commonly, you'll unwrap that Optional safely in the same line with a conditional binding.

To illustrate, here's an artificial test method that can either throw or return a String:

```
func canThrowOrReturnString(shouldThrow:Bool) throws -> String {
    enum Whoops : Error {
        case oops
    }
    if shouldThrow {
        throw Whoops.oops
    }
    return "Howdy"
}
```

We can call that method with `try` inside a `do...catch` construct:

```
do {
    let s = try self.canThrowOrReturnString(shouldThrow: true)
    print(s)
} catch {
    print(error)
}
```

At the other extreme, we can call that method with `try!` anywhere, but if the method throws, we'll crash:

```
let s = try! self.canThrowOrReturnString(shouldThrow: false)
print(s)
```

In between, we can call our method with `try?` anywhere. If the method doesn't throw, we'll receive a String wrapped in an Optional; if it does throw, we won't crash and we'll receive `nil` (but no error information):

```
if let s = try? self.canThrowOrReturnString(shouldThrow: true) {
    print(s)
} else {
    print("failed")
}
```

Just as with an Optional chain, if a `throws` function returns no value, the return type from `try?` is `Void?` — and you can capture that and compare it against `nil` to learn whether there was an error. In this example, `canThrowButReturnsNoValue` throws but returns no value:

```
let ok : Void? = try? self.canThrowButReturnsNoValue()
```

Now, if `ok` is not `nil`, no error was thrown.

Rethrows

A function that receives a throws function parameter, and that calls that function (with try), and that doesn't throw for any *other* reason, may itself be marked as rethrows instead of throws. The difference is that when a rethrows function is called, the caller can pass as argument a function that does *not* throw, and in that case the call doesn't have to be marked with try (and the calling function doesn't have to be marked with throws):

```
func receiveThrower(_ f:(String) throws -> ()) rethrows {
    try f("ok?")
}
func callReceiveThrower() { // no throws needed
    receiveThrower { s in // no try needed
        print("thanks for the string!")
    }
}
```

For a read-only computed property that throws, the rules are just the same; you have to say some form of try as you access the property. This is the Dog with the throwing name getter:

```
let d = Dog()
if let name = try? d.name {
    print("dog's name is", name)
} else {
    print("dog has no name yet")
}
```

An initializer can be a throws function. Such an initializer, too, must be called with some form of try. To illustrate, here's a different Dog:

```
class Dog {
    let name : String
    init(name:String) throws {
        enum Invalid : Error { case empty }
        if name.isEmpty {
            throw Invalid.empty
        }
        self.name = name
    }
}
let dog = try? Dog(name:"Fido")
```

As in that example, when such an initializer *does* throw, it may have failed to fulfill the initializer contract of initializing all stored properties. But the compiler doesn't care, because the initializer has failed. The situation is no different from when a failable initializer prematurely returns nil.

In designing an initializer, when should you prefer a failable initializer and when should you prefer a `throws` initializer? No hard and fast rule can be given; it depends on your overall design goals. In general, `init?` implies simple failure to create an instance, whereas `throws` implies that there is useful information to be gleaned by studying the error.

Even if your own code never uses the keyword `throw` explicitly, you're still very likely, in real life, to call Cocoa methods that are marked with `throws`. (For the details of how the error mechanism works in Objective-C and how this is bridged to the Swift `throws` mechanism, see the Appendix.) Objective-C will be supplying an NSError; this class is bridged to Swift Error. Swift helps you cross the bridge by giving Error a `localizedDescription` property, allowing you to read NSError's `localizedDescription`. Moreover, you can catch a specific NSError by its name. The name you'll use is the NSError `domain`, and optionally (with dot-notation) the Cocoa name of its `code`.

For example, NSString `initWithContentsOfFile:encoding:error:` appears to Swift as a throwing initializer:

```
init(contentsOfFile path: String, encoding enc: String.Encoding) throws
```

So let's say we call that initializer, and we want specifically to catch the error thrown when there is no such file. This NSError's `domain`, according to Cocoa, is `"NSCocoaErrorDomain"`. Its `code` is 260, for which Cocoa provides the name NSFileReadNoSuchFileError (I found that out by looking in the *FoundationErrors.h* header file in Objective-C). The Swift Foundation overlay translates those into CocoaError and `.fileReadNoSuchFile` respectively, so we can catch the error like this:

```
do {
    let f = // path to some file, maybe
    let s = try String(contentsOfFile: f)
    // ... if successful, do something with s ...
} catch CocoaError.fileReadNoSuchFile {
    print("no such file")
} catch {
    print(error)
}
```

Objective-C sees a Swift error coherently as well. By default, it receives a Swift error as an NSError whose `domain` is the name of the Swift object type. If the Swift object type is an enum, the NSError's `code` is the index number of its case; otherwise, the `code` is 1. When you want to provide Objective-C with a fuller complement of information, make your error type adopt one or both of these protocols:

LocalizedError

Adopts Error, adding three optional properties: `errorDescription` (NSError `localizedDescription`), `failureReason` (NSError `localizedFailureReason`), and `recoverySuggestion` (NSError `localizedRecoverySuggestion`). Observe that these are `String?` properties; declaring them as simple String rather than Optional fails to communicate the information to Objective-C, and is a common mistake.

CustomNSError

Adopts Error, adding three properties with default implementations: `errorDomain`, `errorCode`, and `errorUserInfo`, which Objective-C will see as the NSError's `domain`, `code`, and `userInfo`.

Nested scopes

When a local variable needs to exist only for a few lines of code, you might like to define an artificial scope — a custom nested scope, at the start of which you can introduce your local variable, and at the end of which that variable will be permitted to go out of scope, destroying its value automatically. Swift does not permit you to use bare curly braces to do this. Instead, use a bare do block without a `catch`.

Here's a rewrite of our earlier code (Chapter 4) for uniquing an array while keeping its order:

```
var arr = ["Manny", "Manny", "Moe", "Jack", "Jack", "Moe", "Manny"]
do {
    var temp = Set<String>()
    arr = arr.filter { temp.insert($0).inserted }
}
```

The only purpose of `temp` is to act as a "helper" for this one `filter` call. So we embed its declaration and the `filter` call in a bare do block; that way, `temp` is in a lower scope that doesn't clutter up our local namespace, and as soon as the `filter` call is over, we exit the block and `temp` is destroyed.

Another use of a bare do block is to implement the simplest form of early exit. The do block gives you a scope to jump out of; now you can label the do block and break to that label:

```
out: do {
    // ...
    if somethingBadHappened {
        break out
    }
    // we won't get here if somethingBadHappened
}
```

Defer statement

A defer statement applies to the scope in which it appears, such as a function body, a while block, an if construct, a do block, and so on. Wherever you say `defer`, curly braces surround it somehow; the defer block will be executed *when the path of execution leaves those curly braces*. Leaving the curly braces can involve reaching the last line of code within the curly braces, or any of the forms of early exit described earlier in this section.

To see one reason why this is useful, consider the following property:

`self.view.window?.isUserInteractionEnabled`
: A Bool. When set to `false`, prevents all user touches from reaching any view within the window (which usually means any view of the whole app). When set to `true`, restores the ability of user touches to reach those views.

It can be valuable to turn off user touch interactions at the start of some slightly time-consuming operation and then turn them back on after that operation, especially when, during the operation, the interface or the app's logic will be in some state where the user's tapping a button, say, could cause things to go awry. It is not uncommon for a method to be constructed like this:

```
func doSomethingTimeConsuming() {
    self.view.window?.isUserInteractionEnabled = false
    // ... do stuff ...
    self.view.window?.isUserInteractionEnabled = true
}
```

All well and good — *if* we can guarantee that the only path of execution out of this function will be by way of that last line. But what if we need to return early from this function? Our code now looks like this:

```
func doSomethingTimeConsuming() {
    self.view.window?.isUserInteractionEnabled = false
    // ... do stuff ...
    if somethingHappened {
        return
    }
    // ... do more stuff ...
    self.view.window?.isUserInteractionEnabled = true
}
```

Oops! We've just made a terrible mistake. By providing an additional path out of our `doSomethingTimeConsuming` function, we've created the possibility that our code might never encounter the call to set `isUserInteractionEnabled` to `true`. We might leave our function by way of the return statement — and the user will then be left unable to interact with the interface.

Obviously, we need to add another `isUserInteractionEnabled = true` call inside the if construct, just before the return statement. But as we continue to develop our code, we must remember, if we add *further* ways out of this function, to add *yet another* `isUserInteractionEnabled = true` call for *each* of them. This is madness!

The defer statement solves the problem. It lets us specify *once* what should happen when we leave this scope, *no matter how*. Our code now looks like this:

```
func doSomethingTimeConsuming() {
    self.view.window?.isUserInteractionEnabled = false
    defer {
        self.view.window?.isUserInteractionEnabled = true
    }
    // ... do stuff ...
    if somethingHappened {
        return
    }
    // ... do more stuff ...
}
```

The `isUserInteractionEnabled = true` call in the defer block will be executed, not where it appears, but before the return statement, or before the last line of the method — whichever path of execution ends up leaving the function. The defer statement says: "Eventually, and as late as possible, be sure to execute this code." We have *ensured* the necessary balance between turning off user interactions and turning them back on again. Most uses of the defer statement will probably come under this same rubric: you'll use it to balance a command or recover from a disturbed state.

Observe that in the preceding code, I placed the defer statement very early in its surrounding scope. This placement is important because a defer statement is itself a command. If a defer statement is not actually *encountered* by the path of execution before we exit from the surrounding scope, *its block won't be executed.* For this reason, always place your defer statement as close to the start of its surrounding block as you can, to ensure that it will in fact be encountered.

When a defer statement changes a value that is returned by a return statement, the return happens first and the defer statement happens second. In other words, `defer` effectively lets you return a value *and then change it.* This example comes from Apple's own code (in the documentation, demonstrating how to write a struct that can adopt the Sequence protocol):

```
struct Countdown: Sequence, IteratorProtocol {
    var count: Int
    mutating func next() -> Int? {
        if count == 0 {
            return nil
        } else {
            defer { count -= 1 }
```

```
                return count
            }
        }
    }
```

That code returns the current value of `count` and *then* decrements `count`, ready for the next call to the `next` method. Without `defer`, we'd decrement `count` and then return the decremented value, which is not what's wanted.

If the current scope has multiple defer blocks pending, they will be called in the reverse of the order in which they were originally encountered. In effect, there is a defer *stack*; each successive defer statement, as it is encountered, pushes its code onto the top of the stack, and exiting the scope in which a defer statement appeared pops that code and executes it.

Aborting the whole program

Aborting the whole program is an extreme form of flow control; the program stops dead in its tracks. In effect, you have deliberately crashed your own program. This is an unusual thing to do, but it can be useful as a way of raising a very red flag: you don't really *want* to abort, so if you *do* abort, things must be so bad that you've no choice.

One way to abort is by calling the global function `fatalError`. It takes a String parameter permitting you to provide a message to appear in the console. I've already given this example:

```
    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
```

That code says, in effect, that execution should *never* reach this point. We have declared `init(coder:)` just because it is `required`, and we need to satisfy the compiler; but we have no real implementation of `init(coder:)`, and we do not expect to be initialized this way. If we *are* initialized this way, something has gone very wrong, and we *want* to crash, because our program has a serious bug.

Like throwing an error, a `fatalError` call aborts, and the compiler understands that fact. An initializer can call `fatalError` without having initialized stored properties. Similarly, a function that returns a value does not have to return any value if a `fatalError` call is encountered.

You can abort conditionally by calling the `assert` function. Its first parameter is a condition — something that evaluates as a Bool. If the condition is `false`, we will abort; the second parameter is a String message to appear in the console if we *do* abort. The idea here is that you are making a bet (an *assertion*) that the condition is `true` — a bet that you feel so strongly about that if the condition is `false`, there's a serious bug in your program and you want to crash so you can learn of this bug and fix it.

By default, assert works only when you're developing your program. When your program is to be finalized and made public, you throw a build switch that tells the compiler (among other things) that assert should be ignored. In effect, the conditions in your assert calls are then disregarded; they are all seen as true. This means that you can safely leave assert calls in your code. By the time your program ships, of course, none of your assertions should be failing; any bugs that caused them to fail should already have been ironed out.

The disabling of assertions in shipping code is performed in an interesting way. The condition parameter is given an extra layer of indirection by declaring it as an @autoclosure function. This means that, even though the parameter is *not* in fact a function, the compiler will wrap it in a function; the runtime won't call that function unless it has to. In shipping code, the runtime will *not* call that function. This mechanism averts expensive and unnecessary evaluation: an assert condition test may involve side effects, but the test won't even be performed when assertions are turned off in your shipping program.

In addition, Swift provides the assertionFailure function. It's like an assert that always fails — and, like an assert, it *doesn't* fail in your shipping program where assertions are turned off. It's a convenient synonym for assert(false), as a way of assuring yourself that your code never goes where it's never supposed to go.

Finally, precondition and preconditionFailure are similar to assert and assertionFailure, except that they *do* fail even in a shipping program.

Guard

When your code needs to decide whether to exit early, Swift provides a special syntax — the guard construct. In effect, a guard construct is an if construct where you exit early if the condition fails. Its form is shown in Example 5-6.

Example 5-6. The Swift guard construct

```
guard condition else {
    statements
    exit
}
```

A guard construct consists solely of a condition and an else block. The else block *must* jump out of the current scope, by any of the means that Swift provides, such as return, break, continue, throw, or fatalError — anything that guarantees to the compiler that, in case of failure of the condition, execution absolutely will not proceed within the block that contains the guard construct.

Because the guard construct guarantees an exit on failure of the condition, the compiler knows that the condition has succeeded after the guard construct if we do *not* exit. An elegant consequence is that a conditional binding in the condition is in scope *after* the guard construct, without introducing a further nested scope:

```
guard let s = optionalString else {return}
// s is now a String (not an Optional)
```

A guard construct's conditional binding can't use, on the left side of the equal sign, a name already declared in the same scope. This is illegal:

```
let s = // ... some Optional
guard let s = s else {return} // compile error
```

The reason is that `guard let`, unlike `if let` and `while let`, doesn't declare the bound variable for a *nested* scope; it declares it for *this* scope. We can't declare `s` here because `s` has already been declared in the same scope.

In my own code, it's not uncommon to have a series of guard constructs, one after another. This may seem a rather clunky and imperative mode of expression, but I'm fond of it nevertheless. It's a nice alternative to a single elaborate if construct, or to the "pyramid of doom" that I discussed earlier; and it looks like exactly what it is, a sequence of gates through which the code must pass in order to proceed further. Here's an actual example from my real-life code:

```
@objc func tapField(_ g: Any) {
    // g must be a gesture recognizer
    guard let g = g as? UIGestureRecognizer else {return}
    // and the gesture recognizer must have a view
    guard g.view != nil else {return}
    // okay, now we can proceed...
}
```

It's often possible to combine multiple guard statement conditions into a single condition list:

```
@objc func tapField(_ g: Any) {
    // g must be a gesture recognizer
    // and the gesture recognizer must have a view
    guard let g = g as? UIGestureRecognizer, g.view != nil
        else {return}
    // okay, now we can proceed...
}
```

A guard construct will also come in handy in conjunction with `try?`. Let's presume we can't proceed unless `String(contentsOfFile:)` succeeds. Then we can call it like this:

```
let f = // path to some file, maybe
guard let s = try? String(contentsOfFile: f) else {return}
// s is now a String (not an Optional)
```

There is also a `guard case` construct, forming the logical inverse of `if case`. To illustrate, we'll use our MyError enum once again:

```
guard case let .number(n) = err else {return}
// n is now the extracted number
```

`guard case` helps to solve an interesting problem. Suppose we have a function whose returned value we want to check in a `guard` statement:

```
guard howMany() > 10 else {return}
```

All well and good; but suppose also that in the *next* line we want to *use* the value returned from that function. We don't want to call the function *again*; it might be time-consuming and it might have side effects. We want to *capture* the result of calling the function and pass that captured result on into the subsequent code. But we can't do that with `guard let`, because that requires an Optional, and our function `howMany` doesn't return an Optional. `guard case` to the rescue:

```
guard case let output = howMany(), output > 10 else {return}
// now output is in scope
```

Privacy

Privacy (also known as *access control*) refers to the explicit modification of the normal scope rules. I gave an example in Chapter 1:

```
class Dog {
    var name = ""
    private var whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
}
```

The intention here is to limit how other objects can see the Dog property `whatADogSays`. It is a private property, intended primarily for the Dog class's own internal use: a Dog can speak of `self.whatADogSays`, but other objects should not be aware that it even exists.

Swift has five levels of privacy:

`internal`

The default rule is that declarations are *internal*, meaning that they are globally visible *within the containing module*. That is why Swift files within the same module can see one another's top-level contents automatically, with no effort on your part. (That's different from C and Objective-C, where files can't see each other at all unless you explicitly show them to one another through `include` or `import` statements.)

`fileprivate` *(narrower than* `internal`*)*

A thing declared `fileprivate` is visible *only within its containing file*. Two object types declared in the same file can see one another's members declared `fileprivate`, but code in other files cannot see those members.

`private` *(even narrower than* `fileprivate`*)*

A thing declared `private` is visible *only within its containing curly braces*. In effect, the visibility of an object type's member declared `private` is limited to code within this type declaration. (A `private` declaration at the top level of a file is equivalent to `fileprivate`.)

`public` *(wider than* `internal`*)*

A thing declared `public` is visible *even outside its containing module*. Another module must first import this module before it can see anything at all. But even when another module *has* imported this module, it *still* won't be able to see anything in this module that hasn't been explicitly declared `public`. If you don't write any modules, you might never need to declare anything `public`. If you do write a module, you *must* declare *something* `public`, or your module is useless.

`open` *(even wider than* `public`*)*

If a class is declared `open`, code in another module can subclass it; it can't do that if the class is declared merely `public`. If an open class member is declared `open`, code in another module that subclasses this class can override this member; it can't do that if the member is declared merely `public`.

Private and Fileprivate

Declaring something `private` restricts its visibility. In this way, you specify by inversion what the public API of this object is. Here's an example from my own code:

```
class CancelableTimer: NSObject {
    private var q = DispatchQueue(label: "timer")
    private var timer : DispatchSourceTimer!
    private var firsttime = true
    private var once : Bool
    private var handler : () -> ()
    init(once:Bool, handler:@escaping () -> ()) {
        // ...
    }
    func start(withInterval interval:Double) {
        // ...
    }
```

```
        func cancel() {
            // ...
        }
    }
```

The initializer `init(once:handler:)` and the `start(withInterval:)` and `cancel` methods, which are *not* marked `private`, are this class's public API. They say, "Please feel free to call me!" The properties, however, are all private; no other code can see them, either to get them or to set them. They are purely for the internal use of the methods of this class. They maintain state, but it is not a state that any other code needs to know about.

Privacy is not magically violated by the existence of a special object relationship. Even a subclass cannot see its superclass's private members. (This comes as a surprise to those coming from a language with a `protected` privacy level.) You can work around this by declaring the class and its subclass in the same file and declaring those members `fileprivate` instead of `private`.

A nested type can see the private members of the type in which it is nested. This makes sense, because the outer type is a surrounding scope; the nested type sees what everything else inside this type sees.

An extension can see the private members of the type it extends, provided the type and the extension are in the same file:

```
    class Dog {
        private var whatADogSays = "woof"
    }
    extension Dog {
        func speak() {
            print(self.whatADogSays) // ok
        }
    }
```

In effect, an extension sees its type's `private` as meaning `fileprivate`. This lets you break up a type into extensions without being forced to raise the type's private members to `fileprivate` just so the extensions can see them.

It may be that on some occasions you will want to draw a distinction between the privacy of a variable regarding setting and its privacy regarding getting. To draw this distinction, place the word `set` in parentheses after its own privacy declaration. `private(set) var myVar` means that the *setting* of this variable is restricted, but says nothing about the *getting* of this variable, which is left at the default. Similarly, you can say `public private(set) var myVar` to make getting this variable public, while setting this variable is kept private.

The existence of Objective-C adds complications. Things marked `@objc` can be marked `private` without harm; Objective-C can still see them. (That includes things marked `@IBAction` and `@IBOutlet`, which imply `@objc`.) But your implementation of a member defined by an Objective-C protocol cannot be marked `private` without hiding it from Objective-C. In particular, optional methods defined by a Cocoa protocol must be at least `internal` (the default), or Cocoa won't be able to find them and won't call them. You are forced to expose these methods to other files in your module, as if they were part of this class's public API, even though you would probably prefer not to.

Public and Open

If you write a module, you'll need to specify at least some object type declaration as `public`; otherwise, code that imports your module won't be able to see that type. Other declarations that are not declared `public` are internal, meaning that they are private to the module. Judicious use of `public` declarations configures the public API of your module.

The members of a public object type are not, themselves, automatically public. If you want a method to be public, you have to declare it `public`. This is an excellent default behavior, because it means that these members are not shared outside the module unless you want them to be. (As Apple puts it, you must "opt in to publishing" object members.)

In my Zotz app, which is a card game, the files declaring object types for creating and portraying cards and for combining them into a deck are bundled into a framework called ZotzDeck. A framework is a module. The idea is for these files to be able to see one another freely while limiting access from the rest of my app. Many of the ZotzDeck types, such as Card and Deck, are declared `public`. Many utility object types, however, are not; the classes within the ZotzDeck module can see and use them, but code outside the module doesn't need to be aware of them. Moreover, the Card class is declared `public` but its initializer is not, because the public way to get cards is by initializing a Deck; the initializer for Deck *is* declared `public`, so you can do that.

If the only initializer for a public type is implicit, code in another module can't see it and cannot create an instance of this type. If you want other code to be able to create an instance of this type, you must declare the initializer explicitly and make it public.

The `open` access level draws a further distinction. It is applicable only to classes and to members of open classes. A public class can't be subclassed in another module that can see this class; an open class can. A public member of an open class that has been subclassed in another module can't be overridden in that subclass; an open member can.

Privacy Rules

There is an extensive set of rules for ensuring that the privacy level of related things is coherent. Here are some of them:

- A variable can't be public if its type is private, because other code wouldn't be able to use such a variable.
- A subclass can't be public unless the superclass is public.
- A subclass can change an overridden member's access level, but it cannot even *see* its superclass's private members unless they are declared in the same file together.

And so on. I could proceed to list all the rules, but I won't. There is no need for me to enunciate them formally. They are spelled out in great detail in the Swift manual, which you can consult if you need to. In general, you probably won't need to; the privacy rules make intuitive sense, and you can rely on the compiler to help you with useful error messages if you violate one.

Introspection

Swift provides limited ability to *introspect* an object, letting an object display the names and values of its properties. This feature is intended for debugging, not for use in your program's logic. For example, you can use it to modify the way your object is displayed in the Xcode Debug pane.

To introspect an object, use it as the `reflecting:` parameter when you instantiate a Mirror. The Mirror's `children` will then be name–value tuples describing the original object's properties. Here's a Dog class with a `description` property that takes advantage of introspection. Instead of hard-coding a list of the class's instance properties, we introspect the instance to obtain the names and values of the properties. This means that we can later add more properties without having to modify our `description` implementation:

```
struct Dog : CustomStringConvertible {
    var name = "Fido"
    var license = 1
    var description : String {
        var desc = "Dog ("
        let mirror = Mirror(reflecting:self)
        for (k,v) in mirror.children {
            desc.append("\(k!): \(v), ")
        }
        return desc.dropLast(2) + ")"
    }
}
```

If we now instantiate Dog and `print` that instance, this is what we see in the console:

```
Dog (name: Fido, license: 1)
```

The main use of Mirror is to generate the console output for the Swift `dump` function (or the `po` command when debugging). By adopting the CustomReflectable protocol, we can take charge of what a Mirror's `children` are. To do so, we implement the `customMirror` property to return our own custom Mirror object whose `children` property we have configured as a collection of name–value tuples.

In this (silly) example, we implement `customMirror` to supply altered names for our properties:

```
struct Dog : CustomReflectable {
    var name = "Fido"
    var license = 1
    var customMirror: Mirror {
        let children : [Mirror.Child] = [
            ("ineffable name", self.name),
            ("license to kill", self.license)
        ]
        let m = Mirror(self, children:children)
        return m
    }
}
```

The outcome is that when our code says `dump(Dog())`, our custom property names are displayed:

```
* Dog
  - ineffable name : "Fido"
  - license to kill : 1
```

Operators

Swift operators such as `+` and `>` are not magically baked into the language. They are, in fact, functions; they are explicitly declared and implemented just like any other function. That is why, as I pointed out in Chapter 4, the term `+` can be passed as the second parameter in a `reduce` call; `reduce` expects a function taking two parameters and returning a value whose type matches that of the first parameter, and `+` *is* in fact the name of such a function. It also explains how Swift operators can be overloaded for different types of operand. You can use `+` with numbers, strings, or arrays — with a different meaning in each case — because Swift functions can be overloaded; there are multiple declarations of the `+` function, and Swift is able to determine from the parameter types *which* `+` function you are calling.

These facts are not merely an intriguing behind-the-scenes implementation detail. They have practical implications for you and your code. You are free to overload existing operators to apply to *your* object types. You can even invent *new* operators! In this section, we'll do both.

First we'll talk about how operators are declared. Clearly there is some sort of syntactical hanky-panky (a technical computer science term), because you don't *call* an operator function in the same way as a normal function. You don't say `+(1,2)`; you say `1+2`. Even so, `1` and `2` in that second expression *are* the parameters to a `+` function call. How does Swift know that the `+` function uses this special syntax?

To see the answer, look in the Swift header:

```
infix operator + : AdditionPrecedence
```

That is an operator declaration. An operator declaration announces that this symbol *is* an operator, and specifies how many parameters it has and what the usage syntax will be in relation to those parameters. The really important part is the stuff before the colon: the keyword `operator`, preceded by an operator *type* — here, `infix` — and followed by the name of the operator. The types are:

`infix`
: This operator takes two parameters and appears between them.

`prefix`
: This operator takes one parameter and appears before it.

`postfix`
: This operator takes one parameter and appears after it.

The term after the colon in an operator declaration is the name of a precedence group. Precedence groups dictate the order of operations when an expression contains multiple operators. I'm not going to go into the details of how precedence groups are defined. The Swift header declares about a dozen precedence groups, and you can easily see how those declarations work. You will probably have no need to declare a new precedence group; you'll just look for an operator similar to yours and copy its precedence group (or omit the colon and the precedence group from your declaration).

An operator is also a function, so you also need a function declaration stating the type of the parameters and the result type of the function. Again, the Swift header shows us an example:

```
func +(lhs: Int, rhs: Int) -> Int
```

That is one of many declarations for the `+` function in the Swift header. In particular, it is the declaration for when the parameters are both Int. In that situation, the result is itself an Int. (The local parameter names `lhs` and `rhs`, which don't affect the special calling syntax, presumably stand for "left-hand side" and "right-hand side.")

An operator declaration must appear at the top level of a file. The corresponding function declaration may appear either at the top level of a file or at the top level of a type declaration; in the latter case, it must be marked `static`. If the operator is a prefix or postfix operator, the function declaration must start with the word prefix or postfix; the default is infix and can be omitted.

We now know enough to override an operator to work with an object type of our own! As a simple example, imagine a Vial full of bacteria:

```
struct Vial {
    var numberOfBacteria : Int
    init(_ n:Int) {
        self.numberOfBacteria = n
    }
}
```

When two Vials are combined, you get a Vial with all the bacteria from both of them. So the way to add two Vials is to add their bacteria:

```
extension Vial {
    static func +(lhs:Vial, rhs:Vial) -> Vial {
        let total = lhs.numberOfBacteria + rhs.numberOfBacteria
        return Vial(total)
    }
}
```

And here's code to test our new + operator override:

```
let v1 = Vial(500_000)
let v2 = Vial(400_000)
let v3 = v1 + v2
print(v3.numberOfBacteria) // 900000
```

In the case of a compound assignment operator, the first parameter is the thing being assigned to. Therefore, to implement such an operator, the first parameter must be declared inout. Let's do that for our Vial class:

```
extension Vial {
    static func +=(lhs:inout Vial, rhs:Vial) {
        let total = lhs.numberOfBacteria + rhs.numberOfBacteria
        lhs.numberOfBacteria = total
    }
}
```

Here's code to test our += override:

```
var v1 = Vial(500_000)
let v2 = Vial(400_000)
v1 += v2
print(v1.numberOfBacteria) // 900000
```

Next, let's invent a completely new operator. As an example, I'll inject an operator into Int that raises one number to the power of another. As my operator symbol, I'll use ^^ (I'd like to use ^ but it's already in use for something else). For simplicity, I have omitted error-checking for edge cases (such as exponents less than 1):

```
infix operator ^^
extension Int {
    static func ^^(lhs:Int, rhs:Int) -> Int {
        var result = lhs
        for _ in 1..<rhs {result *= lhs}
        return result
    }
}
```

That's all it takes! Here's some code to test it:

```
print(2^^2) // 4
print(2^^3) // 8
print(3^^3) // 27
```

Here's another example. I've already illustrated the use of Range's `reversed` method to allow iteration from a higher value to a lower one. That works, but I find the notation unpleasant. There's an asymmetry with how you iterate up; the endpoints are in the wrong order, and you have to remember to surround a literal range with parentheses:

```
let r1 = 1..<10
let r2 = (1..<10).reversed()
```

Let's define a custom operator that calls `reversed()` for us:

```
infix operator >>> : RangeFormationPrecedence
func >>><Bound>(maximum: Bound, minimum: Bound)
    -> ReversedCollection<Range<Bound>>
    where Bound : Strideable {
        return (minimum..<maximum).reversed()
}
```

Now our expressions can be more symmetrical and compact:

```
let r1 = 1..<10
let r2 = 10>>>1
```

The Swift manual lists the special characters that can be used as part of a custom operator name:

```
/ = - + ! * % < > & | ^ ? ~
```

An operator name can also contain many other symbol characters (that is, characters that can't be mistaken for some sort of alphanumeric) that are harder to type; see the manual for a formal list.

Memory Management

Swift memory management is handled automatically, and you will usually be unaware of it. Objects come into existence when they are instantiated and go out of existence as soon as they are no longer needed. Nevertheless, there are some memory management issues of which you must be conscious.

Memory Management of Reference Types

Memory management of reference type objects ("Value Types and Reference Types" on page 157) is quite tricky under the hood; I'll devote Chapter 13 to a discussion of the underlying mechanism. Swift normally does all the work for you, but trouble can arise when two class instances have references to one another. When that's the case, you can have a *retain cycle* which will result in a *memory leak*, meaning that the two instances *never* go out of existence. Some computer languages solve this sort of problem with a periodic "garbage collection" phase that detects retain cycles and cleans them up, but Swift doesn't do that; you have to fend off retain cycles manually.

One way to test for and observe a memory leak is to implement a class's `deinit`. This method is called when the instance goes out of existence. If the instance never goes out of existence, `deinit` is never called. That's a bad sign, if you were expecting that the instance *should* go out of existence.

Here's an example. First, I'll make two class instances and watch them go out of existence:

```
func testRetainCycle() {
    class Dog {
        deinit {
            print("farewell from Dog")
        }
    }
    class Cat {
        deinit {
            print("farewell from Cat")
        }
    }
    let d = Dog()
    let c = Cat()
}
testRetainCycle() // farewell from Cat, farewell from Dog
```

When we run that code, both "farewell" messages appear in the console. We created a Dog instance and a Cat instance, but the only references to them are automatic (local) variables inside the `testRetainCycle` function. When execution of that function's body comes to an end, all automatic variables are destroyed; that is what it means to be an automatic variable. There are no other references to our Dog and Cat instances that might make them persist, and so they are destroyed in good order.

Now I'll change that code by giving the Dog and Cat objects references to each other:

```
func testRetainCycle() {
    class Dog {
        var cat : Cat?
        deinit {
            print("farewell from Dog")
        }
    }
    class Cat {
        var dog : Dog?
        deinit {
            print("farewell from Cat")
        }
    }
    let d = Dog()
    let c = Cat()
    d.cat = c // create a...
    c.dog = d // ...retain cycle
}
testRetainCycle() // nothing in console
```

When we run that code, *neither* "farewell" message appears in the console. The Dog and Cat objects have references to one another. Those are *strong* references (also called *persisting* references). A strong reference sees to it that as long as our Dog has a reference to a particular Cat, that Cat will not be destroyed, and vice versa. That's a good thing, and is a fundamental principle of sensible memory management. The bad thing is that the Dog and the Cat have strong references *to one another*. That's a retain cycle! Neither the Dog instance nor the Cat instance can be destroyed, because neither of them can "go first" — it's like Alphonse and Gaston who can never get through the door because each requires the other to precede him. The Dog can't be destroyed first because the Cat has a strong reference to it, and the Cat can't be destroyed first because the Dog has a strong reference to it.

These objects are now *leaking*. Our code is over; both `d` and `c` are gone. There are *no* further references to either of these objects; neither object can ever be referred to again. No code can mention them; no code can reach them. But they live on, floating, useless, and taking up memory.

The term "retain cycle" is based on the `retain` command, which is used in Objective-C to form a strong reference. You can't say `retain` in Swift, so Apple often refers to this kind of cycle as a *strong reference cycle*. But a strong reference cycle *is* a retain cycle — the compiler is in fact inserting `retain` commands under the hood — and I'll continue calling it a retain cycle.

Weak references

One solution to a retain cycle is to mark the problematic reference as `weak`. This means that the reference is *not* a strong reference. It is a *weak reference*. The object referred to can now go out of existence even while the referrer continues to exist. Of course, this might present a danger, because now the object referred to may be destroyed behind the referrer's back. But Swift has a solution for that, too: only an Optional reference can be marked as `weak`. That way, if the object referred to *is* destroyed behind the referrer's back, the referrer will see something coherent, namely `nil`. Also, the reference must be a `var` reference, precisely because it can change spontaneously to `nil`.

This code breaks the retain cycle and prevents the memory leak:

```
func testRetainCycle() {
    class Dog {
        weak var cat : Cat?
        deinit {
            print("farewell from Dog")
        }
    }
    class Cat {
        weak var dog : Dog?
        deinit {
            print("farewell from Cat")
        }
    }
    let d = Dog()
    let c = Cat()
    d.cat = c
    c.dog = d
}
testRetainCycle() // farewell from Cat, farewell from Dog
```

I've gone overboard in that code. To break the retain cycle, there's no need to make *both* Dog's `cat` and Cat's `dog` weak references; making just *one* of the two a weak reference is sufficient to break the cycle. That, in fact, is the usual solution when a retain cycle threatens. One of the pair will typically be more of an "owner" than the other; the one that is *not* the "owner" will have a weak reference to its "owner."

Value types are not subject to the same memory management issues as reference types, but a value type can still be *involved* in a retain cycle with a class instance. In my retain cycle example, if Dog is a class and Cat is a struct, we still get a retain cycle. The solution is the same: make Cat's `dog` a weak reference. (You can't make Dog's `cat` a weak reference if Cat is a struct; only a reference to a class type can be declared `weak`.)

Do *not* use weak references unless you have to! Memory management is not to be toyed with lightly. Nevertheless, there are real-life situations in which weak references are the right thing to do, even when no retain cycle appears to threaten. The delegation pattern (Chapter 12) is a typical case in point; an object typically has no business owning (retaining) its delegate. And a view controller `@IBOutlet` property is usually `weak`, because it refers to a subview already owned by its own superview.

Unowned references

There's another Swift solution for retain cycles. Instead of marking a reference as weak, you can mark it as unowned. This approach is useful in special cases where one object absolutely cannot exist without a reference to another, but where this reference need not be a strong reference.

Let's pretend that a Boy may or may not have a Dog, but every Dog must have a Boy — and so I'll give Dog an init(boy:) initializer. The Dog needs a reference to its Boy, and the Boy needs a reference to his Dog if he has one; that's potentially a retain cycle:

```
func testUnowned() {
    class Boy {
        var dog : Dog?
        deinit {
            print("farewell from Boy")
        }
    }
    class Dog {
        let boy : Boy
        init(boy:Boy) { self.boy = boy }
        deinit {
            print("farewell from Dog")
        }
    }
    let b = Boy()
    let d = Dog(boy: b)
    b.dog = d
}
testUnowned() // nothing in console
```

We can solve this by declaring Dog's boy property unowned:

```
func testUnowned() {
    class Boy {
        var dog : Dog?
        deinit {
            print("farewell from Boy")
        }
    }
    class Dog {
        unowned let boy : Boy // *
        init(boy:Boy) { self.boy = boy }
        deinit {
            print("farewell from Dog")
        }
    }
    let b = Boy()
```

```
        let d = Dog(boy: b)
        b.dog = d
    }
    testUnowned() // farewell from Boy, farewell from Dog
```

An advantage of an `unowned` reference is that it doesn't have to be an Optional and it can be a constant (`let`). But an `unowned` reference is also *genuinely* dangerous, because the object referred to can go out of existence behind the referrer's back, and an attempt to use that reference will cause a crash, as I can demonstrate by this rather forced code:

```
    var b = Optional(Boy())
    let d = Dog(boy: b!)
    b = nil // destroy the Boy behind the Dog's back
    print(d.boy) // crash
```

Clearly you should use `unowned` only if you are absolutely certain that the object referred to will outlive the referrer.

Stored anonymous functions

A particularly insidious kind of retain cycle arises when an instance property holds a function referring to the instance:

```
    class FunctionHolder {
        var function : (() -> ())?
        deinit {
            print("farewell from FunctionHolder")
        }
    }
    func testFunctionHolder() {
        let fh = FunctionHolder()
        fh.function = {
            print(fh)
        }
    }
    testFunctionHolder() // nothing in console
```

Oops! I've created a retain cycle, by referring, inside the anonymous function, to the object that is holding a reference to it. Because functions are closures, the FunctionHolder instance `fh`, declared outside the anonymous function, is captured by the anonymous function as a strong reference when the anonymous function says `print(fh)`. But the anonymous function has also been assigned to the `function` property of the FunctionHolder instance `fh`, and that's a strong reference too. So that's a retain cycle: the FunctionHolder persistently refers to the function, which persistently refers to the FunctionHolder.

In this situation, I *cannot* break the retain cycle by declaring the `function` property as `weak` or `unowned`. Only a reference to a class type can be declared `weak` or `unowned`, and a function is not a class. I must declare the captured value fh *inside the anonymous function* as weak or unowned instead.

To do so, use the anonymous function's capture list ("Capture Lists" on page 60). In the square brackets, put a comma-separated list of any problematic references that would be captured from the surrounding environment, each preceded by weak or unowned:

```
class FunctionHolder {
    var function : (() -> ())?
    deinit {
        print("farewell from FunctionHolder")
    }
}
func testFunctionHolder() {
    let fh = FunctionHolder()
    fh.function = {
        [weak fh] in // *
        print(fh)
    }
}
testFunctionHolder() // farewell from FunctionHolder
```

That syntax solves the problem. But marking a reference as weak in a capture list has a side effect: the reference passes into the anonymous function as an Optional. This is good, because it means that if the object referred to goes out of existence behind our back, the value of the Optional is nil. But of course you must also adjust your code accordingly, unwrapping the Optional as needed in order to use it. The usual technique is to perform the *weak–strong dance*: you unwrap the Optional once, right at the start of the function, in a conditional binding:

```
fh.function = {
    [weak fh] in  // weak
    guard let fh = fh else { return }
    print(fh)     // strong (and not Optional)
}
```

The conditional binding let fh = fh elegantly accomplishes three goals:

- It unwraps the Optional version of fh that arrived into the anonymous function.
- It declares another fh that is a normal (strong) reference. So if the unwrapping succeeds, this new fh will persist for the rest of this scope.
- It causes the second fh to overshadow the first fh (because they have the same name). So it is impossible after the guard statement to refer accidentally to the weak Optional fh.

Now, it happens that, in this particular example, there is no way the FunctionHolder instance fh can go out of existence while the anonymous function lives on. There are no other references to the anonymous function; it persists only as a property of fh. Therefore I can avoid some behind-the-scenes bookkeeping overhead, as well as the weak–strong dance, by declaring fh as unowned in my capture list instead of weak. In real life, my own most frequent use of unowned is precisely in this context. Very often, the reference marked as unowned in the capture list will be self.

> Recall, from Chapter 2, that the compiler forces you to say self explicitly when you refer to a property or method of self within the function body of an escaping closure. Now we can see why: you are threatening to form a retain cycle involving self, and the compiler wants to make sure you realize that.

Above all, don't panic. Beginners may be tempted to backstop *all* their anonymous functions with [weak self]. That's wrong. Only a *stored* function can raise even the possibility of a retain cycle. Merely passing a function does *not* introduce such a possibility, especially if the function being passed will be called immediately. And even if a function *is* stored, if it is stored *elsewhere*, it might not imply a retain cycle. Always confirm that you actually *have* a retain cycle before concerning yourself with how to prevent it.

Consider the standard expression of an animation, which I discussed in Chapter 2:

```
UIView.animate(withDuration:0.4) {
    self.myButton3.frame.origin.y += 20
} completion: { _ in
    self.someMethod()
}
```

No one should be using [weak self] in these anonymous functions. There's no retain cycle, because the anonymous functions are not being retained by self (the view controller). Moreover, the view controller is *not* going to go out of existence while the animation is proceeding. In fact, capturing the view controller as self in the body of the anonymous function actually *prevents* it from going out of existence while the animation is proceeding — and that's *good!* Extending the lifetime of an object, such as self, long enough for an anonymous function to be called at some future time, is a *useful* thing to do. (In fact, it's so useful that Swift provides a way to do it *without* explicitly capturing the object in the anonymous function: there's a global function, withExtendedLifetime, for precisely that purpose.)

Memory management of protocol-typed references

Only a reference to an instance of a class type can be declared weak or unowned. A reference to an instance of a struct or enum type cannot be so declared, because its memory management doesn't work the same way (and is not subject to retain cycles). A reference that is declared as a protocol type, therefore, has a problem. A reference typed as a protocol that might be adopted by a struct or an enum cannot be declared weak or unowned. You can only declare a protocol-typed reference weak or unowned if the compiler knows that only a class can adopt it. You can assure the compiler of that by making the protocol a class protocol.

In this code, SecondViewControllerDelegate is a protocol that I've declared. This code won't compile unless SecondViewControllerDelegate is declared as a class protocol:

```
class SecondViewController : UIViewController {
    weak var delegate : SecondViewControllerDelegate?
    // ...
}
```

Here's the actual declaration of SecondViewControllerDelegate; it *is* declared as a class protocol, and that's why the preceding code is legal:

```
protocol SecondViewControllerDelegate : AnyObject {
    func accept(data:Any)
}
```

A protocol declared in Objective-C is implicitly marked as `@objc` and is a class protocol. This declaration from my real-life code is legal:

```
weak var delegate : WKScriptMessageHandler?
```

WKScriptMessageHandler is a protocol declared by Cocoa (in particular, by the Web Kit framework), so only a class can adopt WKScriptMessageHandler, and the compiler is satisfied that the `delegate` variable will be an instance of a class — and the reference can be treated as `weak`.

Exclusive Access to Value Types

Even value types can have memory management issues. In particular, a struct and its members might be directly accessed simultaneously, which could lead to unpredictable results. Fortunately, the compiler will usually stop you before such an issue can arise.

To illustrate, imagine that we have a Person struct with a settable `firstName` string property. Now let's write a function that takes both a Person and a string as `inout` parameters:

```
func change(_ p:inout Person, _ s:inout String) {}
```

So far so good; but now imagine calling that function with both a Person and that same Person's `firstName` as the parameters:

```
var p = Person(firstName: "Matt")
change(&p, &p.firstName) // compile error
```

The compiler will stop you from doing that, with this message: "Overlapping accesses to `p`, but modification requires exclusive access." The problem is that the single function `change` is being given direct access to the memory of both the struct itself and a member of that struct, simultaneously. We are threatening to alter the struct in some unpredictable way. This dangerous situation is forbidden; the compiler enforces *exclusive access* when a struct is being modified.

You may encounter that error message from the compiler under some surprising circumstances:

```
let c = UIColor.purple
var components = Array(repeating: CGFloat(0), count: 4)
c.getRed(&components[0], green: &components[1],
    blue: &components[2], alpha: &components[3]) // compile error
```

In that code, no exclusive access problem is evident to the untrained eye; you just have to take the compiler's word for it. One workaround is to take control of memory access yourself, silencing the compiler:

```
components.withUnsafeMutableBufferPointer { ptr -> () in
    c.getRed(&ptr[0], green: &ptr[1], blue: &ptr[2], alpha: &ptr[3])
}
```

It might be better to write a UIColor extension that assembles the array without any simultaneous memory access to multiple elements of the array:

```
extension UIColor {
    func getRedGreenBlueAlpha() -> [CGFloat] {
        var (r,g,b,a) = (CGFloat(0),CGFloat(0),CGFloat(0),CGFloat(0))
        self.getRed(&r, green: &g, blue: &b, alpha: &a)
        return [r,g,b,a]
    }
}
```

Sometimes the compiler can't see the issue coming, and you'll crash when the app runs instead:

```
var i = 0
func tweak(_ ii:inout Int) {
    print(i) // legal, but crash
}
tweak(&i)
```

In that code, `tweak` is accessing `i` in two ways simultaneously. On the one hand, `i` is being passed into `tweak` as an `inout` parameter. On the other hand, `tweak` is reaching out directly to `i` (which is in scope). Even though `tweak` never actually modifies `i` and never mentions its own `inout` parameter `ii`, that's a simultaneous access, and is forbidden by the runtime.

Miscellaneous Swift Language Features

This chapter is a miscellany, and this section is a miscellany within a miscellany. It surveys some "advanced" Swift language features that have been added relatively recently. The reason for postponing the discussion to this point is in part to avoid cluttering up the earlier exposition and in part to ensure that you know enough to understand the subject matter.

Synthesized Protocol Implementations

A few protocols built into the Swift standard library have the ability to synthesize implementations of their own requirements. Such a protocol can supply code behind the scenes so that an object that adopts the protocol will satisfy the protocol's requirements *automatically*. This feature was introduced in Swift 4.1.

Equatable

Equatable is one such protocol. That's good, because making your custom type adopt Equatable is often a really useful thing to do. Equatable adoption means that the == operator can be used to check whether two instances of this type are equal. The only requirement of the Equatable protocol is that you do, in fact, define == for your type. Using our Vial struct from earlier in this chapter, let's first do that manually:

```
struct Vial {
    var numberOfBacteria : Int
    init(_ n:Int) {
        self.numberOfBacteria = n
    }
}
extension Vial : Equatable {
    static func ==(lhs:Vial, rhs:Vial) -> Bool {
        return lhs.numberOfBacteria == rhs.numberOfBacteria
    }
}
```

Now that Vial is an Equatable, not only can it be compared with ==, but also lots of methods that need an Equatable parameter spring to life. For example, Vial becomes a candidate for use with methods such as `firstIndex(of:)`:

```
let v1 = Vial(500_000)
let v2 = Vial(400_000)
let arr = [v1,v2]
let ix = arr.firstIndex(of:v1) // Optional wrapping 0
```

What's more, the complementary inequality operator `!=` has sprung to life for Vial automatically! That's because it's already defined for *any* Equatable in terms of the == operator.

Our implementation of == for Vial was easy to write, but that's mostly because this struct has just one property. As soon as you have multiple properties, implementing == manually, though theoretically trivial, becomes tedious and possibly hard to maintain. This is just the kind of task computers are good at! You're likely going to want to define == in terms of the equality of all your type's properties simultaneously; well, Swift will implement == *automatically* in exactly that way. All we have to do is declare adoption of Equatable, like this:

```
struct Vial : Equatable {
    var numberOfBacteria : Int
    init(_ n:Int) {
        self.numberOfBacteria = n
    }
}
```

That code compiles, even though we now have no implementation of ==. That's because the implementation has been synthesized for us. Behind the scenes, two Vial objects are now equal just in case their `numberOfBacteria` are equal — which is exactly the implementation we supplied when we wrote the code explicitly.

For Equatable synthesis to operate, the following requirements must be met:

- Our object type is a struct or an enum.
- We have adopted Equatable, *not* in an extension.
- We have *not* supplied an implementation of the == operator.
- All of our struct's stored property types are themselves Equatable.

For an enum, the requirement here is that, if the enum has associated values, the types of those associated values must be Equatable. The synthesized == implementation will then say that two instances of our enum are equal if they are the same case and, if that case has an associated value, the associated values are equal for both instances. Recall that our MyError enum in Chapter 4 couldn't be used with the == operator until we explicitly declared it Equatable:

```
enum MyError : Equatable {
    case number(Int)
    case message(String)
    case fatal
}
```

(If an enum has *no* associated values, then it is already effectively Equatable and there is no need to adopt Equatable explicitly.)

If you don't like the synthesized implementation of == (perhaps because there is a property that you don't want involved in the definition of equality), all you have to do is write your own, explicitly. You lose the convenience of automatic synthesis, but you're no worse off than you were before automatic synthesis existed.

Let's say we have a Dog struct with a `name` property and a `license` property and a `color` property. And let's say we think two Dogs are equal just in case they have the same name and license; we don't care whether the colors are the same. Then we just have to write the implementation of == ourselves, omitting `color` from the calculation:

```
struct Dog : Equatable {
    let name : String
    let license : Int
    let color : UIColor
    static func ==(lhs:Dog,rhs:Dog) -> Bool {
        return lhs.name == rhs.name && lhs.license == rhs.license
    }
}
```

Hashable

Another protocol that performs synthesis of its own implementation is Hashable. Recall that a type must be Hashable to be used in a Set or as the key type of a Dictionary. A struct whose properties are all Hashable, or an enum whose associated values are all Hashable, can conform to Hashable merely by declaring that it adopts Hashable.

Hashable requires that its adopter, in addition to being Equatable, have a `hashValue` Int property; the idea is that two equal objects should have equal hash values. The implicit implementation combines the `hashValue` of the Hashable members to produce a `hashValue` for the object itself. That's good, because *you* would surely have no idea how to do that for yourself. Writing your own hash function is a very tricky business! Thanks to this feature, you don't have to.

But suppose you don't like the synthesized implementation of `hashValue`. Then you *will* have to calculate the `hashValue` yourself. Luckily, Swift gives you a way to do that. You ignore `hashValue`, and instead implement the `hash(into:)` method. There is then no need to implement `hashValue`, because it is autogenerated based on the result of `hash(into:)`. In this method, you are handed a Hasher object; you call `hash(into:)` with that object on every property that you want included in the hash calculation — and omit the ones you don't. For hashability to work properly, these should be the very same properties you've included in the Equatable calculation of ==.

So, for our Dog struct, we could write:

```
struct Dog : Hashable { // and therefore Equatable
    let name : String
    let license : Int
    let color : UIColor
    static func ==(lhs:Dog,rhs:Dog) -> Bool {
        return lhs.name == rhs.name && lhs.license == rhs.license
    }
```

```
        func hash(into hasher: inout Hasher) {
            name.hash(into:&hasher)
            license.hash(into:&hasher)
        }
    }
```

Comparable

Other protocols, alas, do not provide the same convenience. If we want our Vial struct to be Comparable, we must implement < explicitly. (And when we do, the other three comparison operators spring to life automatically as well.)

In Swift 5.3 and later, however, the Comparable protocol does provide a synthesized implementation — but only for an enum, and only if the enum has no raw value. The order of the cases is the order in which they are declared; if a case has an associated value, it must be of a Comparable type, so that if two instances of the enum have the same case, they can be ordered by their associated value:

```
enum Planet : Comparable {
    case mercury
    case venus
    case earth
    case mars
    case asteroid(String)
    case jupiter
}
// let's test it!
let test1 = Planet.mercury < Planet.venus // true
let test2 = Planet.jupiter > Planet.asteroid("Ceres") // true
let test3 = Planet.asteroid("Ceres") < Planet.asteroid("Vesta") // true
```

Key Paths

Key paths, a language feature introduced in Swift 4, effectively stand in relation to properties the way function references stand in relation to function calls — they are a way of storing a reference to a property without actually accessing the property.

Suppose we have a Person struct with a `firstName` String property and a `lastName` String property, and that we want to access one of these properties on a Person `p`, without knowing until runtime *which* property we are to access. We might write something like this:

```
var getFirstName : Bool = // ...
let name : String = {
    if getFirstName {
        return p.firstName
    } else {
        return p.lastName
    }
}()
```

That's not altogether atrocious, but it's hardly elegant. If we do the same sort of thing in several places, the same choice must somehow be repeated in each of those places — and the more choices there are, the more elaborate our code must be each time.

Key paths solve the problem by permitting us to encapsulate the *notion* of accessing a particular property of a type, such as Person's `firstName` or `lastName`, without actually *performing* the access. That notion is expressed as an instance; therefore, we can store it as a variable, or pass it as a function parameter. That instance then acts as a token that we can use to access the actual property on an actual instance of that type at some future time.

The literal notation for constructing a key path is:

```
\Type.property.property...
```

We start with a backslash. Then we have the name of a type, which may be omitted if the type can be inferred from the context. Then we have a dot followed by a property name — and this may be repeated if that property's type itself has a property that we will want to access, and so on.

In our simple case, we might store the notion of accessing a particular property as a key path variable, like this:

```
var prop = \Person.firstName
```

To perform the actual access, start with a reference to a particular instance and fetch its `keyPath:` subscript:

```
let whatname = p[keyPath:prop]
```

If `p` is a Person with a `firstName` of `"Matt"` and a `lastName` of `"Neuburg"`, then `whatname` is now `"Matt"`. Moreover, `whatname` is inferred to be a String, because the key path carries within itself information about the type of the property that it refers to (it is a generic).

Now imagine substituting a different key path for the value of `prop`:

```
var prop = \Person.firstName
// ... time passes ...
prop = \.lastName // inferred as \Person.lastName
```

That substitution is legal, because both `firstName` and `lastName` are Strings. Instantly, throughout our program, all occurrences of the Person `[keyPath:prop]` subscript take on a new meaning!

If the property referenced by a key path is writable and you have a writable object reference, then you can also set *into* the `keyPath:` subscript on that object, changing the value of the property:

```
p[keyPath:prop] = "Ethan"
```

Here's a practical example where my own code takes advantage of key paths. One of my apps is a version of a well-known game involving a deck of 81 cards, where every card has four attributes (color, number, shape, and fill), each of which can have three possible values. (81 is 3^4.) The player's job is to spot three cards obeying the following rule: each attribute has either the *same* value for *all* three cards or a *different* value for *each* of the three cards. The problem is to express that rule succinctly.

For the four card attributes, I use enums with Int raw values. This allows me to represent any attribute as a common type (Int). I express those raw values as computed properties, and I vend a list of all four computed properties as an array of key paths (`attributes`):

```
struct Card {
    let itsColor : Color
    let itsNumber : Number
    let itsShape : Shape
    let itsFill : Fill
    var itsColorRaw : Int { return itsColor.rawValue }
    var itsNumberRaw : Int { return itsNumber.rawValue }
    var itsShapeRaw : Int { return itsShape.rawValue }
    var itsFillRaw : Int { return itsFill.rawValue }
    static let attributes : [KeyPath<Card, Int>] = [
        \.itsColorRaw, \.itsNumberRaw, \.itsShapeRaw, \.itsFillRaw
    ]
    // ...
}
```

Now I can express the rule clearly and elegantly:

```
func isValidTriple(_ cards:[Card]) -> Bool {
    func evaluateOneAttribute(_ n:[Int]) -> Bool {
        let allSame = (n[0] == n[1]) && (n[1] == n[2])
        let allDiff = (n[0] != n[1]) && (n[1] != n[2]) && (n[2] != n[0])
        return allSame || allDiff
    }
    return Card.attributes.allSatisfy {attribute in
        evaluateOneAttribute(cards.map {$0[keyPath:attribute]}) // wow!
    }
}
```

Starting in Swift 5.2, a keypath literal can be used wherever a function is expected that takes that object's type as its sole parameter and produces that property's type as its result. That sentence is enough to make one's head swim, but the idea is quite simple. Consider our Person with a `firstName` and a `lastName` String property. If you have an array of Person, you can extract a list of first names using `map`, like this:

```
let names = arrayOfPersons.map {$0.firstName}
```

Well, now you can use a keypath literal instead:

```
let names = arrayOfPersons.map (\.firstName)
```

That's more than mere syntactic sugar; the point is that `\.firstName` acts as *the name of a function.* Here's a more surprising case in point:

```
let f : (Person) -> String = \Person.firstName
```

Instance as Function

Sometimes a type's primary job is to contain or represent a function. In those situations, it makes for cleaner code if we can treat an instance of that type *as* a function. The ability to do that was introduced in Swift 5.2.

Before I explain, I'll give an example. (This particular example comes more or less directly from Apple.) Imagine I have a struct Adder, whose job is to store a base value and add it to any addend we care to supply:

```
let add3 = Adder(3)
let sum = add3(4)
print(sum) // 7
```

That code, and indeed the entire concept of Adder, should remind you of the notion of a function as a factory for other functions (see "Function Returning Function" on page 54). In the second line, we treat `add3` as a function that takes an Int and returns another Int, namely the first Int with 3 added to it. But here's the thing: `add3` is *not* a function! It is an instance of a struct. We are treating an instance as a function.

The implementation, behind the scenes, is simple. If a type declares an instance method named `callAsFunction`, you can "call" an instance of that type as if it *were* a function: the "call" is routed to the `callAsFunction` method. Here's Adder:

```
struct Adder {
    let base: Int
    init(_ base:Int) {
        self.base = base
    }
    func callAsFunction(_ addend:Int) -> Int {
        return self.base + addend
    }
}
```

So when we create an Adder instance named `add3` and then say `add3(4)`, it is *exactly* the same as if we had said `add3.callAsFunction(4)`.

The function notation in `add3(4)` is mere syntactic sugar. But it's very nice syntactic sugar, because it reflects the truth more compactly. To be sure, we could have given Adder an `add` method that performs the addition, or a `makeAdder` method that produces a function that performs the addition:

```
struct Adder {
    let base: Int
    init(_ base:Int) {
        self.base = base
    }
    func add(_ addend:Int) -> Int {
        return self.base + addend
    }
    // or:
    func makeAdder() -> (Int) -> Int {
        return { addend in self.base + addend }
    }
}
```

Either of those would have been just fine. But the job of Adder is to act like a function, so rather than having it contain or produce that function, we can use callAsFunction to let an Adder effectively *be* that function.

You can give your type multiple callAsFunction overloads, distinguished in the usual way by parameter types, parameter labels, or both. In this way, a single instance can behave as if it were itself an overloaded function.

Dynamic Membership

Dynamic membership was introduced in Swift 4.2, with additional features in Swift 5. It allows you to do two things:

- Access a nonexistent property (a *dynamic* property) of an instance or type.
- Treat a reference to an instance as the name of a nonexistent function (a *dynamic* function).

Before I explain, I'll give an example. Imagine I have a class Flock that acts as a gatekeeper to a dictionary. I proceed to talk to a Flock instance like this:

```
let flock = Flock()
flock.chicken = "peep"
flock.partridge = "covey"
// flock's dictionary is now ["chicken": "peep", "partridge": "covey"]
if let s = flock.partridge {
    print(s) // covey
}
flock(remove:"partridge")
// flock's dictionary is now ["chicken": "peep"]
```

That's surprising, because Flock has no chicken property and no partridge property, and it is not the name of a function with a remove: parameter. So why am I able to talk that way?

Here's how the implementation works:

Dynamic properties

The type must be marked `@dynamicMemberLookup`, and must declare a `subscript` taking a single parameter that is either a string or a key path and has an external name `dynamicMember`. The subscript return type is up to you, and you can have multiple overloads distinguished by the return type. When other code accesses a nonexistent property of this type, the corresponding subscript function — the getter or, if there is one, the setter — is called with the name of the property as parameter.

Dynamic functions

The type must be marked `@dynamicCallable`, and must declare either or both of these methods, where the type T is up to you:

`dynamicallyCall(withArguments:[T])`

Other code uses an instance as a function name and calls it with a variadic parameter of type T. The variadic becomes an array. If T is Int, and the caller says `myObject(1,2)`, then this method is called with parameter `[1,2]`.

`dynamicallyCall(withKeywordArguments:KeyValuePairs<String, T>)`

Other code uses an instance as a function name and calls it with labeled arguments of type T. The label–value pairs become string–T pairs; if a label is missing, it becomes an empty string. If T is String, and the caller says `myObject(label:"one", "two")`, then this method is called with parameter `["label":"one", "":"two"]`.

So here's the implementation of Flock. Its dynamic properties are turned into dictionary keys, and it can be called as a function with a `"remove"` label to remove a key:

```
@dynamicMemberLookup
@dynamicCallable
class Flock {
    var d = [String:String]()
    subscript(dynamicMember s:String) -> String? {
        get { d[s] }
        set { d[s] = newValue }
    }
    func dynamicallyCall(withKeywordArguments kvs:KeyValuePairs<String, String>) {
        if kvs.count == 1 {
            if let (key,val) = kvs.first {
                if key == "remove" {
                    d[val] = nil
                }
            }
        }
    }
}
```

As originally conceived, dynamic membership is not intended for general use; its primary purpose is to prepare Swift for future interoperability with languages like Ruby, and perhaps to permit domain-specific languages. It is, after all, constitutionally opposed to the spirit of Swift: dynamism means that the compiler's validity checking is thrown away.

On the other hand, `@dynamicMemberLookup` with a subscript that takes a key path *is* Swift-like, because the validity of a given key path is checked at compile time. This feature is good particularly for forwarding messages to a wrapped value. Here's a Kennel struct that wraps a Dog:

```
struct Dog {
    let name : String
    func bark() { print("woof") }
}
@dynamicMemberLookup
struct Kennel {
    let dog : Dog
    subscript(dynamicMember kp:KeyPath<Dog,String>) -> String {
        self.dog[keyPath:kp]
    }
}
```

If `k` is a Kennel instance, we can now fetch `k.name` as a way of fetching `k.dog.name`. An attempt to say `k.nickname`, however, won't even compile; key paths maintain validity checking.

Property Wrappers

The idea of property wrappers, as you'll recall from Chapter 3, is that commonly used computed property getter and setter patterns can be encapsulated into a type with a `wrappedValue` computed property:

```
@propertyWrapper struct MyWrapper {
    // ...
    var wrappedValue : SomeType {
        get { /*...*/ }
        set { /*...*/ }
    }
}
```

You can then declare your computed property using the property wrapper type name as a custom attribute:

```
@MyWrapper var myProperty
```

The result is that, behind the scenes, a MyWrapper instance is created for you, and when your code gets or sets the value of `myProperty`, it is the getter or setter of this MyWrapper instance that is called.

In real life, your property wrapper's purpose will almost certainly be to act as a façade for access to a stored instance property, declared inside the property wrapper's type. This alone makes the property wrapper worth the price of admission, because now your main code is not cluttered with private stored properties acting as the backing store for computed properties; the stored properties are hidden in the property wrapper instances.

To introduce the syntax, I'll start with a property wrapper that acts as a façade for storage, and no more:

```
@propertyWrapper struct Facade<T> {
    private var _p : T
    init(wrappedValue:T) {
        self._p = wrappedValue
    }
    var wrappedValue : T {
        get {
            return self._p
        }
        set {
            self._p = newValue
        }
    }
}
```

That's a fairly normal-looking struct. It's a generic so that our property declaration can be of any type. The only special feature here is the initializer. The rule is that if you declare an `init(wrappedValue:)` initializer, it will be called automatically with the value to which the property is initialized:

```
@Facade var p = "test"
```

Here, the property wrapper's generic is resolved to String, and its `init(wrappedValue:)` initializer is called with parameter `"test"`. You can call other initializers instead, by treating the custom attribute name as a type name (which, of course, is just what it is) and putting parentheses after it. Suppose our property wrapper declares an initializer with no external parameter name:

```
init(_ val:T) {
    self._p = val
}
```

Then we can call that initializer by declaring our property like this:

```
@Facade("test") var p
```

More practically, here's a generalized version of the Clamped property wrapper from Chapter 3; we accept any Comparable type (so that `min` and `max` are available), and we accept (and require) initialization with a minimum and maximum to clamp to:

```
@propertyWrapper struct Clamped<T:Comparable> {
    private var _i : T
    private let min : T
    private let max : T
    init(wrappedValue: T, min:T, max:T) {
        self._i = wrappedValue
        self.min = min
        self.max = max
    }
    var wrappedValue : T {
        get {
            self._i
        }
        set {
            self._i = Swift.max(Swift.min(newValue,self.max),self.min)
        }
    }
}
```

And here's how to use it:

```
@Clamped(min:-7, max:7) var i = 0
@Clamped(wrappedValue:0, min:-7, max:7) var ii
```

Those declarations are equivalent, but the first is more natural; behind the scenes, the value with which we initialize the property is routed to the `wrappedValue:` parameter of the property wrapper initializer.

As I've said, when you declare a property with a property wrapper attribute, an actual property wrapper instance is generated. That instance is accessible to our code under the same name as the computed property with underscore (_) prefixed to it. In the case of our `@Facade` property `p`, if we set `p` to `"howdy"` and then say `print(_p)`, the console says `Facade<String>(_p: "howdy")`. You might use this feature for debugging, or to expose additional public members of the struct. This underscore-prefixed variable is declared with `var`, so you can even assign to it a property wrapper instance that you've initialized manually.

The property wrapper may also vend a value that can be referred to elsewhere by prefixing a dollar sign ($) to the property name. It does this by declaring a `projectedValue` property (this will usually be a computed property). You might use this to expose some useful secondary object.

That's how the SwiftUI `@State` attribute works. It is a property wrapper (one of several that are commonly used in SwiftUI). The State property wrapper struct has a `projectedValue` computed property whose getter returns the State struct's `binding` property. That property is a Binding, so when you use your `@State` property's `$` name, you get a Binding (see Chapter 14).

Be careful not to cause a name clash! If you declare a @Facade property p, you can't declare a property _p in the same scope, because that's the name of the synthesized property wrapper instance. (The same issue does not arise for the $ name, because a variable name can't start with $.)

New in Swift 5.5, property wrappers can be applied to function parameters. To see what I mean, let's start with a simple property wrapper that crashes if an assigned string is empty:

```
@propertyWrapper struct Nonempty {
    var wrappedValue : String = "" {
        didSet {
            if wrappedValue == "" {
                fatalError("Not allowed to assign empty string")
            }
        }
    }
}
```

Now we can have an instance property declared like this:

```
@Nonempty var name = "Matt"
```

If we assign "Neuburg" to that property, all is well. If we assign "" to that property, we crash.

You can imagine that it would be nice to extend this sort of validation to a function parameter: if an empty string is passed in as the argument to this parameter, we crash. Starting in Swift 5.5, you can do that:

```
func say(@Nonempty _ what: String) {
    print(what)
}
```

That compiles; but when we call say with an empty string, we *don't* crash. The reason is that, while setting a property-wrapped property sets the property wrapper's wrappedValue and therefore triggers the wrappedValue property's didSet observer, passing a value to a property-wrapped parameter *initializes* the property wrapper struct. So for this to work, we need to extend the same crashing behavior to the property wrapper struct's init(wrappedValue:) initializer:

```
@propertyWrapper struct Nonempty {
    var wrappedValue : String = "" {
        didSet {
            if wrappedValue == "" {
                fatalError("Not allowed to assign empty string")
            }
        }
    }
    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
```

```
            if wrappedValue == "" {
                fatalError("Not allowed to assign empty string")
            }
        }
    }
```

Inside the function that has a property-wrapped parameter, the local parameter value is just like a property-wrapped property. If you use its name, that's the `wrappedValue`. But if you use its name preceded by a dollar sign, that's the `projectedValue`.

To illustrate, imagine we have an extension on String defining a method, `rot13`, that applies ROT13 encipherment to the string. ROT13 encipherment advances each character through the alphabet by 13 characters, starting over at `"a"` if we come to `"z"`; it has the pleasant feature that applying ROT13 to an enciphered string deciphers it (because the alphabet has 26 letters). Here's a property wrapper that vends both the original string and the enciphered string:

```
@propertyWrapper struct Rot13 {
    var wrappedValue : String
    var projectedValue : String {
        wrappedValue.rot13()
    }
}
```

Now we can write a function that prints both the original string and the enciphered string:

```
func cipher(@Rot13 string: String) {
    print("the original string is", string)
    print("the enciphered string is", $string)
}
```

But we can go further. I'm imagining here that we're calling our `cipher` method like this:

```
cipher(string: "hello")
```

However, it is also permitted to call it like this:

```
cipher($string: "uryyb")
```

The dollar sign indicates that the argument here represents *the projected value*. In order to accommodate this, the property wrapper needs to have an `init(projectedValue:)` initializer. I'll rewrite our property wrapper struct so that it does have one:

```
@propertyWrapper struct Rot13 {
    let orig : String
    let ciph : String
    init(wrappedValue:String) {
        self.orig = wrappedValue
        self.ciph = wrappedValue.rot13()
    }
    init(projectedValue:String) {
        self.ciph = projectedValue
        self.orig = projectedValue.rot13()
    }
    var wrappedValue : String { self.orig }
    var projectedValue : String { self.ciph }
}
```

The result is that we can call the `cipher` function both ways, and we get *the same answer* each time:

```
cipher(string: "hello") // the original string is hello
                        // the enciphered string is uryyb
cipher($string: "uryyb") // the original string is hello
                         // the enciphered string is uryyb
```

That's because the presence or absence of the dollar sign in the call tells the property wrapper whether to call `init(projectedValue:)` or `init(wrappedValue:)`, and so it can distinguish whether the argument we're supplying represents the enciphered string or the original string.

Custom String Interpolation

Starting in Swift 5, string interpolation syntax can be customized. The example I suggested in Chapter 3 is an expression such as `"You have \(n, roman:true) widgets"`, where `n` is an Int; the idea is that, if `n` is 5, this would yield `"You have V widgets"`, expressing the Int in Roman numerals instead of Arabic notation. This would be an odd goal to accomplish through string interpolation, but it demonstrates the syntax; so let's implement that example.

To implement string interpolation, you need an adopter of the ExpressibleByStringLiteral protocol that also adopts the ExpressibleByStringInterpolation protocol. This might be some custom type of your own, but for this example we'll just use the built-in type that already adopts both of these protocols — String. `"You have \(n, roman:true) widgets"` is a String, and it already implements string interpolation.

That makes our job easy: all we have to do is customize String's already existing implementation of string interpolation. That implementation hinges on two methods, `appendInterpolation` and `appendLiteral`. The overall string is broken into segments, the interpolations (`appendInterpolation`) and the other parts (`appendLiteral`), and those methods are called; our job is to assemble them into a single object. String is already doing that job, and we don't need to modify the way it implements `appendLiteral`; we just need to modify the way it implements `appendInterpolation`.

To help us do that, there's another protocol, DefaultStringInterpolation. We extend this protocol to inject a version of `appendInterpolation` that takes our interpolated type — in this case, Int — along with any additional parameters. Our implementation should perform any necessary transformations to get a String, and then call the default appendInterpolation(_:).

Here's an Int method toRoman() that yields an Optional String (Int values of 0 or less will return nil, to indicate that they can't be expressed in Roman numerals):

```
extension Int {
    func toRoman() -> String? {
        guard self > 0 else { return nil }
        let rom = ["M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"]
        let ar = [1000,900,500,400,100,90,50,40,10,9,5,4,1]
        var result = ""
        var cur = self
        for (c, num) in zip(rom, ar) {
            let div = cur / num
            if (div > 0) {
                for _ in 0..<div { result += c }
                cur -= num * div
            }
        }
        return result
    }
}
```

Our interpolated type is Int, and we want to add one parameter, roman:, a Bool. So our extension of DefaultStringInterpolation will inject an implementation of appendInterpolation like this:

```
extension DefaultStringInterpolation {
    mutating func appendInterpolation(_ i: Int, roman: Bool) {
        if roman {
            if let r = i.toRoman() {
                self.appendInterpolation(r)
                return
            }
        }
        self.appendInterpolation(i)
    }
}
```

For a practical example of custom string interpolation in action, take a look at the Logger struct, introduced in Swift 5.3 as a Swifty wrapper for OSLog (Chapter 10). Its methods take string literals, but the parameter here is not a String; it is a special OSLogMessage struct that adopts ExpressibleByStringLiteral and ExpressibleByStringInterpolation and performs the interpolation at runtime, with custom interpolation parameters, by way of an OSLogInterpolation struct that adopts StringInterpolationProtocol.

Reverse Generics

Starting in Swift 5.1 and iOS 13, a function return type can be specified as a subtype of some supertype without stating *what* subtype it is. The syntax is the keyword `some` followed by the supertype. Suppose we have a protocol P and a struct S that adopts it:

```
protocol P {}
struct S : P {}
```

Then we can declare a function that returns a P adopter without specifying what adopter it is, by saying `some P`:

```
func f() -> some P {
    return S()
}
```

A computed property getter (or a subscript getter) is a function, so a computed property can also be declared a `some P`:

```
var p : some P { S() }
```

A stored variable cannot be declared a `some P` explicitly, but it can be a `some P` by inference:

```
var p2 = f() // now p2 is typed as some P
```

In those examples, the type *actually* being returned is S — and the compiler knows this by inference. But users of `f` and `p` know only that the type is an unspecified adopter of P. For this reason, this is sometimes called a *reverse generic* (or an *opaque type*). Instead of you declaring a placeholder and helping the compiler resolve it, the compiler has already resolved it and hides that resolution behind the placeholder.

A reverse generic `some P` is different from declaring the returned type as P. For one thing, in some situations you *can't* declare the returned type as P — as when P is a generic protocol. (A generic protocol, you remember, can be used only as a type constraint.) But more important, if `f` declared its return type as P, then it could return *any* P adopter on any occasion. That's not the case with `some P`; `f` returns an S and can return *only* an S. The compiler uses strict typing to enforce this — but the external declaration hides what exactly the compiler is enforcing. If there are two P adopters, S and S2, then a `some P` resolved to S has the same underlying type as another `some P` resolved to S, but it does not have the same type as a `some P` resolved to S2. This should remind you of a regular generic: an `Optional<String>` is a different type from an `Optional<Int>`.

As a result, a `some` type plays well with a generic in a way that an ordinary supertype would not. Suppose we have a Named protocol and an adopter of that protocol:

```
protocol Named {
    var name : String {get set}
}
struct Person : Named {
    var name : String
}
```

And suppose we have a generic function that uses this protocol as a type constraint:

```
func haveSameName<T:Named>(_ named1:T, _ named2:T) -> Bool {
    return named1.name == named2.name
}
```

Obviously, you can hand two Person objects to this generic function, because they resolve the generic placeholder T the same way:

```
let matt = Person(name: "Matt")
let ethan = Person(name: "Ethan")
let ok = haveSameName(matt,ethan) // fine
```

But you couldn't do that with two Named objects, because they might or might not represent the same adopting type:

```
let named1 : Named = Person(name: "Matt")
let named2 : Named = Person(name: "Ethan")
let ok = haveSameName(named1, named2) // compile error
```

Now suppose we have a function `namedMaker` that makes a `some Named` out of a String:

```
func namedMaker(_ name: String) -> some Named {
    return Person(name:name)
}
```

You, the user of `namedMaker`, do not necessarily know what Named adopter is produced by `namedMaker`. But you *do* know that every call to `namedMaker` produces the *same* type of Named adopter! The compiler knows this too, and it also knows *what* type of Named adopter `namedMaker` produces. Therefore you might write this, and the compiler will allow it:

```
let named1 = namedMaker("Matt")
let named2 = namedMaker("Ethan")
let ok = haveSameName(named1, named2) // fine
```

The main place you are likely to encounter `some` is when using SwiftUI or some other domain-specific language or library that wants to vend specific types while masking them under some common identity. (You are less likely to use `some` in your own code, though I suppose it could be useful for masking types in one area of your code from visibility in another area.)

For example, in SwiftUI a View's `body` is typed as `some View`. In this way you are shielded from the underlying complexities of compound view types, such as:

```
    VStack<TupleView<(Text, HStack<TupleView<(Image, Image)>>)>>
```

At the same time, the compiler knows that that *is* the type of this View's `body`, and therefore behind the scenes SwiftUI can reason coherently about this view.

Result Builders

Swift 5.1 introduced the ability to intercept a function passed to another function, capture its parameters and content, and transform them by way of a *result builder* to get a different function that is actually passed. This complex and highly specialized feature is aimed at enabling Swift to embrace domain-specific languages — the chief such language, at the moment, being SwiftUI. You can say this in SwiftUI:

```
VStack {
    Text("Hello")
    Text("World")
}
```

That doesn't look like legal Swift, so what's going on? Well, VStack is a type, and we are creating an instance of that type. We call an initializer to instantiate VStack. This particular initializer takes a function as its `content:` parameter. We could write this as `VStack(content:f)`, but instead we use trailing closure syntax; the curly braces embrace the body of an anonymous function. This body is now handed off to a result builder, which detects its structure and transforms it into something that *is* legal Swift (and that creates a VStack instance consisting of two Texts). The entire look and feel of SwiftUI code rests on this mechanism.

(If you're interested in more details about result builder syntax, there's an excellent WWDC 2021 video on the topic.)

Result

The Result enum expresses the notion of data-or-error in a single object. It's a generic, with a `success` case and a `failure` case, each carrying an associated value which is the data or error respectively.

One of Result's primary uses case is a *completion handler* function. The idea is that, because our code is asynchronous, we cannot directly return a result or throw an error; but we can *call* a function that has been handed to us as a parameter, and as we do, we can pass a parameter to that function. We might want to tell the function that we successfully obtained data, or that we failed because of an error. So we pass a Result object, which expresses both:

```
func doSomeNetworking(completion:@escaping (Result<Data,Error>) -> ()) {
    URLSession.shared.dataTask(with: myURL) { data, _, err in
        if let data = data {
            completion(.success(data))
        }
```

```
            if let err = err {
                completion(.failure(err))
            }
        }.resume()
    }
```

I'll have much more to say about that sort of code in Chapter 6, so if you don't know what a completion handler is or what "asynchronous" means, don't worry; you soon will, and for now you can just appreciate the syntax.

In fact, Result syntax is even cooler than that; it provides an initializer that lets us return the data or throw the error inside the initializer function, as a way of forming the Result object:

```
    func doSomeNetworking(completion:@escaping (Result<Data,Error>) -> ()) {
        URLSession.shared.dataTask(with: myURL) { data, _, err in
            let result = Result<Data,Error> {
                if let err = err {
                    throw err
                }
                return data!
            }
            completion(result)
        }.resume()
    }
```

And things work the same way in reverse at the caller's end, when we need to find out what this Result object consists of. We call the Result object's `get` method, which is a `throws` function. Either this gives us the data or else it throws:

```
    self.doSomeNetworking { result in
        do {
            let data = try result.get()
            // do something with the data
        } catch {
            // respond to the error
        }
    }
```

The advent of Swift structured concurrency (Chapter 6) will eventually obviate that particular use of Result, but there are many situations where it remains handy.