

objc ↑↓ Advanced Swift

Updated for Swift 5.6

By Chris Eidhof, Ole Begemann, Florian Kugler, and Ben Cohen

Version 5.0 (March 2022)

© 2022 Kugler und Eidhof GbR
All Rights Reserved

For more books, articles, and videos visit us at <https://wwwobjc.io>.

Email: mail@objc.io

Twitter: [@objcio](#)

1	Introduction	11
	Who Is This Book For?	12
	Themes	13
	Terminology	16
	Swift Style Guide	21
	Revision History	23
2	Built-In Collections	26
	Arrays	27
	Arrays and Mutability	27
	Array Indexing	29
	Transforming Arrays	31
	Array Slices	44
	Dictionaries	46
	Mutating Dictionaries	47
	Some Useful Dictionary Methods	47
	Hashable Requirement for Keys	49
	Sets	50
	Set Algebra	51
	Using Sets inside Closures	52
	Ranges	53
	Countable Ranges	54
	Partial Ranges	55
	Range Expressions	56
	RangeSet	57
	Recap	58
3	Optionals	59
	Sentinel Values	60
	Replacing Sentinel Values with Enums	62
	A Tour of Optional Techniques	64
	if let	64
	while let	65
	Doubly Nested Optionals	67
	if var and while var	69
	Scoping of Unwrapped Optionals	69
	Optional Chaining	73

The nil-Coalescing Operator	77
Using Optionals with String Interpolation	79
Optional map	80
Optional flatMap	82
Filtering Out nils with compactMap	84
Equating Optionals	85
Comparing Optionals	88
When to Force-Unwrap	88
Improving Force-Unwrap Error Messages	90
Asserting in Debug Builds	90
Implicitly Unwrapped Optionals	92
Implicit Optional Behavior	93
Recap	93

4 Functions	95
Overview	96
Flexibility through Functions	103
Functions as Data	105
Functions as Delegates	109
Delegates, Cocoa Style	109
Functions Instead of Delegates	110
inout Parameters and Mutating Methods	113
Nested Functions and inout	116
When & Doesn't Mean inout	116
Subscripts	117
Custom Subscripts	118
Advanced Subscripts	119
Autoclosures	120
The @escaping Annotation	123
withoutActuallyEscaping	124
Result Builders	126
Blocks and Expressions	127
Overloading Builder Methods	130
Conditionals	131
Loops	136
Other Build Methods	137
Unsupported Statements	138
Recap	138

5	Properties	139
	Change Observers	141
	Lazy Stored Properties	142
	Property Wrappers	144
	Usage	146
	Property Wrapper Ins and Outs	153
	Key Paths	154
	Key Paths Can Be Modeled with Functions	156
	Writable Key Paths	157
	The Key Path Hierarchy	157
	Key Paths Compared to Objective-C	158
	Future Directions	159
	Recap	159
6	Structs and Classes	160
	Value Types and Reference Types	161
	Mutation	165
	Mutating Methods	169
	inout Parameters	170
	Lifecycle	171
	Reference Cycles	172
	Closures and Reference Cycles	176
	Choosing between Unowned and Weak References	178
	Deciding between Structs and Classes	179
	Classes with Value Semantics	180
	Structs with Reference Semantics	181
	Copy-On-Write Optimization	183
	Copy-On-Write Tradeoffs	184
	Implementing Copy-On-Write	185
	willSet Defeats Copy-On-Write	190
	Recap	191
7	Enums	193
	Overview	194
	Enums Are Value Types	195
	Sum Types and Product Types	197
	Pattern Matching	199

Pattern Matching in Other Contexts	203
Designing with Enums	205
Switching Exhaustively	206
Making Illegal States Impossible	207
Modeling State with Enums	211
Choosing between Enums and Structs	215
Drawing Parallels between Enums and Protocols	218
Modeling Recursive Data Structures with Enums	220
Raw Values	224
The RawRepresentable Protocol	225
Manual RawRepresentable Conformance	225
RawRepresentable for Structs and Classes	227
Internal Representation of Raw Values	227
Enumerating Enum Cases	228
Manual Caseltable Conformance	229
Frozen and Non-Frozen Enums	230
Tips and Tricks	233
Recap	238
8 Strings	239
Unicode	240
Grapheme Clusters and Canonical Equivalence	243
Combining Marks	243
Emoji	246
Strings and Collections	249
Bidirectional, Not Random Access	250
Range-Replaceable, Not Mutable	251
String Indices	252
String Parsing	254
Substrings	256
StringProtocol	258
Code Unit Views	261
Index Sharing	263
Strings and Foundation	264
Other String-Related Foundation APIs	266
Ranges of Characters	269
CharacterSet	271
Unicode Properties	271
Internal Structure of String and Character	273
String Literals	274

String Interpolation	276
Custom String Descriptions	278
LosslessStringConvertible	280
Text Output Streams	281
Recap	283
9 Generics	285
Generic Types	286
Extending Generic Types	288
Generics vs. Any	290
Designing with Generics	292
Generics Are Statically Dispatched	295
How Generics Work	297
Generic Specialization	299
Recap	301
10 Protocols	303
Protocol Witnesses	306
Conditional Conformance	308
Protocol Inheritance	309
Designing with Protocols	310
Protocol Extensions	311
Customizing Protocol Extensions	312
Protocol Composition	315
Protocol Inheritance	316
Associated Types	317
Self	318
Example: State Restoration	318
Conditional Conformance with Associated Types	319
Retroactive Conformance	321
Existentials	322
Existentials vs. Generics	324
Existentials and Associated Types	324
Existentials Don't Conform to Protocols	327
Don't Use Existentials Prematurely	328
Opaque Types	329
Information Hiding	329
Rules for Opaque Types	333

Limitations of Opaque Types	335
Imagining the Standard Library with Opaque Types	337
Opaque Types Support Library Evolution	338
Opaque Types vs. Existentials	338
Type Erasers	339
Manual Type Erasure with Unlocked Existentials	341
Recap	343
11 Collection Protocols	344
Sequences	346
Iterators	347
Conforming to Sequence	350
Iterators and Value Semantics	352
Function-Based Iterators and Sequences	354
Single-Pass Sequences	355
The Relationship between Sequences and Iterators	357
Collections	358
A Custom Collection	359
Array Literals	364
Associated Types	365
Indices	367
Index Invalidation	368
Advancing Indices	369
Custom Collection Indices	370
Subsequences	373
Slices	376
Subsequences Share Indices with the Base Collection	377
Specialized Collections	379
BidirectionalCollection	379
RandomAccessCollection	381
MutableCollection	381
RangeReplaceableCollection	383
Lazy Sequences	385
Lazy Processing of Collections	387
Recap	388
12 Concurrency	390
Async/Await	392

How Asynchronous Functions Execute	394
Interfacing with Completion Handlers	397
Structured Concurrency	399
Tasks	399
<code>async let</code>	401
Task Groups	404
Sendable Types and Functions	407
Cancellation	409
Unstructured Concurrency	415
Actors	419
Resource Isolation	419
Reentrancy	422
Actor Performance	424
The Main Actor	425
Reasoning about Execution Contexts	427
Recap	428
13 Error Handling	429
Error Categories	430
The Result Type	432
Throwing and Catching	434
Typed and Untyped Errors	436
Non-Ignorable Errors	438
Error Conversions	440
Converting between throws and Optionals	440
Converting between throws and Result	441
Chaining Errors	442
Chaining throws	443
Chaining Result	443
Errors and Callbacks	445
Cleaning Up Using <code>defer</code>	448
Rethrowing	449
Bridging Errors to Objective-C	451
Recap	453
14 Encoding and Decoding	455
A Minimal Example	457
Automatic Conformance	457

Encoding	459
Decoding	460
Customizing the Encoded Format	461
The Encoding Process	463
Containers	463
How a Value Encodes Itself	466
The Synthesized Code	467
Coding Keys	467
The <code>encode(to:)</code> Method	468
The <code>init(from:)</code> Initializer	468
Raw-Representable and Enums	469
Manual Conformance	471
Custom Coding Keys	471
Custom <code>encode(to:)</code> and <code>init(from:)</code> Implementations	472
Common Coding Tasks	476
Making Types You Don't Own Codable	476
Making Classes Codable	480
Decoding Polymorphic Collections	483
Recap	484
15 Interoperability	486
Wrapping a C Library	487
Package Manager Setup	487
Wrapping the CommonMark Library	491
A Safer Interface	497
An Overview of Low-Level Types	503
Pointers	504
Closures as C Callbacks	508
Making It Generic	510
Recap	513
16 Final Words	515

Introduction

1

Advanced Swift is quite a bold title for a book, so perhaps we should start with what we mean by it.

Swift is a complex language — most programming languages are. But it hides that complexity well. You can get up and running developing apps in Swift without needing to know about generics or overloading or how copy-on-write works under the hood. You can certainly use Swift without ever calling into a C library or writing your own collection type. But after a while, we think you'll find it necessary to know about these things — whether to improve your code's performance, or to make it more elegant or expressive, or just to get certain things done.

Learning more about these features is what this book is about. We intend to answer many of the “How do I do this?” or “Why does Swift behave like that?” questions we've seen come up again and again. Hopefully, once you've read our book, you'll have gone from being aware of the basics of the language to knowing about many advanced features and having a much better understanding of how Swift works. Being familiar with the material presented is probably necessary, if not sufficient, for calling yourself an advanced Swift programmer.

Who Is This Book For?

This book targets experienced (though not necessarily expert) programmers, such as existing Apple-platform developers. It's also for those coming from other languages such as Java or C++ who want to bring their knowledge of Swift to the same level as that of their “go-to” language. Additionally, it's suitable for new programmers who started on Swift, have grown familiar with the basics, and are looking to take things to the next level.

The book isn't meant to be an introduction to Swift; it assumes you're familiar with the syntax and structure of the language. If you want some good, compact coverage of the basics of Swift, the best source is the official Swift book (available on docs.swift.org). If you're already a confident programmer, you could try reading our book and the official book in parallel.

This is also not a book about programming for macOS or iOS devices. Of course, since Swift is used a lot for development on Apple platforms, we've tried to include examples of practical use, but we think this book will be useful for non-Apple-platform programmers as well. The vast majority of the examples in the book should run unchanged on other operating systems. The ones that don't are either fundamentally tied to Apple's platforms (because they use iOS frameworks or rely on the Objective-C

runtime) or only require minimal changes. We can say from personal experience that Swift is a great language for writing server apps running on Linux, and the ecosystem and community have evolved over the past few years to make this a viable option.

Themes

This book is organized in such a way that each chapter covers one specific concept. There are in-depth chapters on some fundamental basic concepts like optionals and strings, along with some deeper dives into topics like C interoperability. But throughout the book, hopefully a few themes regarding Swift emerge:

Swift bridges multiple levels of abstraction. Swift is a high-level language — it allows you to write code similarly to Ruby and Python, with map and reduce, and to write your own higher-order functions easily. Swift also allows you to write fast code that compiles directly to native binaries with performance similar to code written in C.

What's exciting to us, and what's possibly the aspect of Swift we most admire, is that you're able to do both these things *at the same time*. Mapping a closure expression over an array compiles to the same assembly code as looping over a contiguous block of memory does.

However, there are some things you need to know about to make the most of this feature. For example, it will benefit you to have a strong grasp on how structs and classes differ, or an understanding of the difference between dynamic and static method dispatch (we'll cover these topics in depth later on). And if you ever need to drop to a lower level of abstraction and manipulate pointers directly, Swift lets you to do this as well.

Swift is a multi-paradigm language. To someone coming from another language, Swift can resemble everything they like about their language of choice. You can use Swift to write object-oriented code or pure functional code using immutable values, or you can write imperative C-like code using pointer arithmetic.

This is both a blessing and a curse. It's great in that you have a lot of tools available to you, and you aren't forced into writing code one way. But it also exposes you to the risk of mindlessly porting familiar Java or C or Objective-C patterns to Swift without looking for more idiomatic alternatives. Swift's type system offers many capabilities not available in those older languages, enabling (and often encouraging) new solutions for common tasks.

Erik Meijer, a well-known programming language expert, [tweeted the following in October 2015](#):

At this point, @SwiftLang is probably a better, and more valuable, vehicle for learning functional programming than Haskell.

Swift is a good introduction to a more functional style of programming through its use of generics, protocols, value types, and closures. It's even possible to use it to write operators that compose functions together. That said, most people in the Swift community seem to prefer a more imperative style while incorporating patterns that originated in functional programming. Swift's novel mutability model for value types, as well as its error handling model, are examples of the language "hiding" functional concepts behind a friendly imperative syntax.

Swift is very flexible. In the introduction to the book [*On Lisp*](#), Paul Graham writes that:

Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called bottom-up design—changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together.

Swift is very different from Lisp. But still, we feel like Swift also has this characteristic of encouraging "bottom-up" programming — of making it easy to write very general reusable building blocks that you then combine into larger features, which you then use to solve your actual problem. Swift is particularly good at making these building blocks feel like primitives — like part of the language. A good demonstration of this is that the many features you might think of as fundamental building blocks, like optionals or basic operators, are actually defined in a library — the Swift standard library — rather than directly in the language. Trailing closures enable you to extend the language with features that feel like they're built in.

Swift code can be compact and concise while still being clear. Swift lends itself to relatively terse code. There's an underlying goal here, and it isn't to save on typing. The idea is to get to the point quicker and to make code readable by dropping a lot of the

“ceremonial” boilerplate you often see in other languages that obscures rather than clarifies the meaning of the code.

For example, type inference removes the clutter of type declarations that are obvious from the context. Semicolons and parentheses that add little or no value are gone. Generics and protocol extensions encourage you to avoid repeating yourself by packaging common operations into reusable functions. The goal is to write code that's readable at a glance.

At first, this can be off-putting. If you've never before used functions like map, filter, and reduce, they might come across as more difficult to read than a simple for loop. But our hope is that this is a short learning curve and that the reward is code that is more “obviously correct” at first glance.

Swift tries to be as safe as is practical, until you tell it not to be. This is unlike languages such as C and C++ (where you can be unsafe easily just by forgetting to do something), or like Haskell or Java (which are sometimes safe whether or not you like it).

Eric Lippert, one of the principal designers of C#, wrote about his [10 regrets of C#](#), including the lesson that:

sometimes you need to implement features that are only for experts who are building infrastructure; those features should be clearly marked as dangerous—not invitingly similar to features from other languages.

Eric was specifically referring to C#'s finalizers, which are similar to C++ destructors. But unlike destructors, they run at a nondeterministic time (perhaps never) at the behest of the garbage collector (and on the garbage collector's thread). Swift, being reference counted, *does* execute a class's deinits deterministically (though the exact point when an object is freed may differ depending on compiler optimizations).

Swift embodies this sentiment in other ways. Undefined and unsafe behavior is avoided by default. For example, a variable can't be used until it's been initialized, and using out-of-bounds subscripts on an array will trap, as opposed to continuing with possibly garbage values.

There are a number of “unsafe” options available (such as the `unsafeBitCast` function, or the `UnsafeMutablePointer` type) for when you really need them. But with great power comes great undefined behavior. For example, you can write the following:

```
var someArray = [1, 2, 3]
let uhOh = someArray.withUnsafeBufferPointer { ptr in
    return ptr
}
// Later...
print(uhOh[10])
```

It'll compile, but who knows what it'll do. The `ptr` variable is only valid within the closure expression, and returning it to the caller is illegal. But there's nothing stopping you from letting it escape into the wild. However, you can't say nobody warned you.

Swift is an opinionated language. We as authors have strong opinions about the “right” way to write Swift. You’ll see many of them in this book, sometimes expressed as if they’re facts. But they’re just our opinions — feel free to disagree! Regardless of what you’re reading, the most important thing is to try things out for yourself, check how they behave, and decide how you feel about them.

Swift continues to evolve. Swift 1.0 was released in 2014. The early period of major yearly syntax changes is over, and Swift feels more and more “complete” with every version. But important areas of the language are still unfinished (concurrency, string APIs, the generics system) or haven’t been tackled yet (reflection, ownership).

Terminology

‘When I use a word,’ Humpty Dumpty said, in rather a scornful tone, ‘it means just what I choose it to mean — neither more nor less.’

— *Through the Looking Glass*, by Lewis Carroll

Programmers throw around terms of art a lot. To avoid confusion, what follows are some definitions of terms we use throughout this book. Where possible, we’re trying to adhere to the same usage as the official documentation, or sometimes a definition that’s been widely adopted by the Swift community. Many of these definitions are covered in more detail in later chapters, so don’t worry if not everything makes sense right away. If you’re already familiar with all of these terms, it’s still best to skim through to make sure your accepted meanings don’t differ from ours.

In Swift, we make the distinction between values, variables, references, and constants.

A **value** is immutable and forever — it never changes. For example, 1, true, and [1,2,3] are all values. These are examples of **literals**, but values can also be generated at runtime. The number you get when you square the number five is a value.

When we assign a value to a name using `var x = [1,2]`, we're creating a **variable** named `x` that holds the value [1,2]. By changing `x`, e.g. by performing `x.append(3)`, we didn't change the original value. Rather, we replaced the value that `x` holds with the new value, [1,2,3] — at least *logically*, if not in the actual implementation (which might actually just tack a new entry onto the back of some existing memory). We refer to this as **mutating** the variable.

We can declare **constant** variables (constants, for short) with `let` instead of `var`. Once a constant has been assigned a value, it can never be assigned a new value.

We also don't need to give a variable a value immediately. We can declare the variable first (`let x: Int`) and then later assign a value to it (`x = 1`). Swift, with its emphasis on safety, will check that all possible code paths lead to a variable being assigned a value before its value can be read. There's no concept of a variable having an as-yet-undefined value. Of course, if the variable was declared with `let`, it can only be assigned to once.

Structs and enums are **value types**. When you assign one struct variable to another, the two variables will then contain the same value. You can think of the contents as being copied, but it's more accurate to say that one variable was changed to contain the same value as the other.

A **reference** is a special kind of value: it's a value that “points to” some other location in memory. Because two references can refer to the same location, this introduces the possibility of the value stored at that location being mutated by two different parts of the program at once.

Classes and actors are **reference types**. You can't hold an instance of a class (which we might occasionally call an **object** — a term fraught with troublesome overloading!) directly in a variable. Instead, the variable holds a reference to the object and accesses it via that reference.

Reference types have **identity** — you can check if two variables are referring to the exact same object by using `==`. You can also check if two objects are equal, assuming `==` is implemented for the relevant type. Two objects with different identities can still be equal.

Value types don't have identity. You can't check if a particular variable holds the "same" number 2 as another. You can only check if they both contain the value 2. `==` is really asking: "Do both these variables hold the same reference as their value?" In programming language literature, `==` is sometimes called *structural equality*, and `===` is called *pointer equality* or *reference equality*.

Class and actor references aren't the only kind of reference in Swift. For example, there are also pointers, accessed through `withUnsafeMutablePointer` functions and the like. But classes and actors are the simplest reference types to use, in part because their reference-like nature is partially hidden from you by syntactic sugar, meaning you don't need to do any explicit "dereferencing" like you do with pointers in some other languages. (We'll cover the other kind of references in more detail in the [Interoperability chapter](#).)

A variable that holds a reference can be declared with `let` — that is, the reference is constant. This means that the variable can never be changed to refer to something else. But — and this is important — it *doesn't* mean that the object it *refers to* can't be changed. So when referring to a variable as a constant, be careful — it's only constant in what it points to; it doesn't mean what it points to is constant. (Note: If those last few sentences sound like doublespeak, don't worry, as we'll cover this again in the [Structs and Classes chapter](#).) Unfortunately, this means that when looking at a declaration of a variable with `let`, you can't tell at a glance whether or not what's being declared is completely immutable. Instead, you have to *know* whether it's holding a value type or a reference type.

When a value type is copied, it generally performs a *deep copy*, i.e. all values it contains are also copied recursively. This copy can occur eagerly (whenever a new variable is introduced) or lazily (whenever a variable gets mutated). Types that perform deep copies are said to have **value semantics**.

Here we hit another complication. If a struct contains reference types, the referenced objects won't automatically get copied upon assigning the struct to a new variable. Instead, only the references themselves get copied. These are called *shallow copies*.

For example, the `Data` struct in Foundation is a wrapper around a class that stores the actual bytes. This means the bytes themselves won't get copied when passing the array around. However, to preserve value semantics, the authors of the `Data` struct took extra steps to also perform a deep copy of the class instance whenever the `Data` struct is mutated. They do this efficiently using a technique called **copy-on-write**, which we'll explain in the [Structs and Classes chapter](#). For now, it's important to know that this

behavior isn't automatic. If you want your own types to use copy-on-write, you have to implement it yourself.

The collections in the standard library also wrap reference types and use copy-on-write to efficiently provide value semantics. However, if the elements in a collection are references (for example, an array containing objects), the objects won't get copied. Instead, only the references get copied. This means a Swift array only has value semantics if its elements have value semantics too.

Some classes are completely immutable — that is, they provide no methods for changing their internal state after they're created. This means that even though they're classes, they also have value semantics (because even if they're shared, they can never change). Be careful though — only final classes can be guaranteed not to be subclassed with added mutable state.

In Swift, functions are also values you can pass around. You can assign a function to a variable, have an array of functions, and call the function held in a variable. Functions that take other functions as arguments (such as `map`, which takes a function to transform every element of a sequence) or return functions are referred to as **higher-order functions**.

Functions don't have to be declared at the top level — you can declare a function within another function or in a `do` or other scope. Functions defined within an outer scope but passed out from it (say, as the returned value of a function), can "capture" local variables, in which case those local variables aren't destroyed when the local scope ends, and the function can hold state through them. This behavior is called "closing over" variables, and functions that do this are called **closures**.

Functions can be declared either with the `func` keyword or by using a shorthand `{}` syntax called a **closure expression**. Sometimes this gets shortened to "closures," but don't let it give you the impression that only closure expressions can be closures. Functions declared with the `func` keyword are also closures when they close over external variables.

Functions are held by reference. This means assigning a function that has captured state to another variable doesn't copy that state; it shares it, similar to object references. What's more is that when two closures close over the same local variable, they both share that variable, so they share state. This can be quite surprising, and we'll discuss this more in the [Functions](#) chapter.

Functions defined inside a class or protocol are **methods**, and they have an implicit self parameter. Sometimes we call functions that aren't methods **free functions**. This is to distinguish them from methods.

Similarly, variables or constants that are members of a type are called **properties**. Properties/variables that define a memory location for a value are also called **stored properties/variables**. In contrast, a **computed property/variable** has no storage. It's essentially another way to write a method/function that takes no arguments. Computed properties can be read-only or allow getting and setting a value.

A fully qualified function name in Swift includes not just the function's base name (the part before the parentheses), but also the argument labels. For example, the full name of the method for moving a collection index by a number of steps is `index(_:offsetBy:)`, indicating that this function takes two arguments (represented by the two colons), the first one of which has no label (represented by the underscore). We often omit the labels in the book if it's clear from the context what function we're referring to; the compiler allows you to do the same.

Free functions, along with methods called on structs and enums, are **statically dispatched**. This means the function that'll be called is known at compile time. It also means the compiler might be able to **inline** the function, i.e. not call the function at all, but instead replace it with the code the function would execute. The optimizer can also discard or simplify code that it can prove at compile time won't actually run.

Methods on classes or protocols might be **dynamically dispatched**. This means the compiler doesn't necessarily know at compile time which function will run. This dynamic behavior is done either by using vtables (similar to how Java and C++ dynamic dispatch work), or in the case of some @objc classes and protocols, by using selectors and `objc_msgSend` in the Objective-C runtime.

Subtyping and method **overriding** is one way of getting **polymorphic** behavior, i.e. behavior that varies depending on the types involved. A second way is function **overloading**, where a function is written multiple times for different types. (It's important not to mix up overriding and overloading, as they behave very differently.) A third way is via **generics**, where a function or method is written once to take any type that provides certain functions or methods, but the implementations of those functions can vary. Unlike method overriding, the results of function overloading and generics are known statically at compile time. We'll cover this more in the Generics chapter.

Swift code is organized into **modules**. To access a declaration from another module, you need to import that module. The standard library is a module named Swift, and it gets

imported automatically into every source file. The compiler builds all source files in the same module together as a single unit (when whole module optimization is enabled). This unlocks some important optimizations — such as generics specialization and inlining — for code that calls other code in the same module. Cross-module calls are generally less optimizable.

The Swift Package Manager uses the term **target** instead of module. Targets define modules, but they have a slightly broader meaning: a target can also consist of C/C++/Objective-C code and/or have non-code resources. A **product** is a unit of functionality the author wants to vend to external clients. Usually, a product contains a single target, but you can also group multiple targets in one product, and internal targets aren't part of any product. A **package** groups one or more products that are versioned together. Users of a package declare a **package dependency** on that package and then add one or more products of that package to their own targets as **target dependencies**.

Swift Style Guide

When writing this book, and when writing Swift code for our own projects, we try to stick to the following rules:

- For naming, clarity *at the point of use* is the most important consideration. Since APIs are used many more times than they're declared, their names should be optimized for how well they work at the call site. Familiarize yourself with the [Swift API Design Guidelines](#) and try to adhere to them in your own code.
- Clarity is often helped by conciseness, but brevity should never be a goal in and of itself.
- Design your APIs in a way that actively steers the user toward doing the “right thing” ([Xiaodi Wu](#)). Make it hard for programmers to shoot themselves in the foot.
- Always add documentation comments to functions — *especially* generic ones.
- Types start with **UpperCaseLetters**. Functions, variables, and enum cases start with **lowerCaseLetters**.
- Use type inference. Explicit but obvious types get in the way of readability.
- Don't use type inference in cases of ambiguity or when defining contracts (which is why, for example, funcs have an explicit return type).

- Default to structs unless you actually need a class-only feature or reference semantics.
- Mark classes as `final` unless you've explicitly designed them to be inheritable. If you want to use inheritance internally but not allow subclassing for external clients, mark a class `public` but not `open`.
- Use the trailing closure syntax, except when the closure is immediately followed by another opening brace (e.g. in an `if` condition).
- Use `guard` to exit functions early.
- Eschew force-unwraps and implicitly unwrapped optionals. They're occasionally useful, but needing them constantly is usually a sign something is wrong.
- Don't repeat yourself. If you find you've written a very similar piece of code more than a couple of times, extract it into a function. Consider making that function a protocol extension.
- Favor map and filter. But don't force it: use a `for` loop when it makes sense. The purpose of higher-order functions is to make code more readable. An obfuscated use of `reduce` when a simple `for` loop would be clearer defeats this purpose.
- Favor immutable variables: default to `let` unless you know you need mutation. But use mutation when it makes the code clearer or more efficient. Again, don't force it: a mutating method on a struct is often more idiomatic and efficient than returning a brand-new struct.
- Struct properties can generally be mutable because clients control mutability by way of making the struct variable `let` or `var`.
- Leave off `self`. when you don't need it. In closure expressions, the presence of `self`. is a clear signal that `self` is being captured by the closure.
- Instead of writing a free function, write an extension on a type or protocol (whenever you can). This helps with readability and discoverability through code completion.
- Don't hesitate to extend existing (standard library) types when it makes sense.

One final note about our code samples throughout the book: to save space and focus on the essentials, we usually omit import statements that would be required to make the code compile. If you try out the code yourself and the compiler tells you it doesn't recognize a particular symbol, try adding an `import Foundation` or `import UIKit` statement.

Revision History

Fifth Edition (March 2022)

- All chapters revised for Swift 5.6.
- New chapters:
 - [Concurrency](#)
 - [Properties](#) (was previously part of [Functions](#))
- New content:
 - [Built-In Collections:](#)
 - [RangeSet](#)
 - [Functions:](#)
 - [Result Builders](#)
 - [Properties:](#)
 - [Property Wrappers](#)
 - [Structs and Classes](#)
 - [willSet Defeats Copy-On-Write](#)
 - [Generics:](#)
 - [Generics Are Statically Dispatched](#)
 - [How Generics Work](#)
 - [Protocols:](#)
 - [Retroactive Conformance](#)
 - [Existentials](#)
 - [Opaque Types](#)
- Significant changes:
 - [Functions](#)
 - [Encoding and Decoding](#)

Fourth Edition (May 2019)

- All chapters revised for Swift 5.
- New chapter: [Enums](#)
- Rewritten chapters:
 - [Structs and Classes](#)
 - [Generics](#)
 - [Protocols](#)
- Significant changes and new content:
 - [Strings](#)
 - [Collection Protocols](#)
 - [Error Handling](#)
- Reordered the chapters; [Collection Protocols](#) has been moved further back in the book, resulting in a smoother learning curve for readers.
- Florian Kugler joined as a co-author.

Third Edition (October 2017)

- All chapters revised for Swift 4.
- New chapter: [Encoding and Decoding](#)
- Significant changes and new content:
 - [Built-In Collections](#)
 - [Collection Protocols](#)
 - [Functions](#) (new section on key paths)
 - [Strings](#) (more than 40 percent longer)
 - [Interoperability](#)
- Full text available as Xcode playgrounds.

Second Edition (September 2016)

- All chapters revised for Swift 3.

- Split the Collections chapter into Built-In Collections and Collection Protocols.
- Significant changes and new content throughout the book, especially in:
 - Collection Protocols
 - Functions
 - Generics
- Full text available as a Swift playground for iPad.
- Ole Begemann joined as a co-author.

First Edition (March 2016)

- Initial release (covering Swift 2.2).

Built-In Collections

2

Collections of elements are among the most important data types in any programming language. Good language support for different kinds of containers has a big impact on programmer productivity and happiness. Swift places special emphasis on sequences and collections — so much of the standard library is dedicated to this topic that we sometimes have the feeling it deals with little else. The resulting model is more extensible than what you may be used to from other languages, but it's also quite complex.

In this chapter, we'll take a look at the major collection types Swift ships with, with a focus on how to work with them effectively and idiomatically. In the [Collection Protocols](#) chapter later in the book, we'll climb up the abstraction ladder and see how the collection protocols in the standard library work.

Arrays

Arrays are the most common collections in Swift. An array is an ordered container of elements that all have the same type, and it provides random access to each element. As an example, to create an array of numbers, we can write the following:

```
// The Fibonacci numbers
let fibs = [0, 1, 1, 2, 3, 5]
```

Arrays and Mutability

If we try to modify the array defined above (by using `append(_:)`, for example), we get a compile error. This is because the array is defined as a constant, using `let`. In many cases, this is exactly the correct thing to do; it prevents us from accidentally changing the array. If we want the array to be a variable, we have to define it using `var`:

```
var mutableFibs = [0, 1, 1, 2, 3, 5]
```

Now we can easily append a single element or a sequence of elements:

```
mutableFibs.append(8)
mutableFibs.append(contentsOf: [13, 21])
mutableFibs // [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

There are a couple of benefits that come with making the distinction between var and let. Constants defined with let are easier to reason about because they're immutable. When you read a declaration like let fibs = ..., you know the value of fibs will never change — the immutability is enforced by the compiler. This helps greatly when reading through code. However, note that this is only true for types that have value semantics. A let variable containing a reference to a class instance guarantees that the *reference* will never change, i.e. you can't assign another object to that variable. However, the object the reference points to *can* change. We'll go into more detail on these differences in the [Structs and Classes](#) chapter.

Arrays, like all collection types in the standard library, have value semantics. When you assign an existing array to another variable, the array contents are copied over. For example, in the following code snippet, x is never modified:

```
var x = [1,2,3]
var y = x
y.append(4)
y // [1, 2, 3, 4]
x // [1, 2, 3]
```

The statement var y = x makes a copy of x, so appending 4 to y won't change x — the value of x will still be [1, 2, 3]. The same thing happens when you pass an array into a function; the function receives a local copy of the array, and any changes it makes don't affect the caller.

Compare this with the approach to mutability taken in many other languages, such as JavaScript, Java, and Objective-C (using NSArray from Foundation). Arrays in these languages have reference semantics: mutating an array through one variable implicitly changes what all other variables that reference the same array see, because all point to the same storage. Here's a JavaScript example:

```
// 'const' makes the *variables* a and b immutable.
const a = [1,2,3];
const b = a;
// But the object they *reference* is still mutable.
b.push(4);
console.log(b); // [ 1, 2, 3, 4 ]
console.log(a); // [ 1, 2, 3, 4 ]
```

The correct way to write this is to manually create a copy upon assignment:

```
const c = [1,2,3];
// Make an explicit copy.
const d = c.slice();
d.push(4);
console.log(d); // [ 1, 2, 3, 4 ]
console.log(c); // [ 1, 2, 3 ]
```

Forgetting this is easy and error-prone. For example, an object that returns an array from its internal state without making a copy may suddenly have its invariants broken when the caller mutates the array. Swift avoids this problem by giving collections value semantics.

Making a copy on every assignment could be a performance problem, but in practice, all collection types in the Swift standard library are implemented using a technique called copy-on-write, which makes sure the data is only copied when necessary. So in our example, `x` and `y` shared internal storage up until the point `y.append` was called. In the [Structs and Classes](#) chapter, we'll take a deeper look at value semantics — including how to implement copy-on-write for your own types:

Array Indexing

Swift arrays provide all the usual operations you'd expect, such as `isEmpty` and `count`. Arrays also allow for direct access of elements at a specific index through subscripting, like with `fibs[3]`. Keep in mind that you need to make sure the index is within bounds before getting an element via subscript. Fetch the element at index 3, and you'd better be sure the array has at least four elements in it. Otherwise, your program will trap, i.e. abort with a fatal error.

Swift has many ways to work with arrays without you ever needing to calculate an index:

- Want to iterate over the array? `for x in array`
- Want to iterate over all but the first element of an array? `for x in array.dropFirst()`
- Want to iterate over all but the last five elements? `for x in array.dropLast(5)`
- Want to number all the elements in an array?
`for (num, element) in array.enumerated()`
- Want to iterate over indices and elements together?
`for (index, element) in zip(array.indices, array)`

- Want to find the location of a specific element?
`if let idx = array.firstIndex { someMatchingLogic($0) }`
- Want to transform all the elements in an array?
`array.map { someTransformation($0) }`
- Want to fetch only the elements matching a specific criterion?
`array.filter { someCriteria($0) }`

Another sign that Swift wants to discourage you from doing index math is the absence of traditional C-style for loops in the language. Manually fiddling with indices is a rich seam of bugs to mine, so it's often best avoided.

But sometimes you do have to use an index. And with array indices, the expectation is that when you do, you'll have thought very carefully about the logic behind the index calculation. So to have to unwrap the value of a subscript operation is probably overkill — it means you don't trust your code. But chances are you do trust your code, so you'll probably resort to force-unwrapping the result, because you *know* the index must be valid. This is (a) annoying, and (b) a bad habit to get into. When force-unwrapping becomes routine, eventually you're going to slip up and force-unwrap something you don't mean to. So to prevent this habit from becoming routine, arrays don't give you the option.

While a subscripting operation that responds to an invalid index with a controlled crash could arguably be called *unsafe*, that's only one aspect of safety. Subscripting is totally safe in regard to *memory safety* — the standard library collections always perform bounds checks to prevent unauthorized memory access with an out-of-bounds index. In Swift, the term “safety” generally means memory safety and avoiding undefined behavior.

Other operations behave differently. The `first` and `last` properties return an optional value, which is `nil` if the array is empty. `first` is equivalent to `isEmpty ? nil : self[0]`. Similarly, the `removeLast` method will trap if you call it on an empty array, whereas `popLast` will only delete and return the last element if the array isn't empty, and otherwise it'll do nothing and return `nil`. Which one you'd want to use depends on your use case. When you're using the array as a stack, you'll probably always want to combine checking for empty and removing the last entry. On the other hand, if you already know whether or not the array is empty, dealing with the optional is fiddly.

We'll encounter these tradeoffs again later in this chapter when we talk about dictionaries. Additionally, there's an entire chapter dedicated to [Optionals](#).

Transforming Arrays

map

It's common to need to perform a transformation on every value in an array. Every programmer has written similar code hundreds of times: create a new array, loop over all the elements in an existing array, perform an operation on an element, and append the result of that operation to the new array. For example, the following code squares an array of integers:

```
var squared: [Int] = []
for fib in fibs {
    squared.append(fib * fib)
}
squared // [0, 1, 1, 4, 9, 25]
```

Swift arrays have a `map` method, which was adopted from the world of functional programming. Here's the exact same operation using `map`:

```
let squares = fibs.map { fib in fib * fib }
squares // [0, 1, 1, 4, 9, 25]
```

This version has three main advantages. It's shorter, of course. There's also less room for error. But more importantly, it's clearer: all the clutter has been removed. Once you're used to seeing and using `map` everywhere, it acts as a signal — you see `map`, and you immediately know what's happening: a function will be applied to every element and return a new array of the transformed elements.

The declaration of `squares` no longer needs to be made with `var`, because we aren't mutating it — it'll be delivered out of the `map` fully formed, so we can declare `squares` with `let`, if appropriate. And because the type of the contents can be inferred from the function passed to `map`, `squares` no longer needs to be explicitly typed.

The `map` method isn't hard to write — it's just a question of wrapping the boilerplate parts of the `for` loop into a generic function. Here's one possible implementation (though in Swift, it's actually an extension of the `Sequence` protocol, which we'll cover in the [Collection Protocols](#) chapter):

```
extension Array {
    func map<T>(_ transform: (Element) -> T) -> [T] {
```

```
var result: [T] = []
result.reserveCapacity(count)
for x in self {
    result.append(transform(x))
}
return result
}
```

Element is the generic placeholder for whatever type the array contains, and T is a new placeholder that represents the result of the element transformation. The map function itself doesn't care what Element and T are; they can be anything at all. The concrete type T of the transformed elements is defined by the return type of the transform function the caller passes to map. See the [Generics](#) chapter for details on generic parameters.

Really, the signature of this method should be `func map<T>(_ transform: (Element) throws -> T) rethrows -> [T]`, indicating that map will forward any error the transformation function might throw to the caller. We'll cover this in detail in the [Error Handling](#) chapter, but here, we left the error handling annotations out for the sake of simplicity. If you'd like, you can check out [the source code for Sequence.map](#) in the Swift repository on GitHub.

Parameterizing Behavior with Functions

Even if you're already familiar with map, take a moment and consider the map implementation. What makes it so general yet so useful?

map manages to separate out the boilerplate — which doesn't vary from call to call — from the functionality that always varies, i.e. the logic of how exactly to transform each element. It does this through a parameter the caller supplies: the transformation function.

This pattern of parameterizing behavior is found throughout the standard library. For example, there are more than a dozen separate methods on Array (and on other kinds of collections) that take a function to customize their behavior:

- **map** and **flatMap** — transform the elements
- **filter** — include only certain elements
- **allSatisfy** — test all elements for a condition
- **reduce** — fold the elements into an aggregate value
- **forEach** — visit each element
- **sort(by:)**, **sorted(by:)**, **lexicographicallyPrecedes(_:by:)**, and **partition(by:)** — reorder the elements
- **firstIndex(where:)**, **lastIndex(where:)**, **first(where:)**, **last(where:)**, and **contains(where:)** — does an element exist?
- **min(by:)** and **max(by:)** — find the minimum or maximum of all elements
- **elementsEqual(_:by:)** and **starts(with:by:)** — compare the elements to another array
- **split(whereSeparator:)** — break the elements up into multiple arrays
- **prefix(while:)** — take elements from the start as long as the condition holds true
- **drop(while:)** — drop elements until the condition ceases to be true, and then return the rest (similar to prefix, but this returns the inverse)
- **removeAll(where:)** — remove the elements matching the condition

The goal of all these functions is to get rid of the clutter of the uninteresting parts of the code, such as the creation of a new array and the for loop over the source data. Instead, the clutter is replaced with a single word that describes what's being done. This brings the important code – the logic the programmer wants to express – to the forefront.

Several of these functions have a default behavior. `sort` sorts elements in ascending order when they're comparable, unless you specify otherwise, and `contains` can take a value to check for, so long as the elements are equatable. These defaults help make the code even more readable. Ascending order sorting is natural, so the meaning of `array.sort()` is intuitive, and `array.firstIndex(of: "foo")` is clearer than `array.firstIndex { $0 == "foo" }`.

But in every instance, these are just shorthand for the common cases. Elements don't have to be comparable or equatable, and you don't have to compare the entire element — you can sort an array of people by their ages (`people.sort { $0.age < $1.age }`) or check if the array contains anyone underage (`people.contains { $0.age < 18 }`). You can also compare some transformation of the element. For example, an admittedly inefficient

case- and locale-insensitive sort could be performed via
people.sort { \$0.name.uppercased() < \$1.name.uppercased() }.

There are other functions of similar usefulness that would also take a function to specify their behaviors but which aren't in the standard library. You could easily define them yourself (and might like to try):

- **accumulate** — combine elements into an array of running values (like reduce, but returning an array of each interim combination)
- **count(where:)** — count the number of elements that match (this should have been added to Swift 5.0, but it was delayed due to a name clash with the count property; it will hopefully be reintroduced in a subsequent release)
- **indices(where:)** — return a list of indices matching a condition (similar to firstIndex(where:), but it doesn't stop on the first one)

If you find yourself iterating over an array to perform the same task or a similar one more than a couple of times in your code, consider writing a short extension to Array. For example, the following code splits an array into groups of adjacent equal elements:

```
let array: [Int] = [1, 2, 2, 2, 3, 4, 4]
var result: [[Int]] = array.isEmpty ? [] : [[array[0]]]
for (previous, current) in zip(array, array.dropFirst()) {
    if previous == current {
        result[result.endIndex-1].append(current)
    } else {
        result.append([current])
    }
}
result // [[1], [2, 2, 2], [3], [4, 4]]
```

We can formalize this algorithm by abstracting the code that loops over the array in pairs of adjacent elements from the logic that varies between applications (deciding where to split the array). We use a function argument to allow the caller to customize the latter:

```
extension Array {
    func split(where condition: (Element, Element) -> Bool) -> [[Element]] {
        var result: [[Element]] = self.isEmpty ? [] : [[self[0]]]
        for (previous, current) in zip(self, self.dropFirst()) {
            if condition(previous, current) {
                result.append([current])
```

```
    } else {
        result[result.endIndex-1].append(current)
    }
}
return result
}
}
```

This allows us to replace the for loop with the following:

```
let parts = array.split { $0 != $1 }
parts // [[1], [2, 2, 2], [3], [4, 4]]
```

Or, in the case of this particular condition, we can even write:

```
let parts2 = array.split(where: !=)
```

This has all the same benefits we described for map. The example with split(where:) is more readable than the example with the for loop; even though the for loop is simple, you still have to run the loop through in your head, which is a small mental tax. Using split(where:) introduces less chance of error (for example, accidentally forgetting the case of the array being empty), and it allows you to declare the result variable with let instead of var.

The split(where:) operation is also part of Apple's [Swift Algorithms package](#) under the name [chunked\(by:\)](#). Swift Algorithms is an open source library of commonly used collection operations with high-quality implementations. It's intended as a low-friction place for the Swift community to iterate on various algorithms without having to pass the high bar for additions to the standard library right away. Functionality that proves useful can then migrate to the standard library later.

We'll say more about [extending collections](#) and [using functions](#) later in the book.

Mutation and Stateful Closures

When iterating over an array, you could use `map` to perform side effects (e.g. inserting elements into some lookup table). We don't recommend doing this. Take a look at the following:

```
array.map { item in
    table.insert(item)
}
```

This hides the side effect (the mutation of the lookup table) in a construct that looks like a transformation of the array. If you ever see something like the above, it's a clear case for using a plain `for` loop instead of a function like `map`. The `forEach` method would also be more appropriate than `map` in this case, but it has its own issues, so we'll look at `forEach` a bit later.

Performing side effects is different than deliberately giving the closure *local* state, which is a particularly useful technique. In addition to being useful, it's what makes closures — functions that can capture and mutate variables outside their scope — such a powerful tool when combined with higher-order functions. For example, the `accumulate` function described above could be implemented with `map` and a stateful closure, like this:

```
extension Array {
    func accumulate<Result>(_ initialResult: Result,
                           _ nextPartialResult: (Result, Element) -> Result) -> [Result]
    {
        var running = initialResult
        return map { next in
            running = nextPartialResult(running, next)
            return running
        }
    }
}
```

This creates a temporary variable to store the running value and then uses `map` to create an array of the running values as the computation progresses:

```
[1,2,3,4].accumulate(0, +) // [1, 3, 6, 10]
```

filter

Another common operation is that of taking an array and creating a new array that only includes the elements that match a certain condition. The pattern of looping over an array and selecting the elements that match the given predicate is captured in the filter method:

```
let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
nums.filter { num in num % 2 == 0 } // [2, 4, 6, 8, 10]
```

We can use Swift's shorthand notation for arguments of a closure expression to make this even shorter. Instead of naming the num argument, we can write the above code like this:

```
nums.filter { $0 % 2 == 0 } // [2, 4, 6, 8, 10]
```

For very short closures, this can be more readable. If the closure is more complicated, it's almost always a better idea to name the arguments explicitly, as we've done before. It's really a matter of personal taste — choose whichever option is more readable at a glance. A good rule of thumb is this: if the closure fits neatly on one line, shorthand argument names are a good fit.

By combining map and filter, we can write a lot of operations on arrays without having to introduce a single intermediate variable. The resulting code will become shorter and easier to read. For example, to find all squares under 100 that are even, we could map the range 1.. <10 to square its members, and then we could filter out all odd numbers:

```
(1.. $<10$ ).map { $0 * $0 }.filter { $0 % 2 == 0 } // [4, 16, 36, 64]
```

The implementation of filter looks similar to map:

```
extension Array {
    func filter(_ isIncluded: (Element) -> Bool) -> [Element] {
        var result: [Element] = []
        for x in self where isIncluded(x) {
            result.append(x)
        }
        return result
    }
}
```

Two quick performance tips: firstly, note that chaining map and filter in this way creates an intermediate array (the result of the map operation) that's then thrown away. This isn't a problem for our small example, but for large collections or long chains, the extra allocations can negatively impact performance. We avoid these intermediate arrays by inserting `.lazy` at the start of the chain, thereby making all transformations *lazy*. Only at the end do we convert the lazy collection back into a regular array:

```
let lazyFilter = (1..10)
    .lazy.map { $0 * $0 }.filter { $0 % 2 == 0 }
let filtered = Array(lazyFilter) // [4, 16, 36, 64]
```

We'll talk more about lazy sequences in the [Collection Protocols](#) chapter.

Secondly, if you ever find yourself writing something like the following, stop!

```
bigArray.filter { someCondition }.count > 0
```

`filter` creates a brand-new array and processes every element in the array. But this is unnecessary. This code only needs to check if one element matches — in which case, `contains(where:)` will do the job:

```
bigArray.contains { someCondition }
```

This is much faster for two reasons: it doesn't create a whole new array of the filtered elements just to count them, and it exits early — as soon as it finds the first match. Generally, only ever use `filter` if you want all the results.

reduce

Both `map` and `filter` take an array and produce a new, modified array. Sometimes, however, you might want to combine all elements into a single new value. For example, to sum up all the elements, we could write the following code:

```
let fibs = [0, 1, 1, 2, 3, 5]
var total = 0
for num in fibs {
    total = total + num
}
total // 12
```

The `reduce` method takes this pattern and abstracts two parts: the initial value (in this case, zero), and the function for combining the intermediate value (total) and the element (num). Using `reduce`, we can write the same example like this:

```
let sum = fibs.reduce(0) { total, num in total + num } // 12
```

Operators are functions too, so we could've also written the same example like this:

```
fibs.reduce(0, +) // 12
```

The output type of `reduce` doesn't have to be the same as the element type. For example, if we want to convert a list of integers into a string, with each number followed by a comma and a space, we can do the following:

```
fibs.reduce("") { str, num in str + "\\"(num), " } // 0, 1, 1, 2, 3, 5,
```

Here's the implementation for `reduce`:

```
extension Array {
    func reduce<Result>(_ initialResult: Result,
        _ nextPartialResult: (Result, Element) -> Result)
    {
        var result = initialResult
        for x in self {
            result = nextPartialResult(result, x)
        }
        return result
    }
}
```

Another performance tip: `reduce` is very flexible, and it's common to see it used to build arrays and perform other operations. For example, you can implement `map` and `filter` using `reduce` only:

```
extension Array {
    func map2<T>(_ transform: (Element) -> T) -> [T] {
        return reduce([]) {
            $0 + [transform($1)]
        }
    }
}
```

```
func filter2(_ isIncluded: (Element) -> Bool) -> [Element] {
    return reduce([]) {
        isIncluded($1) ? $0 + [$1] : $0
    }
}
```

This is kind of beautiful and has the benefit of not needing those icky imperative for loops. But Swift isn't Haskell, and Swift arrays aren't lists. What's happening here is that on every execution of the combine function, a brand-new array is created by appending the transformed or included element to the previous one. This means that both these implementations are $O(n^2)$, not $O(n)$ — as the length of the array increases, the time these functions take increases quadratically.

There's a second version of `reduce` that has a slightly different type. The reducer function for combining an intermediate result and an element takes `Result` as an `inout` parameter:

```
public func reduce<Result>(into initialResult: Result,
    _ updateAccumulatingResult:
        (_ partialResult: inout Result, Element) throws -> ()
) rethrows -> Result
```

We discuss `inout` parameters in detail in the [Functions and Structs and Classes](#) chapters, but for now, think of the `inout` `Result` parameter as a mutable parameter: we can modify it within the function. This allows us to write `filter` in a much more efficient way:

```
extension Array {
    func filter3(_ isIncluded: (Element) -> Bool) -> [Element] {
        return reduce(into: []) { result, element in
            if isIncluded(element) {
                result.append(element)
            }
        }
    }
}
```

When using `inout`, the compiler doesn't have to create a new array each time, so this version of `filter` is again $O(n)$. When the call to `reduce(into:_:)` is inlined by the compiler, the generated code is often the same as when using a `for` loop.

A Flattening map

Sometimes, you want to map an array where the transformation function returns another array and not a single element.

For example, let's say we have a function, `extractLinks`, that takes a Markdown text and returns an array containing the URLs of all the links in the text. The function signature looks like this:

```
func extractLinks(markdown: String) -> [URL]
```

If we have a bunch of Markdown files and want to extract the links from all files into a single array, we could try to write something like `markdownFiles.map(extractLinks)`. But this returns an array of arrays containing the URLs: one array per file. Now you could just perform the `map`, get back an array of arrays, and then call `joined` to flatten the results into a single array:

```
let markdownFiles: [String] = // ...
let nestedLinks = markdownFiles.map(extractLinks)
let links = nestedLinks.joined()
```

The `flatMap` method combines these two operations — mapping and flattening — into a single step. So `markdownFiles.flatMap(extractLinks)` returns all the URLs in an array of Markdown files as a single array.

The signature for `flatMap` is almost identical to `map`, except its transformation function returns an array. The implementation uses `append(contentsOf:)` instead of `append(_:)` to flatten the result array:

```
extension Array {
    func flatMap<T>(_ transform: (Element) -> [T]) -> [T] {
        var result: [T] = []
        for x in self {
            result.append(contentsOf: transform(x))
        }
        return result
    }
}
```

Another great use case for flatMap is combining elements from different arrays. To get all possible pairs of the elements in two arrays, flatMap over one array and then map over the other in the inner transformation function:

```
let suits = ["♠", "♥", "♣", "♦"]
let ranks = ["J", "Q", "K", "A"]
let result = suits.flatMap { suit in
    ranks.map { rank in
        (suit, rank)
    }
}
/*
[("♠", "J"), ("♠", "Q"), ("♠", "K"), ("♠", "A"), ("♥", "J"), ("♥",
"Q"), ("♥", "K"), ("♥", "A"), ("♣", "J"), ("♣", "Q"), ("♣", "K"),
("♣", "A"), ("♦", "J"), ("♦", "Q"), ("♦", "K"), ("♦", "A")]
*/
```

Iteration Using forEach

The final operation we'd like to discuss is forEach. It works almost like a for loop: the passed-in function is executed once for each element in the sequence. And unlike map, forEach doesn't return anything — it's specifically meant for performing side effects. Let's start by mechanically replacing a loop with forEach:

```
for element in [1, 2, 3] {
    print(element)
}

[1, 2, 3].forEach { element in
    print(element)
}
```

This isn't a big win, but it can be handy if the action you want to perform is a single function call on each element in a collection. Passing a function name to forEach instead of passing a closure expression can lead to clearer and more concise code. For example, if you're writing a view controller on iOS and want to add an array of subviews to the main view, you can just use `theViews.forEach(view.addSubview)`.

However, there are some subtle differences between for loops and forEach. For instance, if a for loop has a return statement in it, rewriting it with forEach can significantly change the code's behavior. Consider the following example, which is written using a for loop with a where condition:

```
extension Array where Element: Equatable {  
    func firstIndex(of element: Element) -> Int? {  
        for idx in self.indices where self[idx] == element {  
            return idx  
        }  
        return nil  
    }  
}
```

We can't directly replicate the where clause in the forEach construct, so we might (incorrectly) rewrite this using filter:

```
extension Array where Element: Equatable {  
    func firstIndex_foreach(of element: Element) -> Int? {  
        self.indices.filter { idx in  
            self[idx] == element  
        }.forEach { idx in  
            return idx  
        }  
        return nil  
    }  
}
```

The return inside the forEach closure doesn't return out of the outer function; it only returns from the closure itself. In this particular case, we'd probably have found the bug, because the compiler generates a warning that the argument to the return statement is unused, but you shouldn't rely on it finding every such issue.

Also, consider the following example:

```
(1..  
10).forEach { number in  
    print(number)  
    if number > 2 { return }  
}
```

It's not immediately obvious that this prints out all the numbers in the input range. The return statement isn't breaking the loop; rather, it's returning from the closure, thus starting a new iteration of the loop inside `forEach`'s implementation.

In some situations, such as the `addSubview` example above, `forEach` can be nicer than a `for` loop. However, because of the non-obvious behavior of `return`, we recommend against most other uses of `forEach`. Just use a regular `for` loop instead.

Array Slices

In addition to accessing a single element of an array by subscript (e.g. `fibs[0]`), we can also retrieve a range of elements by subscript. For example, to get all but the first element of an array, we can do the following:

```
let slice = fibs[1...]
slice // [1, 1, 2, 3, 5]
type(of: slice) // ArraySlice<Int>
```

This gets us a slice of the array starting at the second element. The type of the result is `ArraySlice`, not `Array`. `ArraySlice` is a *view* on arrays. It's backed by the original array, yet it provides a view on just the slice. As a result, creating a slice is cheap — the array elements don't get copied.

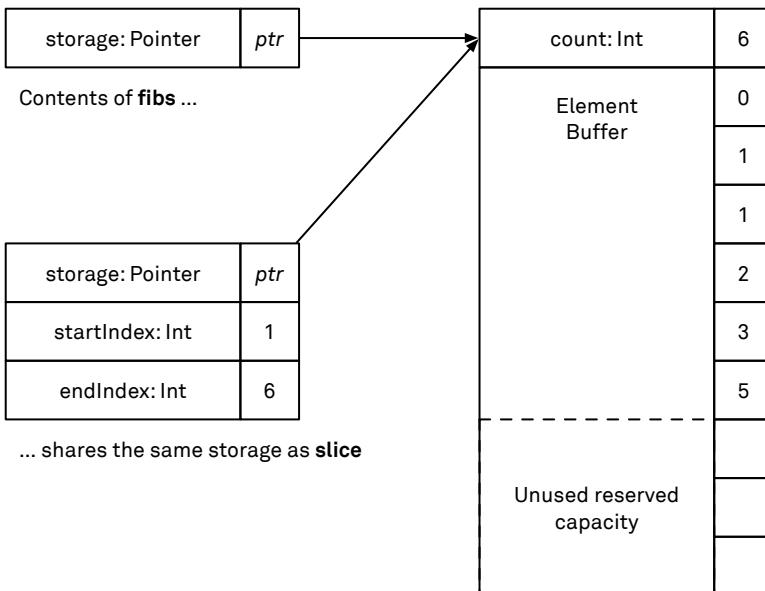


Figure 2.1: Array Slices

The `ArraySlice` type has the same methods defined as `Array` does (because both conform to the same protocols, most importantly `Collection`), so you can use a slice as if it were an array. If you do need to convert a slice into an array, you can just construct a new array out of the slice:

```
let newArray = Array(slice)
type(of: newArray) // Array<Int>
```

It's important to keep in mind that slices always use the same index to refer to a particular element as their base collection does. As a consequence, slice indices don't necessarily start at zero. For example, the first element of the `fibs[1...]` slice we created above is at index 1, and accessing `slice[0]` by mistake would crash our program with an out-of-bounds violation. If you work with indices, we recommend you always base your calculations on the `startIndex` and `endIndex` properties, even if you're dealing with a plain array where 0 and `count-1` would also do the trick. It's just too easy for this implicit assumption to break later on. We'll discuss this property of slices at length in the [Collection Protocols](#) chapter.

Dictionaries

Another key data structure is Dictionary. A dictionary contains unique keys with corresponding values. Retrieving a value by its key takes constant time on average, whereas searching an array for a particular element grows linearly with the array's size. Unlike arrays, dictionaries aren't ordered; the order in which key-value pairs are enumerated in a for loop is undefined.

In the following example, we use a dictionary as the model data for a fictional settings screen in a smartphone app. The screen consists of a list of settings, and each individual setting has a name (the keys in our dictionary) and a value. A value can be one of several data types, such as text, numbers, or Booleans. We use an enum with associated values to model this:

```
enum Setting {
    case text(String)
    case int(Int)
    case bool(Bool)
}

let defaultSettings: [String: Setting] = [
    "Airplane Mode": .bool(false),
    "Name": .text("My iPhone"),
]

defaultSettings["Name"] // Optional(setting.text("My iPhone"))
```

We use subscripting to get the value of a setting. Dictionary lookup always returns an *optional value* — when the specified key doesn't exist, it returns nil. Compare this with arrays, which respond to an out-of-bounds access by crashing the program.

Dictionaries also have a subscript that takes an *index* (in contrast to the commonly used subscript that takes a *key*) as part of their conformance to the Collection protocol. This subscript traps when called with an invalid index, just like the array subscript does, but it's almost never used (except implicitly in generic collection algorithms).

The rationale for this difference is that array indices and dictionary keys are used very differently. We've already seen that it's quite rare that you actually need to work with

array indices directly. And if you do, an array index is usually directly derived from the array in some way (e.g. from a range like `0..); thus, using an invalid index is a programmer error. On the other hand, it's very common for dictionary keys to come from some source other than the dictionary being subscripted.`

Unlike arrays, dictionaries are also sparse. The existence of the value under the key "name" doesn't tell you anything about whether or not the key "address" also exists.

Mutating Dictionaries

Just like arrays, dictionaries defined using `let` are immutable: no entries can be added, removed, or changed. And just like with arrays, we can define a mutable variant using `var`. To remove a value from a dictionary, we can either set it to `nil` using subscripting or call `removeValue(forKey:)`. The latter also returns either the deleted value, or `nil` if the key didn't exist. If we want to take an immutable dictionary and make changes to it, we have to make a copy:

```
var userSettings = defaultSettings
userSettings["Name"] = .text("Jared's iPhone")
userSettings["Do Not Disturb"] = .bool(true)
```

Note that, again, the value of `defaultSettings` didn't change. As with key removal, an alternative to updating via subscript is the `updateValue(_:forKey:)` method, which returns the previous value (if any):

```
let oldName = userSettings
    .updateValue(.text("Jane's iPhone"), forKey: "Name")
userSettings["Name"] // Optional(Setting.text("Jane's iPhone"))
oldName // Optional(Setting.text("Jared's iPhone"))
```

Some Useful Dictionary Methods

What if we wanted to combine the default settings dictionary with any custom settings the user has changed? Custom settings should override defaults, but the resulting dictionary should still include default values for any keys that haven't been customized. Essentially, we want to merge two dictionaries, where the dictionary that's being merged in overwrites duplicate keys.

Dictionary has a `merge(_:_:uniquingKeysWith:)` method, which takes the key-value pairs to be merged in and a function that specifies how to combine two values with the same key. We can use this to merge one dictionary into another, as shown in the following example:

```
var settings = defaultSettings
let overriddenSettings: [String:Setting] = ["Name":.text("Jane's iPhone")]
settings.merge(overriddenSettings, uniquingKeysWith: { $1 })
settings
// ["Name": Setting.text("Jane's iPhone"), "Airplane Mode": Setting.bool(false)]
```

In the example above, we used `{ $1 }` as the policy for combining two values. In other words, if a key exists in both `settings` and `overriddenSettings`, we use the value from `overriddenSettings`.

We can also construct a new dictionary out of a sequence of (Key,Value) pairs. If we guarantee that the keys are unique, we can use `Dictionary(uniqueKeysWithValues:)`. However, if we have a sequence where a key can exist multiple times, we need to provide a function to combine two values for the same keys, just like above. For example, to compute how often elements appear in a sequence, we can map over each element, combine it with a 1, and then create a dictionary out of the resulting element-frequency pairs. If we encounter two values for the same key (in other words, if we see the same element more than once), we simply add the frequencies using `+`:

```
extension Sequence where Element: Hashable {
    var frequencies: [Element:Int] {
        let frequencyPairs = self.map { ($0, 1) }
        return Dictionary(frequencyPairs, uniquingKeysWith: +)
    }
}
let frequencies = "hello".frequencies // ["e": 1, "h": 1, "l": 2, "o": 1]
frequencies.filter { $0.value > 1 } // ["l": 2]
```

Another useful method is a map over the dictionary's values. Because `Dictionary` is a `Sequence`, it already has a `map` method that produces an array. However, sometimes we want to keep the dictionary structure intact and only transform its values. The `mapValues` method does just this:

```
let settingsAsStrings = settings.mapValues { setting -> String in
    switch setting {
        case .text(let text): return text
```

```
case .int(let number): return String(number)
case .bool(let value): return String(value)
}
}
settingsAsStrings // ["Name": "Jane's iPhone", "Airplane Mode": "false"]
```

Hashable Requirement for Keys

Dictionaries are [hash tables](#). A dictionary assigns each key a position in its underlying storage array based on the key’s hash value. This is why Dictionary requires its Key type to conform to the Hashable protocol. All the basic data types in the standard library — including strings, integers, floating-point numbers, and Boolean values — already do. Additionally, many other types — like arrays, sets, and optionals — automatically become hashable if their elements are hashable.

To maintain their performance guarantees, hash tables require the types stored in them to provide a good hash function that doesn’t produce too many collisions. Writing a good hash function that distributes its inputs uniformly over the full integer range isn’t easy. Fortunately, we almost never have to do this ourselves. The compiler can generate the Hashable conformance in many cases, and even if that doesn’t work for a particular type, the standard library comes with a built-in hash function that custom types can hook into.

For structs and enums, Swift can automatically synthesize the Hashable conformance for us as long as those types are themselves composed of hashable types. If all stored properties of a struct are hashable, then the struct itself can be conformed to Hashable without us having to write a manual implementation. Similarly, enums that only contain hashable associated values can be conformed for free (enums without associated values even conform to Hashable without declaring this conformance explicitly). Not only does this save initial implementation work, but it also keeps the implementation up to date automatically as properties get added or removed.

If you can’t take advantage of the automatic Hashable synthesis (because either you’re writing a class or your custom struct has one or more stored properties that should be ignored for hashing purposes), you first need to make the type Equatable. Then you can implement the `hash(into:)` requirement of the Hashable protocol. This method receives a Hasher, which wraps a universal hash function and captures the state of the hash function as clients feed data into it. The hasher has a `combine` method that accepts any hashable value. You should feed all *essential components* of your type to the hasher by passing them to `combine` one by one. The essential components are the properties that

make up the substance of the type — you'll usually want to exclude transient properties that can be recreated lazily or that aren't visible to users of the type. For example, `Array` stores the capacity of its buffer — i.e. the maximum number of elements it can store before having to reallocate — internally. But two arrays that only differ in capacity should compare and hash the same — the capacity isn't an essential component of the `Array` type.

You should use the same essential components for equality checking, because the following important invariant must hold: two instances that are equal (as defined by your `==` implementation) *must* have the same hash value. The reverse isn't true: two instances with the same hash value don't necessarily compare equally. This makes sense, considering there's only a finite number of distinct hash values, while many hashable types (like strings) have essentially infinite cardinality.

The standard library's universal hash function uses a random seed as one of its inputs. In other words, the hash value of, say, the string "abc" will be different on each program execution. Random seeding is a security measure to protect against targeted hash-flooding denial of service attacks. Since `Dictionary` and `Set` iterate over their elements in the order they're stored in the hash table, and since this order is determined by the hash values, this means the same code will produce different iteration orders on each launch. If you need deterministic hashing (e.g. for tests), you can disable random seeding by setting the environment variable `SWIFT_DETERMINISTIC_HASHING=1`, but you shouldn't do this in production.

Finally, be extra careful when you use types that don't have value semantics (e.g. mutable objects) as dictionary keys. If you mutate an object after using it as a dictionary key in a way that changes its hash value and/or equality, you won't be able to find it again in the dictionary. The dictionary now stores the object in the wrong slot, effectively corrupting its internal storage. This isn't a problem with value types because the key in the dictionary doesn't share your copy's storage and therefore can't be mutated from the outside.

Sets

The third major collection type in the standard library is `Set`. A set is an unordered collection of elements, with each element appearing only once. You can essentially think of a set as a dictionary that only stores keys and no values. Like `Dictionary`, `Set` is

implemented with a hash table and has similar performance characteristics and requirements. Testing a value for membership in a set is a constant-time operation, and set elements must be Hashable, just like dictionary keys.

Use a set instead of an array when you need to test efficiently for membership (an $O(n)$ operation for arrays) and the order of the elements isn't important, or when you need to ensure that a collection contains no duplicates.

Set conforms to the `ExpressibleByArrayLiteral` protocol, which means we can initialize it with an array literal like this:

```
let naturals: Set = [1, 2, 3, 2]
naturals // [1, 2, 3]
naturals.contains(3) // true
naturals.contains(0) // false
```

Note that the number 2 appears only once in the set; the duplicate never even gets inserted.

Like all collections, sets support the common operations we've already seen: you can iterate over the elements in a `for` loop, map or filter them, and do all other sorts of things.

Set Algebra

As the name implies, Set is closely related to the mathematical concept of a set; it supports all common set operations you learned in math class. For example, we can subtract one set from another:

```
let iPods: Set = ["iPod touch", "iPod nano", "iPod mini",
  "iPod shuffle", "iPod classic"]
let discontinuedIPods: Set = ["iPod mini", "iPod classic",
  "iPod nano", "iPod shuffle"]
let currentIPods = iPods.subtracting(discontinuedIPods) // ["iPod touch"]
```

We can also form the intersection of two sets, i.e. find all elements that are in both:

```
let touchscreen: Set = ["iPhone", "iPad", "iPod touch", "iPod nano"]
let iPodsWithTouch = iPods.intersection(touchscreen)
// ["iPod nano", "iPod touch"]
```

Or, we can form the union of two sets, i.e. combine them into one (removing duplicates, of course):

```
var discontinued: Set = ["iBook", "PowerBook", "Power Mac"]
discontinued.formUnion(discontinued!Pods)
discontinued
/*
["iPod classic", "Power Mac", "iPod nano", "iPod shuffle", "iBook",
 "iPod mini", "PowerBook"]
*/
```

Here, we used the mutating variant `formUnion` to mutate the original set (which, as a result, must be declared with `var`). Almost all set operations have both non-mutating and mutating forms, and the latter have a form prefix. For even more set operations, check out the `SetAlgebra` protocol.

Using Sets inside Closures

Dictionaries and sets can be very handy data structures to use inside your functions, even when you're not exposing them to the caller. For example, if we want to write an extension on `Sequence` to retrieve all unique elements in the sequence, we could easily put the elements in a set and return its contents. However, that won't be *stable*: because a set has no defined order, the input elements might get reordered in the result. To fix this, we can write an extension that maintains the order by using an internal Set for bookkeeping:

```
extension Sequence where Element: Hashable {
    func unique() -> [Element] {
        var seen: Set<Element> = []
        return filter { element in
            if seen.contains(element) {
                return false
            } else {
                seen.insert(element)
                return true
            }
        }
    }
}
```

```
[1,2,3,12,1,3,4,5,6,4,6].unique() // [1, 2, 3, 12, 4, 5, 6]
```

The method above allows us to find all unique elements in a sequence while still maintaining the original order (with the constraint that the elements must be `Hashable`). Inside the closure we pass to `filter`, we refer to the `seen` variable that we defined outside the closure, thus maintaining state over multiple iterations of the closure. In the [Functions](#) chapter, we'll look at this technique in more detail.

Ranges

A range is an interval of values and is defined by its lower and upper bounds. You create ranges with the two range operators: `.. for half-open ranges that don't include their upper bound, and ... for closed ranges that include both bounds:`

```
// 0 to 9, 10 is not included.  
let singleDigitNumbers = 0..10  
Array(singleDigitNumbers) // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
// "z" is included.  
let lowercaseLetters = Character("a")...Character("z")
```

There are also prefix and postfix variants of these operators, which are used to express one-sided ranges:

```
let fromZero = 0...  
let upToZ = ...<Character("z")
```

There are five distinct concrete types that represent ranges, and each type captures different constraints on the value. The two most essential types are `Range` (a half-open range, created using `..) and ClosedRange (created using ...). Both have a generic Bound parameter: the only requirement is that Bound must be Comparable. For example, the lowercaseLetters expression above is of type ClosedRange<Character>.`

The most basic operation on a range is to test whether or not it contains certain elements:

```
singleDigitNumbers.contains(9) // true  
lowercaseLetters.overlaps("c".."f") // true
```

There are separate types for half-open and closed ranges, because both have a place:

- Only a **half-open range** can represent an **empty interval** (when the lower and upper bounds are equal, as in `5..<5`).
- Only a **closed range** can contain the **maximum value** its element type can represent (e.g. `0...Int.max`). A half-open range always requires at least one representable value that's greater than the highest value in the range.

Countable Ranges

Ranges seem like a natural fit to be sequences or collections. And you can indeed loop over a range of integers or treat it like a collection:

```
for i in 0..<10 {
    print("\\"(i)", terminator: " ")
} // 0 1 2 3 4 5 6 7 8 9

singleDigitNumbers.last // Optional(9)
```

But not all ranges can be used in this way. For example, the compiler won't let us iterate over a range of Characters:

```
// Error: Type 'Character' does not conform to protocol 'Strideable'.
for c in lowercaseLetters {
    ...
}
```

(The reason why iterating over characters isn't as straightforward as it would seem has to do with Unicode. We'll cover this issue at length in the [Strings](#) chapter.)

What's going on here? Range only conforms to the collection protocols *conditionally* if its element type conforms to Strideable (i.e. you can jump from one element to another by adding an offset) and if the stride steps themselves are integers:

```
extension Range: Sequence
    where Bound: Strideable, Bound.Stride: SignedInteger { /* ... */ }

extension Range: Collection, BidirectionalCollection,
    RandomAccessCollection
    where Bound: Strideable, Bound.Stride: SignedInteger { /* ... */ }
```

(We'll cover the Sequence, Collection, BidirectionalCollection, and RandomAccessCollection protocols in depth in the [Collection Protocols](#) chapter.)

In other words, a range must be *countable* for it to be iterated over. Valid bounds for countable ranges (i.e. that match the constraints) include integer and pointer types — but not floating-point types, because of the integer constraint on the type's Stride. If you need to iterate over consecutive floating-point values, you can use the `stride(from:to:by)` and `stride(from:through:by)` functions to create such a sequence.

Before conditional protocol conformance was introduced in Swift 4.1 and 4.2, the standard library included concrete types named `CountableRange` and `CountableClosedRange` for distinguishing countable from non-countable ranges. The names still exist as type aliases for backward compatibility. You can also use them as a shorthand for the mouthful of range-plus-constraints, as the comment in the standard library calls out:

```
// Note: this is not for compatibility only; it is considered useful
// shorthand.
public typealias CountableRange<Bound: Strideable> = Range<Bound>
    where Bound.Stride : SignedInteger
```

Partial Ranges

Partial ranges are constructed using `...` or `.. as a prefix or a postfix operator. These ranges are called partial because they're missing one of their bounds. For example, 0... describes a range that starts at zero and has no upper bound. There are three different kinds:`

```
let fromA: PartialRangeFrom<Character> = Character("a")...
let throughZ: PartialRangeThrough<Character> = ...Character("z")
let upto10: PartialRangeUpTo<Int> = ..<10
```

In the same way that `CountableRange` is a type alias for ranges with `Strideable` element types, `CountablePartialRangeFrom` is a type alias for `PartialRangeFrom`, but with tighter constraints.

When we iterate over a countable `PartialRangeFrom`, iteration starts with the `lowerBound` and repeatedly calls `advanced(by: 1)`. If you use such a range in a `for` loop, you must take care to add a break condition, lest you end up in an infinite loop (or crash when the counter overflows). `PartialRangeThrough` and `PartialRangeUpTo` cannot be iterated over,

regardless of if their element types are strideable or not, because they both lack a lower bound.

Range Expressions

All five range types conform to the RangeExpression protocol. The protocol itself is small enough to print in this book. It allows you to ask whether an element is contained within the range, and given a collection, it can compute a fully specified Range for you:

```
public protocol RangeExpression {
    associatedtype Bound: Comparable
    func contains(_ element: Bound) -> Bool
    func relative<C>(to collection: C) -> Range<Bound>
    where C: Collection, Self.Bound == C.Index
}
```

For partial ranges with a missing lower bound, the relative(to:) method adds the collection's startIndex as the lower bound. For partial ranges with a missing upper bound, the method will use the collection's endIndex. Partial ranges enable a very compact syntax for slicing collections:

```
let numbers = [1,2,3,4]
numbers[2...] // [3, 4]
numbers[..<1] // [1]
numbers[1...2] // [2, 3]
```

This works because the corresponding subscript declaration in the Collection protocol takes a RangeExpression rather than one of the five concrete range types. You can even omit both bounds to get a slice of the entire collection:

```
numbers[...] // [1, 2, 3, 4]
type(of: numbers[...]) // ArraySlice<Int>
```

(This is implemented as a special case in the standard library. Such an *unbounded range* isn't yet a valid RangeExpression, but it should become one eventually.)

If possible, try copying the standard library's approach and make your own functions take a RangeExpression rather than a concrete range type. It's not always possible because the protocol doesn't give you access to the range's bounds unless you're in the

context of a collection, but if it is, you'll give consumers of your APIs much more freedom to pass in any kind of range expression they like.

RangeSet

A RangeSet is a series of ranges of the same element type. Its main use case is to make working with noncontiguous sets of collection indices easy and efficient. You could, of course, use a Set<Index> for this task, but RangeSet is more storage-efficient because it can merge adjacent values into a single range. Say you have a table view with 1,000 elements and you want to use a set to manage the indices of the rows the user has selected. A Set<Int> needs to store up to 1,000 elements, depending on how many rows are selected. A RangeSet, on the other hand, stores contiguous ranges, so a selection of the first 500 rows in the table only takes two integers to store (the selection's lower and upper bounds).

RangeSet has a `ranges` property that exposes a collection interface over the ranges:

```
var indices = RangeSet(1..<5)
indices.insert(contentsOf: 11..<15)
/*show*/ Array(indices.ranges)
```

The ranges are always returned in ascending order, regardless of insertion order, and they never overlap or adjoin. If you instead prefer a collection of the individual elements, use `flatMap`:

```
/*show*/ Array(indices.ranges.flatMap { $0 })
/*show*/ let evenIndices = indices.ranges
    .flatMap { $0 }
    .filter { $0 % 2 == 0 }
```

RangeSet doesn't conform to `SetAlgebra`, but it does implement a subset of `SetAlgebra` operations for forming unions, intersections, etc.

The RangeSet type isn't yet part of the standard library, but it has gone through the Swift Evolution process as proposal [SE-0270](#). It's currently available as part of the [Standard Library Preview Package](#), which acts as a "proving ground" for new standard library additions to enable rapid adoption and allow for a real-world testing period where breaking changes can still be made.

Recap

In this chapter, we discussed a number of different collections: arrays, dictionaries, sets, and ranges. We also looked at both a number of operations that each of these collections have, and how to write powerful algorithms by composing those operations. We saw how Swift's built-in collections allow you to control mutability through `let` and `var`, and we also saw how to make sense of all the different Range types.

Compared to other languages, Swift's standard library offers relatively few general-purpose collection types. If you miss a particular data structure, take a look at the [Swift Collections package](#), where members of the standard library team and the rest of the Swift community work on high-quality implementations for common data structures, such as a double-ended queue and ordered variants of Set and Dictionary.

Strings are collections, too. We didn't discuss them here because they get [a dedicated chapter](#).

We'll revisit this chapter's topic in [Collection Protocols](#), where we'll discuss the protocols on which Swift's collections are built in depth.

Optionals

3

Sentinel Values

An extremely common pattern in programming is to have an operation that may or may not return a value.

Perhaps not returning a value is an expected outcome when you've reached the end of a file you were reading, as in the following C snippet:

```
int ch;
while ((ch = getchar()) != EOF) {
    printf("Read character %c\n", ch);
}
printf("Reached end-of-file\n");
```

EOF is just a #define for -1. As long as there are more characters in the file, getchar returns them. But if the end of the file is reached, getchar returns -1.

Or, perhaps returning no value means “not found,” as in this bit of C++:

```
auto vec = {1, 2, 3};
auto iterator = std::find(vec.begin(), vec.end(), someValue);
if (iterator != vec.end()) {
    std::cout << "vec contains " << *iterator << std::endl;
}
```

Here, `vec.end()` is the iterator “one past the end” of the container; it's a special iterator you can check against the container's end but that you mustn't ever actually use to access a value — similar to a collection's `endIndex` in Swift. The `find` function uses it to indicate that no such value is present in the container.

Or, maybe the value can't be returned because something went wrong during the function's processing. Probably the most notorious example is that of the null pointer. This innocuous-looking piece of Java code will likely throw a `NullPointerException`:

```
int i = Integer.getInteger("123")
```

It happens that `Integer.getInteger` doesn't parse strings into integers, but rather gets the integer value of a system property named “123.” This property probably doesn't exist, in which case `getInteger` returns `null`. When the `null` then gets unboxed into an `int`, Java throws an exception.

Or, take this example in Objective-C:

```
[[NSString alloc] initWithContentsOfURL:url  
encoding:NSUTF8StringEncoding error:&error];
```

This initializer might return `nil`, in which case — and only then — the error pointer should be checked. There's no guarantee the error pointer is valid if the initializer returns non-`nil`.

In all of the above examples, the function returns a special “magic” value to indicate it hasn't returned a real value. Magic values like these are called “sentinel values.”

But this approach is problematic. The returned result looks and feels like a real value. An `int` of `-1` is still a valid integer, but you don't ever want to print it out. `vec.end()` is an iterator, but the results are undefined if you try to use it. And everyone loves seeing a stack dump when your Java program throws a `NullPointerException`.

Unlike Java, Objective-C allows sending messages to `nil`. This is “safe” insofar as the runtime guarantees that the return value from a message to `nil` will always be the equivalent of zero, i.e. `nil` for object return types, `0` for numeric types, and so on. If the message returns a struct, it'll have all its members initialized to zero. With this in mind, consider the following snippet for finding a substring:

```
NSString *someString = ...  
if ([someString rangeOfString:@"Swift"].location != NSNotFound) {  
    NSLog(@"Someone mentioned Swift!");  
}
```

If `someString` is `nil`, whether accidentally or on purpose, the `rangeOfString:` message will return a zeroed `NSRange`. Hence, its `.location` will be zero, and the inequality comparison against `NSNotFound` (which is defined as `NSIntegerMax`) will succeed. Therefore, the body of the `if` statement will be executed when it shouldn't be.

Null references cause so many problems that Tony Hoare, credited with their creation in 1965, calls them his “billion-dollar mistake”:

At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed

automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Another problem with sentinel values is that using them correctly requires prior knowledge. Sometimes there's an idiom that's widely followed in a community, as with the C++ `end` iterator or the error handling conventions in Objective-C. If such rules don't exist or you're not aware of them, you have to refer to the documentation. Moreover, there's no way for the function to indicate it *can't* fail. If a call returns a pointer, that pointer might never be `nil`. But there's no way to tell except by reading the documentation, and even then, the documentation might be wrong.

Replacing Sentinel Values with Enums

Of course, every good programmer knows magic numbers are bad. Most languages support some kind of enumeration type, which is a safer way of representing a set of discrete possible values for a type.

Swift takes enumerations further with the concept of “associated values.” These are enumeration values that can also have another value associated with them. We'll look at enumerations in detail in the [Enums](#) chapter. For now, it's enough to know that `Optional` is defined as an enumeration:

```
enum Optional<Wrapped> {  
    case none  
    case some(Wrapped)  
}
```

The only way to retrieve an enum's associated value is through [pattern matching](#) — for example, in a `switch` or an `if case let` statement. Encoding the “missing value” state in the type system makes code more expressive because callers of an API see at a glance whether or not they have to handle this case. And unlike with a sentinel value, you can't accidentally use the value embedded in an `Optional` without explicitly checking and unpacking it.

So instead of returning a sentinel value, the Swift equivalent of `find` — called `firstIndex(of:)` — returns an `Optional<Index>` with an implementation somewhat similar to this:

```
extension Collection where Element: Equatable {  
    func firstIndex(of element: Element) -> Optional<Index> {  
        var idx = startIndex  
        while idx != endIndex {  
            if self[idx] == element {  
                return .some(idx)  
            }  
            formIndex(after: &idx)  
        }  
        // Not found, return .none.  
        return .none  
    }  
}
```

Since optionals are fundamental to Swift, there's lots of syntax support to tidy this up: `Optional<Index>` can be written `Index?`; optionals conform to `ExpressibleByNilLiteral` so that you can write `nil` instead of `.none`; and non-optional values (like `idx`) are automatically “promoted” to optionals where needed so that you can write `return idx` instead of `return .some(idx)`.

The syntactic sugar is effective in disguising the true nature of the `Optional` type. It's worth remembering that there's nothing magical about it; it's just a normal enum, and if it didn't exist, you could define it yourself.

Now there's no way a user could mistakenly use the value before checking if it's valid:

```
var array = ["one", "two", "three"]  
let idx = array.firstIndex(of: "four") // returns Optional<Int>.  
// Compile-time error: remove(at:) takes an Int, not an Optional<Int>.  
array.remove(at: idx)
```

Instead, you're forced to “unwrap” the optional to get at the index within, assuming you didn't get `none` back:

```
var array = ["one", "two", "three"]  
switch array.firstIndex(of: "four") {  
    case .some(let idx):  
        array.remove(at: idx)
```

```
case .none:  
    break // Do nothing.  
}
```

This switch statement writes out the enumeration syntax for optionals longhand, including unpacking the associated value in the `some` case. This is great for safety, but it's not very pleasant to read or write. A more succinct alternative is to use the `? pattern` suffix syntax to match a `some` optional value, and you can use the `nil` literal to match `none`:

```
switch array.firstIndex(of: "four") {  
case let idx?: // Equivalent to .some(let idx)  
    array.remove(at: idx)  
case nil:  
    break // Do nothing.  
}
```

But this is still clunky. Let's take a look at all the other ways you can make your optional processing short and clear, depending on your use case.

A Tour of Optional Techniques

Optionals have a lot of extra support built into the language. Some of the examples below might look very simple if you've been writing Swift for a while, but it's important to make sure you know all of these concepts well, as we'll be using them again and again throughout the book.

if let

Optional binding with `if let` is just a short step away from the `switch` statement above. An `if let` statement checks if the optional value is non-`nil`, and if so, it unwraps the optional. The type of `idx` is `Int` (non-optional), and `idx` is only available within the scope of the `if let` statement:

```
var array = ["one", "two", "three", "four"]  
if let idx = array.firstIndex(of: "four") {  
    array.remove(at: idx)  
}
```

You can use Boolean clauses together with if let as well. So suppose you didn't want to remove the element if it happened to be the first one in the array:

```
if let idx = array.firstIndex(of: "four"), idx != array.startIndex {  
    array.remove(at: idx)  
}
```

You can also bind multiple values in the same if statement. What's more is that later entries can rely on the earlier ones being successfully unwrapped. This is very useful when you want to make multiple calls to functions that return optionals themselves. For example, the URL and UIImage initializers in the following example are all “failable” — that is, they can return nil — if your URL is malformed or if the data isn't an image. The Data initializer can throw an error, and by using try?, you can convert it into an optional as well. All three can be chained together, like this:

```
let urlString = "https://www.objc.io/logo.png"  
if let url = URL(string: urlString),  
    let data = try? Data(contentsOf: url),  
    let image = UIImage(data: data)  
{  
    let view = UIImageView(image: image)  
    PlaygroundPage.current.liveView = view  
}
```

The synchronous Data(contentsOfURL:) initializer will block its thread for the duration of the download. It's fine for a quick example, but it's not recommended for production code. Always use an asynchronous networking API, such as URLSession.data(from:delegate:).

You can freely mix and match optional bindings, Boolean clauses, and case let bindings within the same if statement.

while let

Very similar to the if let statement is while let — a loop that terminates when its condition returns nil.

The standard library's `readLine` function returns an optional string from the [standard input](#). Once the end of input is reached, it returns `nil`. So to implement a very basic equivalent of the Unix `cat` command, you use `while let`:

```
while let line = readLine() {
    print(line)
}
```

Just like in `if let` statements, you can always add a Boolean clause to your optional binding. So if you want to terminate this loop on either EOF or a blank line, add a clause to detect an empty string. Note that once the condition is `false`, the loop is terminated (you might mistakenly think the Boolean condition functions like a filter):

```
while let line = readLine(), !line.isEmpty {
    print(line)
}
```

As we'll see in the [Collection Protocols](#) chapter, the `for x in seq` loop requires `seq` to conform to `Sequence`. This protocol provides a `makeIterator` method that returns an iterator, which in turn has a `next` method. `next` returns values until the sequence is exhausted, and then it returns `nil`. `while let` is ideal for this:

```
let array = [1, 2, 3]
var iterator = array.makeIterator()
while let i = iterator.next() {
    print(i, terminator: " ")
}
// 1 2 3
```

So given that `for` loops are really just `while` loops, it's not surprising that they also support Boolean clauses, albeit with a `where` keyword:

```
for i in 0..<10 where i % 2 == 0 {
    print(i, terminator: " ")
}
// 0 2 4 6 8
```

Note that the `where` clause above doesn't work like the Boolean clause in a `while` loop. In a `while` loop, iteration stops once the value is `false`, whereas in a `for` loop, it functions like `filter`. If we rewrite the above `for` loop using `while`, it looks like this:

```
var iterator2 = (0..<10).makeIterator()
while let i = iterator2.next() {
```

```
guard i % 2 == 0 else { continue }
print(i)
}
```

Doubly Nested Optionals

This is a good time to point out that the type an optional wraps can itself be optional, which leads to optionals nested inside optionals. To see why this isn't just a strange edge case or something the compiler should automatically coalesce, suppose you have an array of strings of numbers, which you want to convert into integers. You might run them through a map to convert them:

```
let stringNumbers = ["1", "2", "three"]
let maybeInts = stringNumbers.map { Int($0) } // [Optional(1), Optional(2), nil]
```

You now have an array of `Optional<Int>` — i.e. `Int?` — because `Int.init(String)` is failable, since the string might not contain a valid integer. Here, the last entry will be a `nil`, since "`three`" isn't an integer.

When looping over the array with `for`, you'd rightly expect that each element would be an optional integer, because that's what `maybeInts` contains:

```
for maybeInt in maybeInts {
    // maybeInt is an Int?
    // Two numbers and a `nil`.
}
```

Now consider that the implementation of `for...in` is shorthand for the while loop technique above. What's returned from `iterator.next()` would be an `Optional<Optional<Int>>` — or `Int??` — because `next` wraps each element in the sequence inside an optional. The while let unwraps it to check it isn't `nil`, and while it's non-`nil`, binds the unwrapped value and runs the body:

```
var iterator = maybeInts.makeIterator()
while let maybeInt = iterator.next() {
    print(maybeInt, terminator: " ")
}
// Optional(1) Optional(2) nil
```

When the loop gets to the final element — the nil from "three" — what's returned from next is a non-nil value: .some(nil). It unwraps this and binds what's inside (a nil) to maybeInt. Without nested optionals, this wouldn't be possible.

By the way, if you ever want to loop over only the non-nil values with for, you can use case pattern matching:

```
for case let i? in maybeInts {  
    // i will be an Int, not an Int?  
    print(i, terminator: " ")  
}  
// 1 2  
  
// Or only the nil values:  
for case nil in maybeInts {  
    // Will run once for each nil.  
    print("No value")  
}  
// No value
```

This uses a “pattern” of `x?`, which only matches non-nil values. This is shorthand for `.some(x)`, so the loop could be written like this:

```
for case let .some(i) in maybeInts {  
    print(i)  
}
```

This case-based pattern matching is a way to apply the same rules that work in switch statements to if, for, and while. It’s most useful with optionals, but it also has other applications — for example:

```
let j = 5  
if case 0..10 = j {  
    print("\(j) within range")  
} // 5 within range
```

We'll discuss pattern matching in depth in the [Enums](#) chapter.

if var and while var

Instead of let, you can use var with if, while, and for. This allows you to mutate the variable inside the statement body:

```
let number = "1"
if var i = Int(number) {
    i += 1
    print(i)
} // 2
```

But note that i will be a local copy; any changes to i won't affect the value inside the original optional. Optionals are value types, and unwrapping them copies the value inside.

Scoping of Unwrapped Optionals

Sometimes it's limiting to only have access to an unwrapped variable within the if block it has defined. For example, take the first property on arrays — a property that returns an optional of the first element, or nil when the array is empty. This is convenient shorthand for the following common bit of code:

```
let array = [1,2,3]
if !array.isEmpty {
    print(array[0])
}
// Outside the block, the compiler can't guarantee that array[0] is valid.
```

Using the first property is preferable, because you *have* to unwrap the optional to use it — you can't accidentally forget:

```
if let firstElement = array.first {
    print(firstElement)
}
// Outside the block, you can't use firstElement.
```

The unwrapped value is only available inside the if let block. This is great, but it's impractical if the purpose of the if statement is to exit early from a function when some condition isn't met. This early exit can help avoid annoying nesting or repeated checks later in the function. You might write the following:

```
func doStuff(withArray a: [Int]) {
    if a.isEmpty {
        return
    }
    // Now use a[0] or a.first! safely.
}
```

Here, the if let binding wouldn't work because the bound variable wouldn't be in scope after the if block. But you can nevertheless be sure the array will contain at least one element, so force-unwrapping the first element is safe, even if the syntax is still unappealing.

One option for using an unwrapped optional outside the scope it was bound in is to rely on Swift's deferred initialization capabilities. Consider the following example, which reimplements part of the pathExtension property from URL and NSString:

```
extension String {
    var fileExtension: String? {
        let period: String.Index
        if let idx = lastIndex(of: ".") {
            period = idx
        } else {
            return nil
        }
        let extensionStart = index(after: period)
        return String(self[extensionStart...])
    }
}

"hello.txt".fileExtension // Optional("txt")
```

The compiler checks your code to confirm there are only two possible paths: one in which the function returns early, and another where period is properly initialized. There's no way period could be nil (it isn't optional) or uninitialized (Swift won't let you use a variable that hasn't been initialized). So after the if statement, the code can be written without you having to worry about optionals at all.

However, the two previous examples are pretty ugly. Really, what's needed is some kind of if not let — which is exactly what guard let does:

```
func doStuff(withArray a: [Int]) {
```

```
guard let firstElement = a.first else {
    return
}
// firstElement is unwrapped here.
}
```

And the second example becomes much clearer:

```
extension String {
    var fileExtension: String? {
        guard let period = lastIndex(of: ".") else {
            return nil
        }
        let extensionStart = index(after: period)
        return String(self[extensionStart...])
    }
}
```

Anything can go in the `else` clause here, including multiple statements just like an `if ... else`. The only requirement is that the `else` block must leave the current scope. That might mean `return`, throwing an error, or calling `fatalError` (or any other function that returns `Never`). If the guard were in a loop, `break` or `continue` would also be allowed.

A function with the return type `Never` signals to the compiler that it'll never return. There are two common types of functions that do this: those that abort the program, such as `fatalError`; and those that run for the entire lifetime of the program, like `dispatchMain`. The compiler uses this information for its control flow diagnostics. For example, the `else` branch of a `guard` statement must either exit the current scope or call one of these never-returning functions.

`Never` is what's called an *uninhabited type*. It's a type that has no valid values and thus can't be constructed. A function declared to return an uninhabited type can never return normally.

In Swift, an uninhabited type is implemented as an enum that has no cases:

```
public enum Never { }
```

You won't usually need to define your own never-returning functions unless you write a wrapper for `fatalError` or `preconditionFailure`. One interesting use case is while you're writing new code: say you're working on a complex switch statement, gradually filling in all the cases, and the compiler is bombarding you with error messages for empty case labels or missing return values, while all you'd like to do is concentrate on the one case you're working on. In this situation, a few carefully placed calls to `fatalError()` can do wonders to silence the compiler. Consider writing a function called `unimplemented()` to better communicate the temporary nature of these calls:

```
func unimplemented() -> Never {  
    fatalError("This code path is not implemented yet.")  
}
```

`Never` is also commonly used in combination with `Result` and similar types in generic contexts. For example, consider a reactive programming framework like Apple's Combine. Combine models event streams with a `Publisher` protocol with two associated types, `Output` and `Failure`. `Output` describes the payload type of the emitted event, whereas `Failure` represents the error case. A text field object might provide an `AnyPublisher<String, Never>` that fires an event every time the user edits the text. Using `Never` as the failure type indicates to the programmer and the compiler that this publisher will never emit a failure because such a value is impossible to construct.

Swift meticulously distinguishes between different kinds of “nothingness.” In addition to `nil` and `Never`, there’s also `Void`, which is another way of writing an empty tuple:

```
public typealias Void = ()
```

The most common use of `Void` or `()` is in the types of functions that don’t return anything, but it has other applications too. For instance, a publisher for a button object would emit an event when the user taps the button, but it has no additional payload to send — its event stream should have the type `AnyPublisher<(), Never>`.

As [David Smith put it](#), Swift makes a careful distinction between the “absence of a thing” (`nil`), the “presence of nothing” (`Void`), and a “thing which cannot be” (`Never`).

Just like if, guard isn't limited to binding. guard can take any condition you might find in a regular if statement, so the empty array example could be rewritten with it:

```
func doStuff2(withArray a: [Int]) {  
    guard !a.isEmpty else { return }  
    // Now use a[0] or a.first! safely.  
}
```

Unlike the optional binding case, this guard isn't a big win — in fact, it's slightly more verbose than the original version. But it's still worth considering doing this with any early exit situation. For one, sometimes (though not in this case) the inversion of the Boolean condition can make things clearer. Additionally, guard is a clear signal when reading the code. It says: "We only continue if the following condition holds." Finally, the Swift compiler will check that you're definitely exiting the current scope and raise a compilation error if you don't. For this reason, we'd suggest using guard even when an if would do.

Optional Chaining

In Objective-C, sending a message to nil is a no-op. In Swift, the same effect can be achieved via "optional chaining":

```
delegate?.callback()
```

Unlike with Objective-C though, the Swift compiler will force you to acknowledge that the receiver could be nil. The question mark is a clear signal to the reader that the method might not be called.

When the method you call via optional chaining returns a result, that result will also be optional. Consider the following code to see why this must be the case:

```
let str: String? = "Never say never"  
// We want upper to be the uppercase string.  
let upper: String  
if str != nil {  
    upper = str!.uppercase()  
} else {  
    // No reasonable action to take at this point.  
    fatalError("No idea what to do now...")  
}
```

If str is non-nil, upper will have the desired value. But if str is nil, then upper can't be set to a value. So in the optional chaining case, upper2 *must* be optional to account for the possibility that str could've been nil:

```
let upper2 = str?.uppercased() // Optional("NEVER SAY NEVER")
```

You're not limited to a single method call after the question mark. As the term *optional chaining* implies, you can chain calls on optional values:

```
let lower = str?.uppercased().lowercased() // Optional("never say never")
```

This might look a bit surprising. Didn't we just say that the result of optional chaining is an optional? So why don't you need a ?. after uppercased()? This is because optional chaining is a “flattening” operation. If str?.uppercased() returned an optional and we called ?.lowercased() on it, then logically we'd get an optional optional. But we just want a regular optional, so instead we write the second chained call without the question mark to represent the fact that the optionality is already captured.

On the other hand, if the uppercased method itself returned an optional, then we *would* need to add a second ? to express that we were chaining *that* optional. For example, let's extend the Int type with a computed property named half. This property returns the result of dividing the integer by two, but only if the number is big enough to be divided. When the number is smaller than two, it returns nil:

```
extension Int {  
    var half: Int? {  
        guard self < -1 || self > 1 else { return nil }  
        return self / 2  
    }  
}
```

Because calling half returns an optional result, we need to keep putting in ? when calling it repeatedly. After all, at every step, the function might return nil:

```
20.half?.half?.half // Optional(2)
```

Notice that the compiler is still smart enough to flatten the result type for us. The type of the expression above is Int? and not Int???, as you might expect. The latter would give you more information — namely, which part of the chain failed — but it'd also make it a lot more cumbersome to deal with the result, destroying the convenience optional chaining adds in the first place.

So far, we've seen optional chaining for method calls and property accesses. It also applies to subscripts — for example:

```
let dictOfArrays = ["nine": [0, 1, 2, 3]]  
dictOfArrays["nine"]?[3] // Optional(3)
```

Additionally, you can use optional chaining to call optional functions:

```
let dictOfFunctions: [String: (Int, Int) -> Int] = [  
    "add": (+),  
    "subtract": (-)  
]  
dictOfFunctions["add"]?(1, 1) // Optional(2)
```

This is handy in typical callback situations where a class stores a callback function to inform its owner when an event occurs. Consider a `TextField` class:

```
class TextField {  
    private(set) var text = ""  
    var didChange: ((String) -> ())?  
  
    // Event handler called by the framework.  
    func textDidChange(newText: String) {  
        text = newText  
        // Trigger callback if non-nil.  
        didChange?(text)  
    }  
}
```

The `didChange` property stores a callback function, which the text field calls every time the user edits the text. Because the text field's owner doesn't have to register a callback, the property is optional; its initial value is `nil`. When the time comes to invoke the callback (in the `textDidChange` method above), optional chaining lets us do so in a very compact way.

Optional vars (not lets) are implicitly initialized to `nil` if you don't assign a value. This is the only exception Swift makes from its strict explicit initialization policy. The rationale for it is convenience, and that optionals have an “obvious” default value. Nevertheless, some members of the Swift

team have expressed regret at this design decision and would prefer to change it if it didn't break existing code. Curiously, the implicit initialization only kicks in for declarations that use the ...? shorthand for the type annotation. Had we written var didChange: Optional<(String) -> ()> above, the compiler would've complained that the property hadn't been initialized.

Optional Chaining and Assignments

You can even assign *through* an optional chain. Suppose you have an optional variable, and if it's non-nil, you wish to update one of its properties:

```
struct Person {  
    var name: String  
    var age: Int  
}  
  
var optionalLisa: Person? = Person(name: "Lisa Simpson", age: 8)  
// Increment age if non-nil.  
if optionalLisa != nil {  
    optionalLisa!.age += 1  
}
```

This is rather verbose and ugly. Note that you can't use optional binding in this case. Since Person is a struct and thus a value type, the bound variable is a local copy of the original value; mutating the former won't change the latter:

```
if var lisa = optionalLisa {  
    // Mutating lisa doesn't change optionalLisa.  
    lisa.age += 1  
}
```

This *would* work if Person were a class. We'll talk more about the differences between value and reference types in the [Structs and Classes](#) chapter. Regardless, it's still too verbose. Instead, you can assign to the chained optional value, and if it isn't nil, the assignment will go through:

```
optionalLisa?.age += 1
```

A weird (but logical) edge case of this feature is that it works for direct assignment to optional values. This is perfectly valid:

```
var a: Int? = 5  
a? = 10  
a // Optional(10)
```

```
var b: Int? = nil  
b? = 10  
b // nil
```

Notice the subtle difference between `a = 10` and `a? = 10`. The former assigns a new value unconditionally, whereas the latter only performs the assignment if the value of `a` is non-nil *before* the assignment.

The nil-Coalescing Operator

Often you want to unwrap an optional and replace `nil` with some default value. This is a job for the nil-coalescing operator:

```
let stringteger = "1"  
let number = Int(stringteger) ?? 0
```

If the string can be converted to an integer, `number` will be that integer, unwrapped. If it isn't and `Int.init` returns `nil`, the default value of 0 will be substituted. So `lhs ?? rhs` is analogous to `lhs != nil ? lhs! : rhs`.

If you're coming to Swift from another language, you might think the nil-coalescing operator is similar to the ternary operator (`a ? b : c`). For example, to get the first element out of an array that could be empty, you might write the following code:

```
let array = [1,2,3]  
!array.isEmpty ? array[0] : 0
```

Because Swift arrays provide a `first` property that's `nil` if the array is empty, you can use the nil-coalescing operator instead:

```
array.first ?? 0 // 1
```

This is cleaner and clearer — the intent (grab the first element in the array) is up front, with the default tacked on the end, joined with a ?? that signals “this is a default value.” Compare this with the ternary operator version, which starts with the check, is followed by the value, and ends with the default. And the check is awkwardly negated (the alternative being to put the default in the middle and the actual value on the end). And, as is the case with optionals, it’s impossible to forget that first is optional and accidentally use it without the check, because the compiler will stop you if you try.

Whenever you find yourself guarding a statement with a check to make sure the statement is valid, it’s a good sign optionals would be a better solution. Suppose that instead of an empty array, you’re checking a value that’s within the array bounds:

```
array.count > 5 ? array[5] : 0 // 0
```

Unlike first and last, getting an element out of an array by its index doesn’t return an Optional. But it’s easy to extend Array to add this functionality:

```
extension Array {  
    subscript(guarded idx: Int) -> Element? {  
        guard (startIndex..            return nil  
        }  
        return self[idx]  
    }  
}
```

This now allows you to write the following:

```
array[guarded: 5] ?? 0 // 0
```

Coalescing can also be chained — so if you have multiple possible optionals and you want to choose the first non-nil value, you can write them in sequence:

```
let i: Int? = nil  
let j: Int? = nil  
let k: Int? = 42  
i ?? j ?? k ?? 0 // 42
```

Because of this chaining, if you’re ever presented with a doubly nested optional and want to use the ?? operator, you must take care to distinguish between a ?? b ?? c (chaining) and (a ?? b) ?? c (unwrapping the inner and then outer layers):

```
let s1: String?? = nil
(s1 ?? "inner") ?? "outer" // inner
let s2: String?? = .some(nil)
(s2 ?? "inner") ?? "outer" // outer
```

If you think of the ?? operator as similar to an “or” statement, you can think of an if let with multiple clauses as an “and” statement:

```
if let n = i, let m = j {}
// similar to if i != nil && j != nil
```

Just like the || operator, the ?? operator uses short circuiting: when we write l ?? r, the right-hand side of the operator is only evaluated when the left-hand side is nil. This works because the function declaration for the operator uses an @autoclosure for its second parameter. We’ll discuss how autoclosures work in detail in the [Functions](#) chapter.

Using Optionals with String Interpolation

You may have noticed that the compiler emits a warning when you print an optional value or use one in a string interpolation expression:

```
let bodyTemperature: Double? = 37.0
let bloodGlucose: Double? = nil
print(bodyTemperature) // Optional(37.0)
// Warning: Expression implicitly coerced from 'Double?' to Any.
print("Blood glucose level: \(bloodGlucose)") // Blood glucose level: nil
// Warning: String interpolation produces a debug description
// for an optional value; did you mean to make this explicit?
```

This is generally helpful because it’s all too easy for a stray “Optional(...)” or “nil” to accidentally creep into a text displayed to the user. You should never use optionals directly in user-facing strings and always unwrap them first. Since all types are allowed in a string interpolation (including Optional), the compiler can’t make this a hard error, though — a warning is really the best it can do.

Sometimes you may *want* to use an optional in a string interpolation — to log its value for debugging, for example — and in that case, the warnings can become annoying. The compiler offers several fix-its to silence the warning: add an explicit cast with as Any,

force-unwrap the value with ! (if you're certain it can't be nil), wrap it in String(describing:), or provide a default value with the nil-coalescing operator.

The latter is often a quick and pretty solution, but it has one drawback: the types on both sides of the ?? expression must match, so the default value you provide for a Double? must be of type Double. Since the ultimate goal is to turn the expression into a string, it'd be convenient if you could provide a string as a default value in the first place.

Swift's ?? operator doesn't support this kind of type mismatch — after all, what would the type of the expression be if the two sides didn't have a common base type? But it's easy to add your own operator that's tailor-made for the purpose of working with optionals in string interpolation. Let's name it ???:

infix operator ???: NilCoalescingPrecedence

```
public func ???<T>(optional: T?, defaultValue: @autoclosure () -> String)
    -> String
{
    switch optional {
        case let value?: return String(describing: value)
        case nil: return defaultValue()
    }
}
```

This takes any optional T? on the left side and a string on the right. If the optional is non-nil, we unwrap it and return its string description. Otherwise, we return the default string that was passed in. The @autoclosure annotation makes sure we only evaluate the second operand when needed. In the [Functions](#) chapter, we'll go into this in more detail.

Now we can write the following and we won't get any compiler warnings:

```
print("Body temperature: \(bodyTemperature ??? "n/a")")
// Body temperature: 37.0
print("Blood glucose level: \(bloodGlucose ??? "n/a")")
// Blood glucose level: n/a
```

Optional map

Let's say we have an array of characters and we want to turn the first element into a string:

```
let characters: [Character] = ["a", "b", "c"]
String(characters[0]) // a
```

If it's possible for characters to be empty, we can use an if let to create the string only if the array is non-empty:

```
var firstCharAsString: String? = nil
if let char = characters.first {
    firstCharAsString = String(char)
}
```

So now, if the array contains at least one element, firstCharAsString will contain that element as a String. But if it doesn't, firstCharAsString will be nil.

This pattern — take an optional, and transform it if it isn't nil — is common enough that there's a method on optionals for doing this. It's called map, and it takes a function that represents how to transform the contents of the optional. Here's the above function, rewritten using map:

```
let firstChar = characters.first.map { String($0) } // Optional("a")
```

This map is, of course, very similar to the map on arrays or other sequences. But instead of operating on a sequence of values, it operates on just one value: the possible one inside the optional. You can think of an optional as being a collection of either zero or one values, with map either doing nothing to zero values or transforming one.

Here's one way to implement map on optionals:

```
extension Optional {
    func map<U>(transform: (Wrapped) -> U) -> U? {
        guard let value = self else { return nil }
        return transform(value)
    }
}
```

An optional map is especially nice when you already want an optional result. Suppose you wanted to write another variant of reduce for arrays. Instead of taking an initial value, it uses the first element in the array (in some languages, this might be called reduce1, but we'll call it reduce and rely on overloading).

Because of the possibility that the array might be empty, the result needs to be optional — without an initial value, what else could it be? You might write it like this:

```
extension Array {  
    func reduce(_ nextPartialResult: (Element, Element) -> Element)  
        -> Element?  
    {  
        // first will be nil if the array is empty.  
        guard let fst = first else { return nil }  
        return dropFirst().reduce(fst, nextPartialResult)  
    }  
}
```

And you can use it like this:

```
[1, 2, 3, 4].reduce(+) // Optional(10)
```

Since optional map returns nil if the optional is nil, our variant of reduce could be rewritten using a single return statement (and no guard):

```
extension Array {  
    func reduce_alt(_ nextPartialResult: (Element, Element) -> Element)  
        -> Element?  
    {  
        first.map {  
            dropFirst().reduce($0, nextPartialResult)  
        }  
    }  
}
```

Optional flatMap

As we saw in the [Built-In Collections](#) chapter, it's common to want to map over a collection with a function that returns a collection but collect the results as a single array rather than an array of arrays.

Similarly, if you want to perform a map on an optional value but your transformation function also has an optional result, you'll end up with a doubly nested optional. An example of this is when you want to fetch the first element of an array of strings as a number, using first on the array and then map to convert it to a number:

```
let stringNumbers = ["1", "2", "3", "foo"]
let x = stringNumbers.first.map { Int($0) } // Optional(Optional(1))
```

The problem is that since map returns an optional (first might have been nil) and Int(string) returns an optional (the string might not be an integer), the type of x will be Int??.

flatMap will instead flatten the result into a single optional. As a result, y will be of type Int?:

```
let y = stringNumbers.first.flatMap { Int($0) } // Optional(1)
```

Instead, you could've written this with if let, because values that are bound later can be computed from earlier ones:

```
if let a = stringNumbers.first, let b = Int(a) {
    print(b)
} // 1
```

This shows that flatMap and if let are very similar. Earlier in this chapter, we saw an example that uses a multiple-if-let statement, and we can rewrite it using map and flatMap instead:

```
let urlString = "https://www.objc.io/logo.png"
let view = URL(string: urlString)
    .flatMap { try? Data(contentsOf: $0) }
    .flatMap { UIImage(data: $0) }
    .map { UIImageView(image: $0) }

if let view = view {
    PlaygroundPage.current.liveView = view
}
```

Optional chaining is also similar to flatMap: i?.advance(by: 1) is essentially equivalent to i.flatMap { \$0.advance(by: 1) }.

Since we've shown that a multiple-if-let statement is equivalent to flatMap, we could implement one in terms of the other:

```
extension Optional {  
    func flatMap<U>(transform: (Wrapped) -> U?) -> U? {  
        if let value = self, let transformed = transform(value) {  
            return transformed  
        }  
        return nil  
    }  
}
```

Filtering Out nils with compactMap

If you have a sequence and it contains optionals, you might not care about the nil values. In fact, you might just want to ignore them.

Suppose you wanted to process only the numbers in an array of strings. This is easily done in a for loop using optional pattern matching:

```
let numbers = ["1", "2", "3", "foo"]  
var sum = 0  
for case let i? in numbers.map({ Int($0) }) {  
    sum += i  
}  
sum // 6
```

You might also want to use ?? to replace the nils with zeros:

```
numbers.map { Int($0) }.reduce(0) { $0 + ($1 ?? 0) } // 6
```

But really, you just want a version of map that filters out nil and unwraps the non-nil values. Enter the standard library's `compactMap` on sequences, which does exactly that:

```
numbers.compactMap { Int($0) }.reduce(0, +) // 6
```

To define our own version of `compactMap`, we first map over the entire array, and then we filter out the non-nil values, and finally we unwrap each element:

```
extension Sequence {
    func compactMap<B>(_ transform: (Element) -> B?) -> [B] {
        return lazy.map(transform).filter { $0 != nil }.map { $0! }
    }
}
```

In the implementation, we use `lazy` to defer the creation of the array until the last moment. This is possibly a micro-optimization, but it might be worthwhile for larger sequences. Using `lazy` saves the allocation of multiple intermediate arrays. The standard library doesn't do this in its [compactMap implementation](#) though. In the [Collection Protocols](#) chapter, we'll look at lazy sequences and collections in more detail.

Equating Optionals

Often you don't care whether a value is `nil` or not — just whether it contains a certain value:

```
let regex = "^Hello$"
// ...
if regex.first == "^" {
    // Match only the start of the string.
}
```

In this case, it doesn't matter if the value is `nil` or not — if the string is empty, the first character can't be a caret, so you don't want to run the block. But you still want the protection and simplicity of `first`. The alternative, `if !regex.isEmpty && regex[regex.startIndex] == "^"`, is horrible.

The code above relies on two things to work. First of all, `Optional` conforms to `Equatable`, but only if its `Wrapped` type also conforms to `Equatable`:

```
extension Optional: Equatable where Wrapped: Equatable {
    static func ==(lhs: Wrapped?, rhs: Wrapped?) -> Bool {
        switch (lhs, rhs) {
            case (nil, nil): return true
            case let (x?, y?): return x == y
        }
    }
}
```

```
    case (_, nil), (nil, _?): return false
}
}
}
```

When comparing two optionals, there are four possibilities: they're both nil, or they both have a value, or either one or the other is nil. The switch exhaustively tests all four possibilities (hence no need for a default clause). It defines two nils to be equal to each other, nil to never be equal to non-nil, and two non-nil values to be equal if their unwrapped values are equal.

Second, notice that we did *not* have to write the following:

```
if regex.first == Optional("^") { // or: == .some("^")
    // Match only the start of the string.
}
```

This is because whenever you have a non-optional value, Swift will always be willing to promote it to an optional value to make the types match.

This implicit conversion is incredibly useful for writing clear, compact code. Suppose there was no such conversion, but to make things nice for the caller, you wanted a version of `==` that worked between both optional and non-optional types. You'd have to write three separate versions:

```
// Both optional.
func == <T: Equatable>(lhs: T?, rhs: T?) -> Bool
// lhs non-optional.
func == <T: Equatable>(lhs: T, rhs: T?) -> Bool
// rhs non-optional.
func == <T: Equatable>(lhs: T?, rhs: T) -> Bool
```

But instead, only the first version is necessary, and the compiler will convert to optionals where necessary.

In fact, we've been relying on this feature throughout the book. For example, when we implemented `Optional.map`, we transformed the inner value and returned it. But the return value of `map` is an `Optional`. The compiler automatically converted the value for us — we didn't have to write `return Optional(transform(value))`.

Swift code constantly relies on this implicit conversion. For example, dictionary subscript lookup by key returns an optional (the key might not be present). But it also takes an optional on assignment — subscripts have to both take and receive the same type. Without implicit conversion, you'd have to write `myDict["someKey"] = Optional(someValue)`.

Incidentally, if you're wondering what happens to dictionaries with key-based subscript assignment when you assign a nil value, the answer is that the key is removed. This can be useful, but it also means you need to be a little careful when dealing with a dictionary with an optional value type. Consider this dictionary:

```
var dictWithNils: [String: Int?] = [
    "one": 1,
    "two": 2,
    "none": nil
]
```

The dictionary has three keys, and one of them has a value of nil. Suppose we wanted to set the value of the "two" key to nil as well. This will *not* do that:

```
dictWithNils["two"] = nil
dictWithNils // ["one": Optional(1), "none": nil]
```

Instead, it'll *remove* the "two" key.

To change the value for the key, you'd have to write one of the following (they all work, so choose whichever you feel is clearer):

```
dictWithNils["two"] = Optional(nil)
dictWithNils["two"] = .some(nil)
dictWithNils["two"]? = nil
dictWithNils // ["one": Optional(1), "two": nil, "none": nil]
```

Note that the third version above is slightly different than the other two. It works because the "two" key is already in the dictionary, so it uses optional chaining to set its value if successfully fetched. Now try this with a key that isn't present:

```
dictWithNils["three"]? = nil
dictWithNils.index(forKey: "three") // nil
```

You can see that nothing would be updated/inserted.

Comparing Optionals

Similar to `==`, there used to be implementations of `<`, `>`, `<=`, and `>=` for optionals. For Swift 3.0, these comparison operators were removed for optionals because they can easily yield unexpected results.

For example, `nil < .some(_)` would return true. In combination with higher-order functions or optional chaining, this can be very surprising. Consider the following (outdated) example:

```
let temps = ["-459.67", "98.6", "0", "warm"]
let belowFreezing = temps.filter { Double($0) < 0 }
```

Because `Double("warm")` will return `nil` and `nil` was defined as less than 0, it would've been included in the `belowFreezing` temperatures. This is unexpected indeed.

If you need inequality relations between optionals, you now have to unwrap the values first and thereby explicitly decide how `nil` values should be handled.

When to Force-Unwrap

Given all these techniques for cleanly unwrapping optionals, when should you use `!`, the force-unwrap operator? There are many opinions on this scattered throughout the internet, including “never,” “whenever it makes the code clearer,” and “when you can’t avoid it.” We propose the following rule, which encompasses most of them:

Use `!` when you’re so certain that a value won’t be `nil` that you *want* your program to crash if it ever is.

As an example, take the implementation of `compactMap` from above:

```
extension Sequence {
    func compactMap<B>(_ transform: (Element) -> B?) -> [B] {
        return lazy.map(transform).filter { $0 != nil }.map { $0! }
    }
}
```

Here, there's no possible way that `$0!` inside `map` will ever hit a `nil`, since the `nil` elements were all filtered out in the preceding step. This function could certainly be written to eliminate the force-unwrap operator by looping over the array and adding non-`nil` values into an array. But the `filter/map` version is cleaner and probably clearer, so the `!` is justified.

However, these cases are pretty rare. If you have full mastery of all the unwrapping techniques described in this chapter, chances are there's a better way than force-unwrapping. Whenever you do find yourself reaching for `!`, it's worth taking a step back and wondering if there really is no other option.

As another example, consider the following code that fetches all the keys in a dictionary with values matching a certain condition:

```
let ages = [
  "Tim": 53, "Angela": 54, "Craig": 44,
  "Jony": 47, "Chris": 37, "Michael": 34,
]
ages.keys
  .filter { name in ages[name]! < 50 }
  .sorted()
// ["Chris", "Craig", "Jony", "Michael"]
```

Again, the `!` is perfectly safe here — since all the keys came from the dictionary, there's no possible way a key could be missing from the dictionary.

But you could also rewrite the statement to eliminate the need for a force-unwrap altogether. Using the fact that dictionaries present themselves as sequences of key-value pairs, you could just filter this sequence and then run it through a map to remove the value:

```
ages.filter { (_, age) in age < 50 }
  .map { (name, _) in name }
  .sorted()
// ["Chris", "Craig", "Jony", "Michael"]
```

This version even has a possible performance benefit because it avoids unnecessary key lookups.

Nonetheless, sometimes life hands you an optional, and you know *for certain* that it isn't `nil`. So certain are you of this that you'd *rather* your program crash than continue,

because encountering a nil value would mean there's a very nasty bug in your logic. Better to trap than to continue under those circumstances, so ! acts as a combined unwrap-or-error operator in one handy character. This approach is often a better move than just using the nil chaining or coalescing operators to sweep theoretically impossible situations under the carpet.

Improving Force-Unwrap Error Messages

That said, even when you're force-unwrapping an optional value, you have options other than using the ! operator. When your program does error, you don't get much by way of description as to why in the output log.

Chances are, you'll leave a comment as to why you're justified in force-unwrapping. So why not have that comment serve as the error message too? Here's an operator, !!; it combines unwrapping with supplying a more descriptive error message to be logged when the application exits:

infix operator !!

```
func !!<T>(wrapped: T?, failureText: @autoclosure () -> String) -> T {  
    if let x = wrapped { return x }  
    fatalError(failureText())  
}
```

Now you can write a more descriptive error message, including the value you expected to be able to unwrap:

```
let s = "foo"  
let i = Int(s) !! "Expecting integer, got \"\\"(s)\\""
```

Asserting in Debug Builds

Still, choosing to crash even on release builds is quite a bold move. Often, you might prefer to assert during debug and test builds, but in production, you'd substitute a valid default value — perhaps zero or an empty array.

Enter the interrobang operator, !?. We define this operator to assert on failed unwraps and also to substitute a default value when the assertion doesn't trigger in release mode:

infix operator !?

```
func !?<T: ExpressibleByIntegerLiteral>
(wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? 0
}
```

Now, the following will assert while debugging but print 0 in release:

```
let s = "20"
let i = Int(s) !? "Expecting integer, got \"\$(s)\""
```

Overloading for other literal convertible protocols enables a broad coverage of types that can be defaulted:

```
func !?<T: ExpressibleByArrayLiteral>
(wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? []
}

func !?<T: ExpressibleByStringLiteral>
(wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? ""
}
```

And for when you want to provide a different explicit default, or for non-standard types, you can define a version that takes a pair — the default and the error text:

```
func !?<T>(wrapped: T?,
nilDefault: @autoclosure () -> (value: T, text: String)) -> T
{
    assert(wrapped != nil, nilDefault().text)
    return wrapped ?? nilDefault().value
}
```

```
// Asserts in debug, returns 5 in release.
```

```
Int(s) !? (5, "Expected integer")
```

Since optionally chained method calls on methods that return Void return Void?, you can also write a non-generic version to detect when an optional chain hits a nil, resulting in a no-op:

```
func !?(wrapped: ()?, failureText: @autoclosure () -> String) {  
    assert(wrapped != nil, failureText())  
}  
  
var output: String? = nil  
output?.write("something") !? "Wasn't expecting chained nil here"
```

There are three ways to halt execution. The first option, fatalError, takes a message and stops execution unconditionally. The second option, assert, checks a condition and a message and stops execution if the condition evaluates to false. In release builds, the assert gets removed — the condition isn't checked (and execution is never halted). The third option is precondition, which has the same interface as assert, but it doesn't get removed from release builds, so if the condition evaluates to false, execution is stopped.

Implicitly Unwrapped Optionals

Make no mistake: implicitly unwrapped optionals — types marked with an exclamation point, such as `UIView!` — are still optionals, albeit ones that are automatically force-unwrapped whenever you use them. Now that we know force-unwraps will crash your application if they're ever nil, why on earth would you use them? Well, two reasons really.

Reason 1: Temporarily, because you're calling Objective-C code that hasn't been audited for nullability, or because you're calling into a C library that doesn't have Swift-specific annotations.

The sole reason implicitly unwrapped optionals even exist is to make interoperability with Objective-C and C easier. Of course, on the first day you start writing Swift against an existing Objective-C codebase, any Objective-C method that returns a reference will translate into an implicitly unwrapped optional. Since, for most of Objective-C's lifetime, there was no way to indicate that a reference was nullable, there was little option other

than to assume any call returning a reference might return a `nil` reference. But few Objective-C APIs *actually* return null references, so it'd be incredibly annoying to automatically expose them as optionals. Since everyone was used to dealing with the “maybe null” world of Objective-C objects, implicitly unwrapped optionals were a reasonable compromise.

So you see them in unaudited bridged Objective-C code. But you should *never* see a pure native Swift API returning an implicit optional (or passing one into a callback).

Reason 2: Because a value is `nil` *very* briefly, for a well-defined period of time, and is then never `nil` again.

The most common scenario is two-phase initialization. By the time your class is ready to use, the implicitly wrapped optionals will all have a value. This is the reason Xcode/Interface Builder uses them in the view controller lifecycle: in Cocoa and Cocoa Touch, view controllers create their views lazily, so there exists a time window — after a view controller has been initialized but before it has loaded its view — when the view objects referenced by the view controller have not yet been created.

Implicit Optional Behavior

While implicitly unwrapped optionals usually behave like non-optional values, you can still use most of the unwrap techniques to safely handle them like optionals — chaining, `nil`-coalescing, `if let`, `map`, or just comparing them to `nil` all work the same:

```
var s: String! = "Hello"
s?.isEmpty // Optional(false)
if let s = s { print(s) } // Hello
s = nil
s ?? "Goodbye" // Goodbye
```

Recap

Optionals are touted as one of Swift's biggest features for writing safer code, and we certainly agree. If you think about it though, the real breakthrough isn't optionals — it's *non-optionals*. Almost every mainstream language has the concept of “null” or “nil”; what most of them don't have is the ability to declare a value as “never nil.” Or, alternatively, some types (like non-class types in Objective-C or Java) are “always

non-nil,” forcing developers to come up with magic values to represent the absence of a value.

APIs whose inputs and outputs are carefully designed with optionals in mind are more expressive and easier to use; there’s less need to refer to the documentation because the types carry more information.

All the unwrapping techniques we demonstrated in this chapter are Swift’s attempt to make bridging the two worlds of optional and non-optional values as painless as possible. Which method you should use is often a matter of personal preference.

Functions

4

Overview

To open this chapter, let's recap some main points regarding functions. If you're already familiar with first-class functions, feel free to skip ahead to the [next section](#). But if you're even slightly unsure about them, read through what's below.

To understand functions and closures in Swift, you really need to understand three things, in roughly this order of importance:

0. Functions can be assigned to variables and passed in and out of other functions as arguments, just as an `Int` or a `String` can be.
1. Functions can *capture* variables that exist outside of their local scope.
2. There are two ways of creating functions — either with the `func` keyword, or with `{}`. Swift calls the latter *closure expressions*.

Sometimes people new to the topic of closures come at it in reverse order and maybe miss one of these points, or they conflate the terms *closure* and *closure expression* — and this can cause a lot of confusion. It's a three-legged stool, and if you miss one of the three points above, you'll fall over when you try to sit down.

1. Functions can be assigned to variables and passed in and out of other functions as arguments.

In Swift, as in many modern languages, functions are referred to as “first-class objects.” You can assign functions to variables, and you can pass them in and out of other functions to be called later.

This is *the most important thing* to understand. “Getting” this for functional programming is akin to “getting” pointers in C. If you don’t quite grasp this part, everything else will just be noise.

Let's start with a function that just prints an integer:

```
func printInt(i: Int) {  
    print("You passed \(i).")  
}
```

To assign the function to a variable, `funVar`, we use the function name as the value. Note the absence of parentheses after the function name:

```
let funVar = printInt
```

Now we can call the `printInt` function using the `funVar` variable. Note the use of parentheses after the variable name:

```
funVar(2) // You passed 2.
```

It's also noteworthy that we *must not* include an argument label in the `funVar` call, whereas `printInt` calls *require* the argument label, as in `printInt(i: 2)`. Swift only allows argument labels in function *declarations*; the labels aren't included in a function's *type*. This means you currently can't assign argument labels to a variable of a function type, though this will likely change in a future Swift version.

We can also write a function that takes a function as an argument:

```
func useFunction(function: (Int) -> () {
    function(3)
}

useFunction(function: printInt) // You passed 3.
useFunction(function: funVar) // You passed 3.
```

Why is being able to treat functions like this such a big deal? Because it enables us to easily write “higher-order” functions, which take functions as arguments and apply them in useful ways, as we saw in the Built-In Collections chapter.

Functions can also return other functions:

```
func returnFunc() -> (Int) -> String {
    func innerFunc(i: Int) -> String {
        return "You passed \(i)."
    }
    return innerFunc
}
let myFunc = returnFunc()
myFunc(3) // You passed 3.
```

2. Functions can *capture* variables that exist outside of their local scope.

When a function references variables outside its scope, those variables are *captured* and stick around after they'd otherwise fall out of scope and be destroyed.

To see this, let's revisit our `returnFunc` function but add a counter that increases each time we call it:

```
func makeCounter() -> (Int) -> String {  
    var counter = 0  
    func innerFunc(i: Int) -> String {  
        counter += i // counter is captured  
        return "Running total: \(counter)"  
    }  
    return innerFunc  
}
```

Normally `counter`, being a local variable of `makeCounter`, would go out of scope just after the `return` statement, and it'd be destroyed. Instead, because it's captured by `innerFunc`, the Swift runtime will keep it alive until the function that captured it gets destroyed. We can call the inner function multiple times, and we see that the running total increases:

```
let f = makeCounter()  
f(3) // Running total: 3  
f(4) // Running total: 7
```

If we call `makeCounter()` again, a fresh counter variable will be created and captured:

```
let g = makeCounter()  
g(2) // Running total: 2  
g(2) // Running total: 4
```

This doesn't affect our first function, which still has its own captured version of `counter`:

```
f(2) // Running total: 9
```

Think of these functions combined with their captured variables as similar to instances of classes with a single method (the function) and some member variables (the captured variables).

In programming terminology, a combination of a function and an environment of captured variables is called a *closure*. So f and g above are examples of closures, because they capture and use a non-local variable (counter) that was declared outside of them.

3. Functions can be declared using the {} syntax for closure expressions.

In Swift, you can define functions in two ways. One is with the `func` keyword. The other way is to use a *closure expression*. Consider this simple function to double a number:

```
func doubler(i: Int) -> Int {  
    return i * 2  
}  
[1, 2, 3, 4].map(doubler) // [2, 4, 6, 8]
```

And here's the same function written using the closure expression syntax. Just like before, we can pass it to `map`:

```
let doublerAlt = { (i: Int) -> Int in return i*2 }  
[1, 2, 3, 4].map(doublerAlt) // [2, 4, 6, 8]
```

Functions declared as closure expressions can be thought of as *function literals* in the same way that 1 and "hello" are integer and string literals. They're also anonymous — they aren't named, unlike with the `func` keyword. The only way they can be used is to assign them to a variable when they're created (as we do here with `doubler`), or to pass them to another function or method.

There's a third way anonymous functions can be used: you can call a function directly in line as part of the same expression that defines it. This can be useful for defining properties whose initialization requires more than one line. We'll see an example of this in the [Properties](#) chapter.

The `doubler` declared using the closure expression and the one declared earlier using the `func` keyword are completely equivalent, apart from the differences in their handling of argument labels that we mentioned above. They even exist in the same “namespace,” unlike in some languages.

Why is the {} syntax useful then? Why not just use func every time? Well, it can be a lot more compact, especially when writing quick functions to pass into other functions, such as map. Here's our doubler map example written in a much shorter form:

```
[1, 2, 3].map { $0 * 2 } // [2, 4, 6]
```

This looks very different because we leveraged several features of Swift to make the code more concise. Here they are, one by one:

0. If you're passing the closure in as an argument and that's all you need it for, there's no need to store it in a local variable first. Think of this like passing in a numeric expression, such as $5*i$, to a function that takes an Int as a parameter.
1. If the compiler can infer a type from the context, you don't need to specify it. In our example, the function passed to map takes an Int (inferred from the type of the array elements) and returns an Int (inferred from the type of the multiplication expression).
2. If the closure expression's body contains just a single expression, it'll automatically return the value of the expression, and you can leave off the return.
3. Swift automatically provides shorthand names for the arguments to the function — \$0 for the first, \$1 for the second, etc.
4. If the last argument to a function is a closure expression, you can move the expression outside the parentheses of the function call. This *trailing closure syntax* is nice if you have a multi-line closure expression, as it more closely resembles a regular function definition or other block statement, such as if expr {}. Since Swift 5.3, even multiple trailing closures are supported.
5. Finally, if a function has no arguments other than a closure expression, you can leave off the parentheses after the function name altogether.

Using each of these rules, we can boil down the expression below to the form shown above:

```
/* */ [1, 2, 3].map( { (i: Int) -> Int in return i * 2 } )
/* */ [1, 2, 3].map( { i in return i * 2 } )
/* */ [1, 2, 3].map( { i in i * 2 } )
/* */ [1, 2, 3].map( { $0 * 2 } )
/* */ [1, 2, 3].map() { $0 * 2 }
/* */ [1, 2, 3].map { $0 * 2 }
```

If you're new to Swift's syntax, and to functional programming in general, these compact function declarations might seem daunting at first. But as you get more comfortable with the syntax and the functional programming style, they'll start to feel more natural, and you'll appreciate the ability to remove the clutter so you can see more clearly what the code is doing. Once you get used to reading code written like this, it'll be even easier to parse at a glance than the equivalent code written with a conventional for loop.

Sometimes, Swift needs a helping hand with inferring types. And sometimes, you may get something wrong and the types aren't what you think they should be. If ever you get a mysterious error when trying to supply a closure expression, it's a good idea to write out the full form (first version above), complete with types. In many cases, that will help clear up where things are going wrong. Once you have the long form compiling, take the types out again one by one until the compiler complains. And if the error was yours, you'll have fixed your code in the process.

Swift will also insist you be more explicit sometimes. For example, you can't completely ignore input parameters. Suppose you wanted an array of random numbers. A quick way to do this is to map a range with a function that just generates random numbers. Nonetheless, you must supply an argument. You can use `_` in such a case to indicate to the compiler that you acknowledge there's an argument but that you don't care what it is:

```
(0..3).map { _ in Int.random(in: 1..100) } // [26, 57, 48]
```

When you need to explicitly type the variables, you don't have to do it inside the closure expression. For example, try defining `isEven` without any types:

```
let isEven = { $0 % 2 == 0 }
```

Above, the type of `isEven` is inferred to be `(Int) -> Bool` in the same way that `let i = 1` is inferred to be `Int` — because `Int` is the default type for integer literals.

This is because of a type alias, `IntegerLiteralType`, in the standard library:

```
protocol ExpressibleByIntegerLiteral {
    associatedtype IntegerLiteralType
    /// Create an instance initialized to `value`.
    init(integerLiteral value: IntegerLiteralType)
}
```

```
/// The default type for an otherwise-unconstrained integer literal.  
typealias IntegerLiteralType = Int
```

If you were to define your own type alias, it would override the default one and change this behavior:

```
typealias IntegerLiteralType = UInt32  
let i = 1 // i will be of type UInt32.
```

This is almost certainly a bad idea.

If, however, you need a version of `isEven` for a different type, you could type the argument and the return value inside the closure expression:

```
let isEvenAlt = { (i: Int8) -> Bool in i % 2 == 0 }
```

But you could also supply the context from *outside* the closure:

```
let isEvenAlt2: (Int8) -> Bool = { $0 % 2 == 0 }  
let isEvenAlt3 = { $0 % 2 == 0 } as (Int8) -> Bool
```

Since closure expressions are most commonly used in some context of existing input or output types, adding an explicit type isn't often necessary, but it's useful to know you can do this.

Of course, it would've been much better to define a generic version of `isEven` that works on *any* integer as a computed property:

```
extension BinaryInteger {  
    var isEven: Bool { return self % 2 == 0 }  
}
```

Alternatively, we could have chosen to define an `isEven` variant for all `Integer` types as a free function:

```
func isEven<T: BinaryInteger>(_ i: T) -> Bool {  
    return i % 2 == 0  
}
```

If you want to assign that free function to a variable, this is also when you'd have to lock down which specific types it's operating on. A variable can't hold a generic function — only a specific one:

```
let int8IsEven: (Int8) -> Bool = isEven
```

One final point on naming. It's important to keep in mind that functions declared with `func` can be closures, just like ones declared with `{}` can. Remember, a closure is a function combined with any captured variables. While functions created with `{}` are called *closure expressions*, people often refer to this syntax as just *closures*. But don't get confused and think that functions declared with the closure expression syntax are different from other functions — they aren't. They're all functions, and they can all be closures.

Flexibility through Functions

In the [Built-In Collections](#) chapter, we talked about [parameterizing behavior by passing functions as arguments](#). Let's look at another example of this: sorting.

Sorting a collection in Swift is simple:

```
let myArray = [3, 1, 2]
myArray.sorted() // [1, 2, 3]
```

There are four sort methods: the non-mutating variant `sorted(by:)`, and the mutating `sort(by:)`, times two for the versions that default to sorting comparable things in ascending order and take no arguments. For the most common case, `sorted()` is all you need. And if you want to sort in a different order, just supply a function:

```
myArray.sorted(by: >) // [3, 2, 1]
```

You can also supply a function if your elements don't conform to `Comparable` but *do* have a `<` operator, like tuples do:

```
var numberStrings = [(2, "two"), (1, "one"), (3, "three")]
numberStrings.sort(by: <)
numberStrings // [(1, "one"), (2, "two"), (3, "three")]
```

(A proposal to automatically conform tuples to standard protocols such as Comparable, [SE-0283](#), was accepted in 2020 but isn't yet implemented as of Swift 5.5.)

Or, you can supply a more complicated function if you want to sort by some arbitrary criteria:

```
let animals = ["elephant", "zebra", "dog"]
animals.sorted { lhs, rhs in
    let l = lhs.reversed()
    let r = rhs.reversed()
    return l.lexicographicallyPrecedes(r)
}
// ["zebra", "dog", "elephant"]
```

It's this last ability — the ability to use any comparison function to sort a collection — that makes the Swift sort so powerful.

However, what if we wanted to sort by multiple criteria? For example, consider a Person struct that we want to sort by last name, and then by first name if the last names are equal. In Objective-C, this was done using NSSortDescriptor. While NSSortDescriptor (and its modern variant, SortDescriptor) is flexible and powerful, it only works on NSObject. This limitation exists because it uses Objective-C's runtime system. In this section, we'll use higher-order functions to reimplement our own SortDescriptor that's equally flexible and powerful.

We start by defining a Person type:

```
struct Person {
    let first: String
    let last: String
    let yearOfBirth: Int
}
```

Let's also define an array of people with different names and birth years:

```
let people = [
    Person(first: "Emily", last: "Young", yearOfBirth: 2002),
    Person(first: "David", last: "Gray", yearOfBirth: 1991),
    Person(first: "Robert", last: "Barnes", yearOfBirth: 1985),
    Person(first: "Ava", last: "Barnes", yearOfBirth: 2000),
    Person(first: "Joanne", last: "Miller", yearOfBirth: 1994),
```

```
Person(first: "Ava", last: "Barnes", yearOfBirth: 1998),  
]
```

We want to sort this array first by last name, then by first name, and finally by birth year. The ordering should respect the user's locale settings. At first, we'll sort by just one key, the last name:

```
/* */people.sorted { p1, p2 in  
    p1.last.localizedStandardCompare(p2.last) == .orderedAscending  
}
```

If we want to compare by last name and then first name, it already gets way more complicated:

```
/* */people.sorted { p1, p2 in  
    switch p1.last.localizedStandardCompare(p2.last) {  
        case .orderedAscending:  
            return true  
        case .orderedDescending:  
            return false  
        case .orderedSame:  
            return p1.first.localizedStandardCompare(p2.first) == .orderedAscending  
    }  
}
```

Functions as Data

Rather than writing an even more complicated function to include the birth year as well, we can take a step back and try to introduce an abstraction that describes the ordering of values. The standard library's `sort(by:)` and `sorted(by:)` methods use a comparison function that takes two objects and returns true if they're ordered correctly. We could name this our sort descriptor by defining a generic type alias for it:

```
typealias SortDescriptor<Root> = (Root, Root) -> Bool
```

Another alternative is to define a wrapper struct around that function. The advantage of wrapping the function in a struct is that we can define multiple initializers and instance methods and make them easy to discover through code completion. Whether you wrap a

function in a struct or not is mostly a question of personal preference, but in this case, it makes the resulting API feel more at home in Swift:

```
struct SortDescriptor<Root> {
    var areInIncreasingOrder: (Root, Root) -> Bool
}
```

As an example, we could define a sort descriptor that compares two Person values by year of birth, or a sort descriptor that sorts by last name:

```
let sortByYear: SortDescriptor<Person> = .init { $0.yearOfBirth < $1.yearOfBirth }
let sortByLastName: SortDescriptor<Person> = .init {
    $0.last.localizedStandardCompare($1.last) == .orderedAscending
}
```

Rather than writing the sort descriptors by hand, we can write a function that generates them. It's not nice that we have to write the same property twice: in sortByLastName, we could have easily made a mistake and accidentally compared \$0.last with \$1.first. Also, it's tedious to write these sort descriptors; to sort by first name, it's probably easiest to copy and paste the sortByLastName definition and modify it. First, let's make it easier to create a sort descriptor that works for any property that's Comparable:

```
extension SortDescriptor {
    init<Value: Comparable>(_ key: @escaping (Root) -> Value) {
        self.areInIncreasingOrder = { key($0) < key($1) }
    }
}
```

The key function describes how to drill down into an element of type Root and extract the value of type Value that's relevant for one particular sorting step. It has a lot in common with Swift's key paths, which is the reason we borrowed the naming of the generic parameters — Root and Value — from the KeyPath type. Later in this chapter, we'll discuss how to rewrite sort descriptors using key paths.

Now we can define our sortByYear descriptor using the new initializer:

```
let sortByYearAlt: SortDescriptor<Person> = .init { $0.yearOfBirth }
```

Similarly, we can create an initializer for the case where we have a function with the same shape as `localizedStandardCompare` and the other Foundation methods that return a `ComparisonResult`. If we make the `String` part generic, our initializer looks like this:

```
extension SortDescriptor {  
    init<Value>(_ key: @escaping (Root) -> Value,  
                 by compare: @escaping (Value) -> (Value) -> ComparisonResult) {  
        self.areInIncreasingOrder = {  
            compare(key($0))(key($1)) == .orderedAscending  
        }  
    }  
}
```

If you check out the type of the expression `String.localizedStandardCompare`, you'll notice that it's `(String) -> (String) -> ComparisonResult`. What's going on there? Under the hood, instance methods are modeled as functions that, given an instance, return another function that then operates on the instance. `someString.localizedStandardCompare` is really just another way of writing `String.localizedStandardCompare(someString)` — both expressions return a function of type `(String) -> ComparisonResult`, and this function is a closure that has captured `someString`.

This allows us to write `sortByFirstName` in a very concise way:

```
let sortByFirstName: SortDescriptor<Person> =  
.init({ $0.firstName }, by: String.localizedStandardCompare)
```

When we want to sort by multiple properties, we can combine two sort descriptors into one. We can first compare using the primary sort descriptor, and if the values are in neither increasing nor decreasing order, we use the result from the second sort descriptor:

```
extension SortDescriptor {  
    func then(_ other: SortDescriptor<Root>) -> SortDescriptor<Root> {  
        SortDescriptor { x, y in  
            if areInIncreasingOrder(x,y) { return true }  
            if areInIncreasingOrder(y,x) { return false }  
        }  
    }  
}
```

```
        return other.areInIncreasingOrder(x,y)
    }
}
}
```

This allows us to chain all three of our sort descriptors into a single sort descriptor that combines them:

```
let combined = sortByLastName.then(sortByFirstName).then(sortByYear)
people.sorted(by: combined.areInIncreasingOrder)
/*
[Person(first: "Ava", last: "Barnes", yearOfBirth: 1998),
 Person(first: "Ava", last: "Barnes", yearOfBirth: 2000),
 Person(first: "Robert", last: "Barnes", yearOfBirth: 1985),
 Person(first: "David", last: "Gray", yearOfBirth: 1991),
 Person(first: "Joanne", last: "Miller", yearOfBirth: 1994),
 Person(first: "Emily", last: "Young", yearOfBirth: 2002)]
*/
```

While our solution isn't yet as expressive as Foundation's sort descriptors, it works with all values in Swift, and not just with NSObject. In addition, because we don't rely on runtime programming, the compiler can optimize our code much better.

One drawback of the function-based approach is that functions are opaque. We can take an NSSortDescriptor and print it to the console, and we get some information about the sort descriptor: the key path, the selector name, and the sort order. We can even serialize and deserialize an NSSortDescriptor using NSSecureCoding. Our function-based approach can't do this.

Yet the approach of using functions as data — storing them in arrays and building those arrays at runtime — opens up a new level of dynamic behavior, and it's one way in which a statically typed compile-time-oriented language like Swift can replicate some of the dynamic behavior of languages like Objective-C or Ruby.

We also saw the usefulness of writing functions that combine other functions, which is one of the building blocks of functional programming. For example, our `then` method took two sort descriptors and combined them into a single sort descriptor. This is a very powerful technique with many different applications.

Functions as Delegates

Delegates. They're everywhere. Drummed into the heads of Objective-C (and Java) programmers is this message: use protocols (interfaces) for callbacks. You define a protocol, your owner implements that protocol, and it registers itself as your delegate so that it gets callbacks.

If a delegate protocol contains only a single method, you can mechanically replace the property storing the delegate object with one that stores the callback function directly. However, there are a number of tradeoffs to keep in mind.

Delegates, Cocoa Style

Let's start by creating a protocol in the same way that Cocoa defines its countless delegate protocols. Most programmers who come from Objective-C have written code like this many times over:

```
protocol AlertViewDelegate: AnyObject {  
    func buttonTapped(atIndex: Int)  
}
```

AlertViewDelegate is defined as a class-only protocol (by inheriting from AnyObject) because we want our AlertView class to hold a weak reference to the delegate. This way, we don't have to worry about reference cycles. An AlertView will never strongly retain its delegate, so even if the delegate (directly or indirectly) has a strong reference to the alert view, all is well. If the delegate is deinitialized, the delegate property will automatically become nil:

```
class AlertView {  
    var buttons: [String]  
    weak var delegate: AlertViewDelegate?  
  
    init(buttons: [String] = ["OK", "Cancel"]) {  
        self.buttons = buttons  
    }  
  
    func fire() {
```

```
    delegate?.buttonTapped(atIndex: 1)
}
}
```

This pattern works really well when we're dealing with classes. For example, suppose we have a `ViewController` class that initializes the alert view and sets itself as the delegate. Because the delegate is marked as weak, we don't need to worry about reference cycles:

```
class ViewController: AlertViewDelegate {
    let alert: AlertView

    init() {
        alert = AlertView(buttons: ["OK", "Cancel"])
        alert.delegate = self
    }

    func buttonTapped(atIndex index: Int) {
        print("Button tapped: \(index)")
    }
}
```

It's common practice to always mark delegate properties as weak. This convention makes reasoning about the memory management very easy, because classes that implement the delegate protocol don't have to worry about creating a reference cycle.

Functions Instead of Delegates

If the delegate protocol only has a single method defined, we can replace the delegate property with a property that stores the callback function directly. In our case, this could be an optional `buttonTapped` property, which is `nil` by default:

```
class AlertView {
    var buttons: [String]
    var buttonTapped: ((_ buttonIndex: Int) -> ())?

    init(buttons: [String] = ["OK", "Cancel"]) {
        self.buttons = buttons
    }

    func fire() {
```

```
    buttonTapped?(1)
}
}
```

The `(_ buttonIndex: Int) -> ()` notation for the function type is a little weird because the internal name, `buttonIndex`, isn't relevant anywhere else in the code. We mentioned above that function types unfortunately can't have parameter labels; they *can*, however, have an explicit blank parameter label combined with an internal argument name. This is the officially sanctioned workaround to give parameters in function types labels for documentation purposes until Swift supports a better way.

We can now create a logger struct and then create an alert view instance and a logger variable:

```
struct TapLogger {
    var taps: [Int] = []

    mutating func logTap(index: Int) {
        taps.append(index)
    }
}

let alert = AlertView()
var logger = TapLogger()
```

However, we can't simply assign the `logTap` method to the `buttonTapped` property. The Swift compiler tells us that “partial application of a ‘mutating’ method is not allowed”:

```
alert.buttonTapped = logger.logTap // Error
```

In the code above, it's not clear what should happen in the assignment. Does the logger get copied? Or should `buttonTapped` mutate the original variable (i.e. logger gets captured)?

To make this work, we have to wrap the right-hand side of the assignment in a closure. This has the benefit of making it very clear that we're now capturing the original logger variable (not the value), and that we're mutating it:

```
alert.buttonTapped = { logger.logTap(index: $0) }
```

As an additional benefit, the naming is now decoupled: the callback property is called `buttonTapped`, but the function that implements it is called `logTap`. Rather than a method, we could also specify an anonymous function:

```
alert.buttonTapped = { print("Button \"\$0\" was tapped") }
```

When combining callbacks with classes, there are some caveats. Let's go back to our view controller example. In its initializer, instead of assigning itself as the alert view's delegate, the view controller can now assign its `buttonTapped` method to the alert view's callback handler:

```
class ViewController {
    let alert: AlertView

    init() {
        alert = AlertView(buttons: ["OK", "Cancel"])
        alert.buttonTapped = self.buttonTapped(atIndex:)
    }

    func buttonTapped(atIndex index: Int) {
        print("Button tapped: \(index)")
    }
}
```

The `alert.buttonTapped = self.buttonTapped(atIndex:)` line looks like an innocent assignment, but beware: we've just created a reference cycle! Every reference to an instance method of an object (like `self.buttonTapped` in the example) implicitly captures the object. To see why this has to be the case, consider the perspective of the alert view: when the alert view calls the callback function that's stored in its `buttonTapped` property, the function must somehow "know" which object's instance method it needs to call — it's not enough to just store a reference to `ViewController.buttonTapped(atIndex:)` without knowing the instance.

We could've shortened `self.buttonTapped(atIndex:)` to `self.buttonTapped` or just `buttonTapped`; all three refer to the same function. Parameter labels can be omitted as long as doing so doesn't create ambiguities.

To avoid a strong reference, it's often necessary to wrap the method call in another closure that captures the object weakly:

```
alert.buttonTapped = { [weak self] index in
    self?.buttonTapped(atIndex: index)
}
```

This way, the alert view doesn't have a strong reference to the view controller. If we can guarantee that the alert view's lifetime is tied to the view controller, another option is to use unowned instead of weak. With weak, should the alert view outlive the view controller, self will be nil inside the closure when the function gets called.

As we've seen, there are definite tradeoffs between protocols and callback functions. A protocol adds some verbosity, but a class-only protocol with a weak delegate removes the need to worry about introducing reference cycles. Replacing the delegate with a function adds a lot of flexibility and allows you to use structs and anonymous functions. However, when dealing with classes, you need to be careful not to introduce a reference cycle.

Also, when you need multiple callback functions that are closely related (for example, when providing the data for a table view), it can be helpful to keep them grouped together in a protocol rather than having individual callbacks. On the flipside, when using a protocol, a single type *has* to implement all the methods.

To unregister a delegate or a function callback, we can set it to nil. What about when our type stores an array of delegates or callbacks? With class-based delegates, we can remove an object from the delegate list. With callback functions, this isn't so simple; we'd need to add extra infrastructure for unregistering, because functions can't be compared.

inout Parameters and Mutating Methods

The “`&`” we use at the front of an inout argument in Swift might give you the impression — especially if you have a C or C++ background — that inout parameters are essentially pass-by-reference. But they aren't. inout is pass-by-value-and-copy-back, *not* pass-by-reference. To quote *The Swift Programming Language*:

An in-out parameter has a value that's passed *in* to the function, is modified by the function, and is passed back *out* of the function to replace the original value.

To understand what kind of expressions can be passed as an inout parameter, we need to make the distinction between lvalues and rvalues. An *lvalue* describes a memory location. *lvalue* is short for “left value,” because lvalues are expressions that can appear on the left side of an assignment. For example, `array[0]` is an lvalue, as it describes the memory location of the first element in the array. An *rvalue* describes a value. `2 + 2` is an rvalue, as it describes the value 4. You can’t put `2 + 2` or `4` on the left side of an assignment statement.

For inout parameters, you can only pass lvalues, because it doesn’t make sense to mutate an rvalue. When you’re working with inout parameters in regular functions and methods, you need to be explicit about passing them in: every lvalue needs to be prefixed with an `&`. For example, when we call the `increment` function (which takes an inout `Int`), we can pass in a variable by prefixing it with an ampersand:

```
func increment(value: inout Int) {
    value += 1
}
var i = 0
increment(value: &i)
```

If we define a variable using `let`, we can’t use it as an lvalue. This makes sense, because we’re not allowed to mutate `let` variables; we can only use “mutable” lvalues:

```
let y: Int = 0
increment(value: &y) // Error
```

In addition to variables, a few more things are also lvalues. For example, we can also pass in an array subscript (if the array is defined using `var`):

```
var array = [0, 1, 2]
increment(value: &array[0])
array // [1, 1, 2]
```

In fact, this works with every subscript (including your own custom subscripts), as long as they have both a get and a set defined. Likewise, we can use properties as lvalues, but again, only if they have both get and set defined:

```
struct Point {
    var x: Int
    var y: Int
}
```

```
var point = Point(x: 0, y: 0)
increment(value: &point.x)
point // Point(x: 1, y: 0)
```

If a property is read-only (that is, only `get` is available), we can't use it as an `inout` parameter:

```
extension Point {
    var squaredDistance: Int {
        return x*x + y*y
    }
}
increment(value: &point.squaredDistance) // Error
```

Operators can also take an `inout` value, but for the sake of simplicity, they don't require the ampersand when called; we just specify the lvalue. For example, let's add back the postfix increment operator, which was removed in Swift 3:

```
postfix func ++(x: inout Int) {
    x += 1
}
point.x++
point // Point(x: 2, y: 0)
```

A mutating operator can even be combined with optional chaining. Here, we chain the increment operation to a dictionary subscript access:

```
var dictionary = ["one": 1]
dictionary["one"]?++
dictionary["one"] // Optional(2)
```

Note that the `++` operator won't get executed if the key lookup returns `nil`.

The compiler may optimize an `inout` variable to pass-by-reference rather than copying in and out. However, it's explicitly stated in the documentation that we shouldn't rely on this behavior.

We'll get back to `inout` next, in the [Structs and Classes](#) chapter, where we'll look at the similarities between mutating methods and functions that take an `inout` parameter.

Nested Functions and inout

You can use an inout parameter inside nested functions, and Swift will make sure your usage is safe. For example, you can define a nested function (either using `func` or using a closure expression) and safely mutate an inout parameter:

```
func incrementTenTimes(value: inout Int) {
    func inc() {
        value += 1
    }
    for _ in 0..<10 {
        inc()
    }
}

var x = 0
incrementTenTimes(value: &x)
x // 10
```

However, you're not allowed to let that inout parameter escape (we'll talk more about escaping functions later in this chapter):

```
func escapeIncrement(value: inout Int) -> () -> () {
    func inc() {
        value += 1
    }
    // Error: nested function cannot capture inout parameter
    // and escape.
    return inc
}
```

This makes sense, given that the inout value is copied back just before the function returns. If we could somehow modify it later, what should happen? Should the value get copied back at some point? What if the source no longer exists? Having the compiler verify this is critical for safety.

When & Doesn't Mean inout

Speaking of unsafe functions, you should be aware of the other meaning of `&`: converting a function argument to an unsafe pointer.

If a function takes an `UnsafeMutablePointer` as a parameter, then you can pass a var into it using `&`, similar to how you would with an `inout` argument. But here, you *really are* passing by reference — by pointer in fact.

Here's `increment`, written to take an unsafe mutable pointer instead of an `inout`:

```
func incref(pointer: UnsafeMutablePointer<Int>) -> () -> Int {  
    // Store a copy of the pointer in a closure.  
    return {  
        pointer.pointee += 1  
        return pointer.pointee  
    }  
}
```

As we'll discuss in later chapters, Swift arrays implicitly decay to pointers to make C interoperability nice and painless. Now, suppose you pass in an array that goes out of scope before you call the resulting function:

```
let fun: () -> Int  
do {  
    var array = [0]  
    fun = incref(pointer: &array)  
}  
/*_*/fun()
```

This opens up an exciting world of undefined behavior. In testing, the above code printed different values on each run: sometimes 0, sometimes 1, sometimes 140362397107840 — and sometimes it produced a runtime crash.

The moral here is: know what you're passing in to. When appending an `&`, you could be invoking nice safe Swift `inout` semantics, or you could be casting your poor variable into the brutal world of unsafe pointers. When dealing with unsafe pointers, be very careful about the lifetime of variables. We'll go into more detail on this in the [Interoperability](#) chapter.

Subscripts

We've already seen subscripts in the standard library. For example, we can perform a dictionary lookup like so: `dictionary[key]`. These subscripts are very much a hybrid of

functions and properties, with their own special syntax. Like functions, they take arguments. Like computed properties, they can be either read-only (using `get`) or read-write (using `get set`). Just like normal functions, we can overload them by providing multiple variants with different types — something that isn't possible with properties. For example, arrays have two subscripts by default — one to access a single element, and one to get at a slice (to be precise, these are declared in the Collection protocol):

```
let fibs = [0, 1, 1, 2, 3, 5]
let first = fibs[0] // 0
fibs[1..3] // [1, 1]
```

Custom Subscripts

We can add subscripting support to our own types, and we can also extend existing types with new subscript overloads. As an example, let's define a Collection subscript that takes a list of indices and returns an array of all elements at those indices:

```
extension Collection {
    subscript(indices indexList: Index...) -> [Element] {
        var result: [Element] = []
        for index in indexList {
            result.append(self[index])
        }
        return result
    }
}
```

Note how we used an explicit parameter label to disambiguate our subscript from those in the standard library. The three dots indicate that `indexList` is a *variadic parameter*. The caller can pass zero or more comma-separated values of the specified type (here, the collection's `Index` type). Inside the function, the parameters are made available as an array.

We can use the new subscript like this:

```
Array("abcdefghijklmnopqrstuvwxyz")[indices: 7, 4, 11, 11, 14]
// ["h", "e", "l", "l", "o"]
```

Advanced Subscripts

Subscripts aren't limited to a single parameter. We've already seen an example of a subscript that takes more than one parameter: the dictionary subscript that takes a key and a default value. Check out [its implementation](#) in the Swift source code if you're interested.

Subscripts can also be generic in their parameters or their return type. Consider a heterogeneous dictionary of type [String: Any]:

```
var japan: [String: Any] = [  
    "name": "Japan",  
    "capital": "Tokyo",  
    "population": 126_440_000,  
    "coordinates": [  
        "latitude": 35.0,  
        "longitude": 139.0  
    ]  
]
```

If you want to mutate a nested value in this dictionary, e.g. the coordinate's latitude, you'll find it isn't so easy:

```
// Error: Type 'Any' has no subscript members.  
japan["coordinates"]?["latitude"] = 36.0
```

OK, that's understandable. The expression `japan["coordinate"]` has the type Any?, so you'd probably try to cast it to a dictionary before applying the nested subscript:

```
// Error: Cannot assign to immutable expression.  
(japan["coordinates"] as? [String: Double])?["latitude"] = 36.0
```

Alas, not only is this becoming ugly fast, but it doesn't work either. The problem is that you can't mutate a variable through a typecast — the expression `japan["coordinates"] as? [String: Double]` is no longer an lvalue. You'd have to store the nested dictionary in a local variable first, then mutate that variable, and then assign the local variable back to the top-level key.

We can do better by extending Dictionary with a generic subscript that takes the desired target type as a second parameter and attempts the cast inside the subscript implementation:

```

extension Dictionary<Result> {
    subscript<Result>(key: Key, as type: Result.Type) -> Result? {
        get {
            return self[key] as? Result
        }
        set {
            // Delete existing value if caller passed nil.
            guard let value = newValue else {
                self[key] = nil
                return
            }
            // Ignore if types don't match.
            guard let value2 = value as? Value else {
                return
            }
            self[key] = value2
        }
    }
}

```

Since we no longer have to downcast the value returned by the subscript, the mutation operation goes through to the top-level dictionary variable:

```

japan["coordinates", as: [String: Double].self]?["latitude"] = 36.0
japan["coordinates"] // Optional(["latitude": 36.0, "longitude": 139.0])

```

It's nice that generic subscripts make this possible, but you'll notice the final syntax in this example is still quite ugly. Swift is generally not well suited for working on heterogeneous collections like this dictionary. In most cases, you'll be better off defining your own custom types for your data (e.g. here, a `Country` struct) and conforming those types to `Codable` for converting values to and from data transfer formats.

Autoclosures

We're all familiar with how the logical AND operator `&&` evaluates its arguments. It first evaluates its left operand and immediately returns if that evaluation yields false. Only if the left operand evaluates to true is the right operand evaluated. After all, if the left operand evaluates to false, there's no way the entire expression can evaluate to true.

This behavior is called *short circuiting*. For example, if we want to check if a condition holds for the first element of an array, we could write the following code:

```
let evens = [2,4,6]
if !evens.isEmpty && evens[0] > 10 {
    // Perform some work.
}
```

In the snippet above, we rely on short circuiting: the array lookup happens only if the first condition holds. Without short circuiting, this code would crash on an empty array.

A better way to write this particular example is to use an if let binding:

```
if let first = evens.first, first > 10 {
    // Perform some work.
}
```

This is another form of short circuiting: the second condition is only evaluated if the first one succeeds.

In almost all languages, short circuiting for the `&&` and `||` operators is built into the language. However, it's often not possible to define your own operators or functions that have the same behavior. If a language supports first-class functions, we can fake short circuiting by providing an anonymous function instead of a value. For example, let's say we wanted to define an `and` function in Swift to have the same behavior as the `&&` operator:

```
func and(_ l: Bool, _ r: () -> Bool) -> Bool {
    guard l else { return false }
    return r()
}
```

The function above first checks the value of `l` and returns `false` if `l` evaluates to `false`. Only if `l` is `true` does it return the value that comes out of the closure `r`. However, using it is a little bit uglier than using the `&&` operator because the right operand now has to be a function:

```
if and(!evens.isEmpty, { evens[0] > 10 }) {
    // Perform some work.
}
```

Swift has a nice feature to make this prettier. We can use the `@autoclosure` attribute to tell the compiler that it should wrap a particular argument in a closure expression. The definition of `and` is almost the same as above, except for the added `@autoclosure` annotation:

```
func and(_ l: Bool, _ r: @autoclosure () -> Bool) -> Bool {  
    guard l else { return false }  
    return r()  
}
```

However, the usage of `and` is now much simpler, as we don't need to wrap the second parameter in a closure. We can just call it as if it took a regular `Bool` argument, and the compiler transparently wraps the argument in a closure expression:

```
if and(!events.isEmpty, events[0] > 10) {  
    // Perform some work.  
}
```

This allows us to define our own functions and operators with short-circuiting behavior. For example, operators like `??` and `!?` (as defined in the [Optionals chapter](#)) are now straightforward to write. In the standard library, functions like `assert` and `precondition` also use autoclosures to only evaluate arguments when really needed. By deferring the evaluation of assertion conditions from the call sites to the body of the `assert` function, these potentially expensive operations can be stripped completely in optimized builds where they're not needed.

Autoclosures can also come in handy when writing logging functions. For example, here's how you could write your own `log` function, which only evaluates the log message if the condition is true:

```
func log(ifFalse condition: Bool,  
        message: @autoclosure () -> (String),  
        file: String = #fileID, function: String = #function, line: Int = #line)  
{  
    guard !condition else { return }  
    print("Assertion failed: \(message()), \(file):\((function)) (\(line)\(\(line)))")  
}
```

This means you can perform expensive computations in the expression you pass as the `message` argument without incurring the evaluation cost if the value isn't used. The `log` function also uses the debugging identifiers `#fileID`, `#function`, and `#line`. They're

especially useful when passed as default arguments to a function, because they'll receive the values of the file name, function name, and line number at the call site.

Use autoclosures sparingly, though. Their behavior violates normal expectations — for example, if a side effect of an expression isn't executed because the expression is wrapped in an autoclosure. To quote Apple's Swift book:

Overusing autoclosures can make your code hard to understand. The context and function name should make it clear that evaluation is being deferred.

The @escaping Annotation

You might have noticed that the compiler requires you to be explicit about accessing `self` in some closure expressions but not in others. For example, we need to use an explicit `self` in the completion handler of a network request, whereas we don't need to be explicit about `self` in closures passed to `map` or `filter`. The difference between the two is whether the closure is being stored for later use (as with the network request), or if it's only used synchronously within the scope of the function (as with `map` and `filter`).

If a closure is stored somewhere (e.g. in a property) to be called later, it's said to be *escaping*. Conversely, closures that never leave a function's local scope are *non-escaping*. With escaping closures, the compiler forces us to be explicit about using `self` in closure expressions, because unintentionally capturing `self` strongly is one of the most frequent causes of reference cycles. A non-escaping closure can't create a permanent reference cycle because it's automatically destroyed when the function it's defined in returns.

Closure arguments are non-escaping by default. If you want to store a closure for later use, you need to mark the closure argument as `@escaping`. The compiler will verify this: unless you mark the closure argument as `@escaping`, it won't allow you to store the closure (or return it to the caller, for example).

In the sort descriptors example, there were multiple function parameters that required the `@escaping` attribute:

```
extension SortDescriptor {  
    init<Value: Comparable>(_ key: @escaping (Root) -> Value) {  
        self.areInIncreasingOrder = { key($0) < key($1) }  
    }  
}
```

Note that the non-escaping-by-default rule only applies to function parameters, and then only for function types in *immediate parameter position*. This means stored properties that have a function type are always escaping (which makes sense). Surprisingly, the same is true for functions that *are* used as parameters but are wrapped in some other type, such as a tuple or an optional. Since the closure is no longer an *immediate* parameter in this case, it automatically becomes escaping. As a consequence, you can't write a function that takes a function argument where the parameter is both optional and non-escaping. In many situations, you can avoid making the argument optional by providing a default value for the closure. If that's not possible, a workaround is to use overloading to write two variants of the function — one with an optional (escaping) function parameter, and one with a non-optional, non-escaping parameter:

```
func transform(_ input: Int, with f: ((Int) -> Int)?) -> Int {  
    print("Using optional overload")  
    guard let f = f else { return input }  
    return f(input)  
}  
  
func transform(_ input: Int, with f: (Int) -> Int) -> Int {  
    print("Using non-optional overload")  
    return f(input)  
}
```

This way, calling the function with a `nil` argument (or a variable of an optional type) will use the optional variant, whereas passing a literal closure expression will invoke the non-escaping, non-optional overload:

```
_ = transform(10, with: nil) // Using optional overload  
_ = transform(10) { $0 * $0 } // Using non-optional overload
```

withoutActuallyEscaping

You may run into a situation where you *know* that a closure doesn't escape but the compiler can't *prove* it, forcing you to add an `@escaping` annotation. To illustrate this, let's look at an example from the standard library documentation. We write a custom implementation of the `allSatisfy` method on `Array` that uses a *lazy view* of the array (not to be confused with lazy properties, which we discuss in the [Properties](#) chapter) internally. We then apply a filter to the lazy view and check whether any element got through the filter (i.e. whether at least one element didn't satisfy the predicate). Our first attempt results in a compile error:

```
extension Array {  
    func allSatisfy2(_ predicate:(Element) -> Bool) -> Bool {  
        // Error: Closure use of non-escaping parameter 'predicate'  
        // may allow it to escape.  
        return self.lazy.filter({ !predicate($0) }).isEmpty  
    }  
}
```

We'll say more about the lazy collection APIs in the [Collection Protocols](#) chapter. For now, it's enough to know that the lazy view stores subsequent transformations (e.g. the closure passed to filter) in an internal property to apply them later. This requires any closure that's passed in to be escaping, and this is the cause of the error, since our predicate parameter is non-escaping.

We could fix this by annotating the parameter with `@escaping`, but in this case, we *know* the closure won't escape because the lifetime of the lazy collection view is bound to the lifetime of the function. Swift provides an escape hatch for situations like this in the form of the `withoutActuallyEscaping` function. It allows you to pass a non-escaping closure to a function that expects an escaping one. This compiles and works correctly:

```
extension Array {  
    func allSatisfy2(_ predicate:(Element) -> Bool) -> Bool {  
        return withoutActuallyEscaping(predicate) { escapablePredicate in  
            self.lazy.filter { !escapablePredicate($0) }.isEmpty  
        }  
    }  
}  
  
let areAllEven = [1,2,3,4].allSatisfy2 { $0 % 2 == 0 } // false  
let areAllOneDigit = [1,2,3,4].allSatisfy2 { $0 < 10 } // true
```

Note that the lazy implementation isn't any more efficient than the standard library's implementation of `allSatisfy`, which uses a simple `for` loop and doesn't require `withoutActuallyEscaping`. The lazy implementation only serves to demonstrate the case where we know that a closure doesn't escape but the compiler can't prove it.

Be aware that you're entering unsafe territory by using `withoutActuallyEscaping`. Allowing the copy of the closure to escape from the call to `withoutActuallyEscaping` results in undefined behavior.

Result Builders

Result builders are special kinds of functions that allow us to build up result values from multiple statements using a concise and expressive syntax.

The most prominent example, and arguably the motivation for this Swift feature, is SwiftUI's view builder syntax. Using view builders, you can define the contents of e.g. a horizontal stack view like this:

```
HStack {  
    Text("Finish the Advanced Swift Update")  
    Spacer()  
    Button("Complete") { /* ... */}  
}
```

The trailing closure of the `HStack` is a result builder function, although it's not annotated in any special way. The annotation — `@ViewBuilder` — can be found in the `HStack`'s initializer, which we simplified slightly for the sake of this example:

```
struct HStack<Content>: View where Content: View {  
    public init(  
        alignment: VerticalAlignment = .center,  
        spacing: CGFloat? = nil,  
        @ViewBuilder content: () -> Content)  
    // ...  
}
```

In the view builder above, we wrote three expressions on separate lines. In normal Swift code, this wouldn't make any sense: we're not doing anything with the values of these expressions, and they don't have any side effects either. Within a result builder function though, the compiler rewrites this code to produce a composite value from all the statements in the function. To do so, it uses static methods defined on the corresponding result builder type, like `ViewBuilder` in this example.

To rewrite the example above, the compiler uses the static `buildBlock` method on `ViewBuilder`, which accepts three parameters conforming to the `View` protocol:

```
@resultBuilder public struct ViewBuilder {  
    // ...  
    public static func buildBlock<C0, C1, C2>(_ c0: C0, _ c1: C1, _ c2: C2)
```

```
-> TupleView<(C0, C1, C2)>
where C0: View, C1: View, C2: View
// ...
}
```

The rewritten code, which no longer relies on result builders, looks like this:

```
HStack {
    return ViewBuilder.buildBlock(
        Text("Finish the Advanced Swift Update"),
        Spacer(),
        Button("Complete") {/* ... */}
    )
}
```

Result builder types are marked with `@resultBuilder` and can implement a variety of different static `build...` methods, which we'll examine in more detail in the remainder of this section. Which of these methods are implemented determines which kinds of expressions and statements can be used in the result builder function.

Blocks and Expressions

The most basic builder methods are `buildBlock` and `buildExpression`. Like all builder methods, they're static methods. The builder type just serves as a namespace and never gets instantiated.

Implementing at least one `buildBlock` method is the only formal requirement for a `@resultBuilder` type. Most of the time, you'll implement more than one variant of `buildBlock`, differing in the number and perhaps types of the parameters, since this allows combining of multiple partial results into one result value. For example, the view builder's `buildBlock` method comes in variants from zero to ten parameters, allowing for zero to ten views to be combined into a `TupleView` in a view builder function.

SwiftUI's `buildBlock` methods only accept `View` values as arguments, but it's not a general requirement for `buildBlock` to limit itself to one argument type. We can also write multiple overloads with different parameter types, although implementing `buildExpression` is often a more elegant way to support multiple types in the builder function.

`buildExpression` isn't a requirement for builder types, but it can be very useful to support different types of expressions. When you do implement one or more variants of this method, Swift will apply `buildExpression` to every expression in the builder function first, before passing the partial results to `buildBlock`.

The `buildExpression` method takes one parameter, but you can implement multiple variants to support different parameter types. The return value can be of any type, as long as that type is supported by some `buildBlock` method.

Before we look into the other `build...` methods that can be implemented on result builder types, we'll use `buildBlock` and `buildExpression` and create our own result builder type. As an example, we'll implement a result builder for building strings. In its simplest form, the string builder type looks like this:

```
@resultBuilder
struct StringBuilder {
    static func buildBlock() -> String {
        ""
    }
}
```

By implementing `buildBlock` with no parameters, we support empty string builder functions, like this one:

```
@StringBuilder func build() -> String {
}

build() //
```

The result of executing this function is the empty string. Under the hood, the compiler rewrites the `build` function to:

```
func build_rewritten() -> String {
    StringBuilder.buildBlock()
}
```

To support building up a string from multiple strings, we add a `buildBlock` variant that takes a variadic number of string arguments (a variadic argument list also accepts zero arguments, so this can replace the previous no-parameter method):

```
static func buildBlock(_ strings: String...) -> String {  
    strings.joined()  
}
```

The variadic parameter covers everything from zero to any number of string arguments with a single method. This works for our example because we expect all arguments to be the same type. Contrast this with SwiftUI's view builder, which provides separate overloads of `buildBlock` for different parameter counts and is thus artificially limited to ten views (because Apple decided to stop there). SwiftUI does it this way because each argument can be of a different type (each conforming to `View`) and the framework wants to preserve those types. We expect a future Swift release to support a form of variadic generics, which would enable SwiftUI (and others) to provide a variadic overload of `buildBlock` that accepted generic arguments.

Now we can write a string builder function like this:

```
@StringBuilder func greeting() -> String {  
    "Hello,"  
    "World!"  
}  
  
greeting() // Hello, World!
```

Swift translates this string builder function into this code:

```
func greeting_rewritten() -> String {  
    StringBuilder.buildBlock(  
        "Hello,"  
        "World!"  
    )  
}
```

Next, we can add support for other types — integers, for example — by implementing `buildExpression`:

```
static func buildExpression(_ s: String) -> String {  
    s  
}
```

```
static func buildExpression(_ x: Int) -> String {  
    ""  
}
```

Note that we have to implement `buildExpression` for all types we want to support as expressions — not just the ones for which no suitable `buildBlock` method is available. Now we can seamlessly mix strings and integers to construct a string result:

```
let planets = [  
    "Mercury", "Venus", "Earth", "Mars", "Jupiter",  
    "Saturn", "Uranus", "Neptune"  
]
```

```
@StringBuilder func greetEarth() -> String {  
    "Hello, Planet "  
    planets.firstIndex(of: "Earth")!  
    "!"  
}
```

```
greetEarth() // Hello, Planet 2!
```

Here's the rewritten code for this example:

```
func greetEarth_rewritten() -> String {  
    StringBuilder.buildBlock(  
        StringBuilder.buildExpression("Hello, Planet "),  
        StringBuilder.buildExpression(planets.firstIndex(of: "Earth")!),  
        StringBuilder.buildExpression("!")  
    )  
}
```

Overloading Builder Methods

As we saw above, we can write multiple overloads for methods like `buildBlock` or `buildExpression`. Mostly, we provide overloads to support different types in the builder function. For example, we implemented multiple `buildExpression` methods with different parameter types to support strings and integers in the string builder, and SwiftUI implements multiple variants of `buildBlock` to support composing different

numbers of views. However, overloads can also be used to enable some other useful features.

For example, by default, it isn't possible to use a `fatalError` statement in a builder function, because it's seen as an expression with type `Never`. Implementing a variant of `buildExpression` for the `Never` type fixes this problem (unfortunately, it's not very practical because the compiler will flag this implementation with a non-suppressible "Will never be executed" warning, which you may not want to have in your code):

```
static func buildExpression(_ x: Never) -> String {  
    fatalError()  
}
```

Similarly, we can't write `print` statements in a builder function, because `print` has a `Void` return type. To enable `print` statements, we can add a variant of `buildExpression` that accepts a `void` parameter without affecting the result that's being built up:

```
static func buildExpression(_ x: Void) -> String {  
    ""  
}
```

Furthermore, overloads of `buildExpression` can also be used to provide clearer compiler diagnostics. For example, we can add a second, more generic variant of `buildExpression` to provide a clear error message for values of unsupported types:

```
@available(*, unavailable,  
    message: "String Builders only support string and integer values")  
static func buildExpression<A>(_ expression: A) -> String {  
    ""  
}
```

Since the existing overloads for `String` and `Int` are more specific than this generic function, the compiler will only use this "error" variant for types we don't want to support. In this case, the `@available` attribute marks this implementation as unavailable and specifies a custom error message.

Conditionals

So far, our string builder is more or less a different syntax for string interpolation (which we discuss in the [Strings](#) chapter). By adding the `buildIf` and `buildEither` methods to the

builder type, we can extend its capabilities and support if, if ... else, and switch statements.

We start by implementing the `buildIf` method, which enables support for simple if statements without an else branch. The parameter of `buildIf` is an optional value, which will be `nil` if the condition didn't hold:

```
static func buildIf(_ s: String?) -> String {  
    s ?? ""  
}
```

This allows us to rewrite our previous example like this:

```
@StringBuilder func greet(planet: String) -> String {  
    "Hello, Planet"  
    if let idx = planets.firstIndex(of: planet) {  
        ""  
        idx  
    }  
    "!"  
}  
  
greet(planet: "Earth") // Hello, Planet 2!  
greet(planet: "Sun") // Hello, Planet!
```

Now that we've introduced conditions to the builder function, the rewritten version starts to become more interesting:

```
func greet_rewritten(planet: String) -> String {  
    let v0 = "Hello, Planet"  
    var v1: String?  
    if let idx = planets.firstIndex(of: planet) {  
        v1 = StringBuilder.buildBlock(  
            StringBuilder.buildExpression(" "),  
            StringBuilder.buildExpression(idx)  
        )  
    }  
    let v2 = StringBuilder.buildIf(v1)  
    return StringBuilder.buildBlock(v0, v2)  
}
```

First, an optional variable is declared for the partial result of the condition (the variable names – v0, v1, etc. – are generic names we picked for the sake of this example). Swift then rewrites the statements within the condition in the same way we saw before: buildExpression is called for each expression, and then buildBlock is called with all the partial results from the buildExpression calls. Finally, buildIf is called with the partial result of the condition.

To handle invalid inputs to the greeting function better, support for if ... else would be really helpful. For this, we have to implement the buildEither(first:) and buildEither(second:) builder methods.

buildEither(first:) will get called with the result of the first branch, and buildEither(second:) will get called with the result of the second branch. If there are multiple chained if ... else statements, the compiler represents them as nested calls of buildEither. For example, consider a condition like this:

```
if x == 0 {  
    // ...  
} else if x < 10 {  
    // ...  
} else {  
    // ...  
}
```

We can write the same statement as follows:

```
if x == 0 {  
    // ...  
} else {  
    if x < 10 {  
        // ...  
    } else {  
        // ...  
    }  
}
```

Both if statements have one else branch now, and they'll be rewritten using the first and second variants of buildEither.

For the string builder example, the implementation of buildEither is extremely simple, since we only need to return the partial result from the first or second branch:

```
static func buildEither(first component: String) -> String {
    component
}

static func buildEither(second component: String) -> String {
    component
}
```

In other use cases, these two variants could be used to distinguish how the result is being transformed in the if or else branch, e.g. to preserve information about which branch got executed.

With added support for if ... else (and switch at the same time), we can extend our example:

```
@StringBuilder func greet2(planet: String) -> String {
    "Hello,"
    if let idx = planets.firstIndex(of: planet) {
        switch idx {
            case 2:
                "World"
            case 1, 3:
                "Neighbor"
            default:
                "planet "
                idx + 1
        }
    } else {
        "unknown planet"
    }
    "!"
}

greet2(planet: "Earth") // Hello, World!
greet2(planet: "Mars") // Hello, Neighbor!
greet2(planet: "Jupiter") // Hello, planet 5!
greet2(planet: "Pluto") // Hello, unknown planet!
```

The code, as rewritten by the Swift compiler for the greeting function above, already becomes quite unwieldy:

```
func greet2_rewritten(planet: String) -> String {
    let v0 = StringBuilder.buildExpression("Hello, ")
    let v1: String
    if let idx = planets.firstIndex(of: planet) {
        let v1_0: String
        switch idx {
            case 2:
                v1_0 = StringBuilder.buildEither(first:
                    StringBuilder.buildBlock(StringBuilder.buildExpression("World")))
            )
            case 1, 3:
                v1_0 = StringBuilder.buildEither(second:
                    StringBuilder.buildEither(first:
                        StringBuilder.buildBlock(StringBuilder.buildExpression("Neighbor")))
                )
            )
        default:
            let v1_0_0 = StringBuilder.buildExpression("planet")
            let v1_0_1 = StringBuilder.buildExpression(idx + 1)
            v1_0 = StringBuilder.buildEither(second:
                StringBuilder.buildEither(second:
                    StringBuilder.buildBlock(v1_0_0, v1_0_1)
                )
            )
        }
        v1 = StringBuilder.buildEither(first: v1_0)
    } else {
        v1 = StringBuilder.buildEither(second:
            StringBuilder.buildBlock(
                StringBuilder.buildExpression("unknown planet")
            )
        )
    }
    let v2 = StringBuilder.buildExpression("!")
    return StringBuilder.buildBlock(v0, v1, v2)
}
```

Loops

There's one more statement we can enable in result builder functions: `for...in` loops. To allow for loops, we have to add the `buildArray` method to our builder type:

```
static func buildArray(_ components: [String]) -> String {  
    components.joined(separator: "")  
}
```

This allows us to write loops like this:

```
@StringBuilder func greet3(planet: String?) -> String {  
    "Hello "  
    if let p = planet {  
        p  
    } else {  
        for p in planets.dropLast() {  
            "\((p), "  
        }  
        "and \(planets.last)!"  
    }  
}  
  
greet3(planet: nil)  
// Hello Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune!
```

Swift takes the partial results of each iteration of the loop and collects them in an array. This array then gets passed into `buildArray` to construct the partial result for the entire loop:

```
func greet3_rewritten(planet: String?) -> String {  
    let v0 = StringBuilder.buildExpression("Hello ")  
    let v1: String  
    if let p = planet {  
        v1 = StringBuilder.buildBlock(StringBuilder.buildExpression(p))  
    } else {  
        var v1_0: [String] = []  
        for p in planets.dropLast() {  
            let v1_0_0 = StringBuilder.buildBlock(  
                StringBuilder.buildExpression("\((p), ")  
            )  
        }  
    }  
}
```

```

        v1_0.append(v1_0_0)
    }
    let v1_1 = StringBuilder.buildArray(v1_0)
    let v1_2 = "and \(planets.last!)!"
    v1 = StringBuilder.buildBlock(v1_1, v1_2)
}
return StringBuilder.buildBlock(v0, v1)
}

```

Other Build Methods

Result builder types can implement two more methods that we can use to transform the result that has been built up. The first one, `buildLimitedAvailability`, can be used to transform the result type of a limited availability context (e.g. if `#available(...)`).

For example, SwiftUI uses this method within its view builder type to wrap the partial result from a limited availability context in a type-erasing `AnyView` wrapper. This is necessary because the result type of the limited availability context might contain types that aren't available outside of this context.

The last transformation method on result builder types is `buildFinalResult`: as the name suggests, this method is applied once to the result of the entire result builder function. One use case for this method is to hide internal types that have been used to build up partial results from the outside world.

For example, the string builder could use `[String]` as the internal type to build up the result, and then transform the array into a string in `buildFinalResult`. This means that all build methods will have a result type of `[String]`, except for `buildFinalResult`, which returns the finalized string:

```

@resultBuilder
struct StringBuilder {
    static func buildBlock(_ x: [String]...) -> [String] { x.flatMap { $0 } }
    static func buildIf(_ x: [String]?) -> [String] { x ?? [] }
    static func buildExpression(_ x: String) -> [String] { [x] }
    static func buildExpression(_ x: Int) -> [String] { ["\(x)"] }
    static func buildExpression(_ x: Never) -> [String] {}
    static func buildExpression(_ x: Void) -> [String] { [] }
    static func buildArray(_ x: [[String]]) -> [String] { x.flatMap { $0 } }
    static func buildEither(first x: [String]) -> [String] { x }
}

```

```
static func buildEither(second x: [String]) -> [String] { x }
static func buildFinalResult(_ x: [String]) -> String { x.joined() }
}
```

Unsupported Statements

We've seen above how you can enable certain kinds of statements in result builder functions, like `if`/`if let`, `if ... else`, `switch`, and `for ... in`. At the time of writing, almost all other statements — including `guard`, `defer`, `do ... catch`, `break`, and `continue` — are unsupported.

Recap

Functions are first-class objects in Swift, and treating functions as data can make our code more flexible. We saw how simple functions can replace certain kinds of runtime programming and delegates. We looked at mutating functions and `inout` parameters, subscripts (which really are a special kind of function), and the `@autoclosure` and `@escaping` attributes. Finally, we looked at result builders as a special kind of function, which enabled us to build return values with an expressive and concise syntax.

In the next chapter, we'll look at a closely related subject: properties. In the [Generics](#) and [Protocols](#) chapters, we'll discuss more ways to use functions in Swift to gain flexibility.

Properties

5

Properties in Swift come in two variants: stored properties and computed properties. Stored properties store values, whereas computed properties are similar to functions: they don't provide storage, rather they only provide a way to get and (optionally) set a value. In a way, you can think of computed properties like methods with a different syntax.

You can think of properties like variables that are defined on a type. Most of what we say in this chapter also applies to local and global variables. Variables can be stored or computed, have change observers, and use property wrappers. We consider properties a “special case” of variables, rather than the other way around.

There are two important features built on top of properties: key paths and property wrappers. Key paths are a way to reference the path of a property without referencing the value. More and more libraries adopt key paths as a way to write very concise, generic code, and we'll see some examples of that in this chapter. A property wrapper allows you to modify the behavior of a property with a very minimal syntax. Property wrappers were instrumental in providing SwiftUI with its lightweight syntax.

Let's look at the various ways to define properties. We'll start with a struct that represents a GPS track. It stores all the recorded points in an array called `record`, which is a *stored property*:

```
import CoreLocation

struct GPSTrack {
    var record: [(CLLocation, Date)] = []
}
```

If we want to make the `record` property available as read-only to the outside but read-write internally, we can use the `private(set)` or `fileprivate(set)` modifiers:

```
struct GPSTrack {
    private(set) var record: [(CLLocation, Date)] = []
}
```

To access all the timestamps in a GPS track, we create a *computed property*:

```
extension GPSTrack {
    /// Returns all the timestamps for the GPS track.
    /// - Complexity: O(*n*), where *n* is the number of points recorded.
    var timestamps: [Date] {
```

```
        return record.map { $0.1 }
    }
}
```

Because we didn't specify a setter, the `timestamps` property is read-only. The result isn't cached; each time you access the property, it computes the result. The [Swift API Design Guidelines](#) recommend you document the complexity of every computed property that isn't $O(1)$, because callers might assume that accessing a property is cheap.

Change Observers

We can also implement the `willSet` and `didSet` handlers for stored properties and variables to be called every time a property is set (even if the value doesn't change). These are called immediately before and after the new value is stored, respectively. One useful case is when a view needs to lay itself out again based on certain properties. By calling `setNeedsLayout` in `didSet` we can be sure this will always happen. (In the section on property wrappers, we'll look at an even shorter way to do this.)

```
class MyView: UIView {
    var pageSize: CGSize = CGSize(width: 800, height: 600) {
        didSet {
            self.setNeedsLayout()
        }
    }
}
```

The observers have to be defined at the declaration site of a property — you can't add one retroactively in an extension. Therefore, they're a tool for the *designer* of the type, and not the user. The `willSet` and `didSet` handlers are essentially shorthand for defining a pair of properties: one private stored property that provides the storage, and a public computed property whose setter performs additional work before and/or after storing the value in the stored property. This is fundamentally different from the [key-value observing](#) mechanism in Foundation, which is often used by *consumers* of an object to observe internal changes, whether or not the class's designer intended this.

You can, however, override a property in a subclass to add an observer. Here's an example:

```
class Robot {
```

```

enum State {
    case stopped, movingForward, turningRight, turningLeft
}
var state = State.stopped
}

class ObservableRobot: Robot {
    override var state: State {
        willSet {
            print("Transitioning from \(state) to \(newValue)")
        }
    }
}

var robot = ObservableRobot()
robot.state = .movingForward // Transitioning from stopped to movingForward

```

This is still consistent with the nature of change observers as an internal characteristic of a type. If it weren't allowed, a subclass could achieve the same effect by overriding a stored property with a computed setter that performs additional work.

The difference in usage is reflected in the implementation of these features. KVO uses the Objective-C runtime to dynamically add an observer to a class's setter, which would be impossible to implement in the current version of Swift, especially for value types. Property observing in Swift is a purely compile-time feature.

Lazy Stored Properties

Initializing a value lazily is such a common pattern that Swift has a special keyword, `lazy`, for defining lazy properties. Note that a lazy property must always be declared as `var` because its initial value might not be set until after initialization completes. Swift has a strict rule that `let` constants must have a value *before* an instance's initialization completes. The `lazy` modifier is a very specific form of memoization.

For example, if we have a view controller that displays a GPSTrack, we might want to have a preview image of the track. By making the property for that lazy, we can defer the expensive generation of the image until the property is accessed for the first time:

```

class GPSTrackViewController: UIViewController {
    var track: GPSTrack = GPSTrack()

```

```
lazy var preview: UIImage = {
    for point in track.record {
        // Do some expensive computation.
    }
    return UIImage(/* ... */)
}()
```

Notice how we defined the lazy property: it's a closure expression that returns the value we want to store — in our case, an image. When the property is first accessed, the closure is executed (note the parentheses at the end), and its return value is stored in the property. This is a common pattern for lazy properties that require more than a one-liner to be initialized.

Because a lazy variable needs storage, we're required to define the lazy property in the definition of `GPSTrackViewController`. Unlike computed properties, stored properties and stored lazy properties can't be defined in an extension. Likewise, unlike computed properties, stored properties and stored lazy properties aren't recomputed every time one of these properties is accessed. For example, when the `track` property changes, the `preview` won't automatically be recomputed.

Let's look at an even simpler example to see what's going on. We have a `Point` struct, and we store `distanceFromOrigin` as a lazy computed property:

```
struct Point {
    var x: Double
    var y: Double
    private(set) lazy var distanceFromOrigin: Double
        = (x*x + y*y).squareRoot()

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}
```

When we create a point, we can access the `distanceFromOrigin` property, and it'll compute the value and store it for reuse. However, if we then change the `x` value, this won't be reflected in `distanceFromOrigin`:

```
var point = Point(x: 3, y: 4)
point.distanceFromOrigin // 5.0
point.x += 10
point.distanceFromOrigin // 5.0
```

It's important to be aware of this. One way around it would be to recompute `distanceFromOrigin` in the `didSet` property observers of `x` and `y`, but then `distanceFromOrigin` wouldn't really be lazy anymore: it'd get computed each time `x` or `y` changes. Of course, in this example, the solution is easy: we should have made `distanceFromOrigin` a regular (non-lazy) computed property from the beginning.

Accessing a lazy property is a mutating operation because the property's initial value is set on the first access. When a struct contains a lazy property, any owner of the struct that accesses the lazy property must therefore declare the variable containing the struct as `var`, because accessing the property means potentially mutating its container. So this isn't allowed:

```
let immutablePoint = Point(x: 3, y: 4)
immutablePoint.distanceFromOrigin
// Error: Cannot use mutating getter on immutable value.
```

Forcing all users of the `Point` type who want to access the lazy property to use `var` is a huge inconvenience, which often makes lazy properties a bad fit for structs.

Additionally, be aware that the `lazy` keyword doesn't perform any thread synchronization. If multiple threads access a lazy property at the same time before the value has been computed, it's possible the computation could be performed more than once, along with any side effects the computation may have.

Property Wrappers

One of the driving motivations to add property wrappers to Swift was almost certainly the release of SwiftUI. However, they were discussed before that, and they're useful beyond just SwiftUI — both Apple and the community have written property wrappers that work in many different situations. In fact, one of the motivations for property wrappers was being able to write lazy properties as a library rather than using the built-in functionality of the compiler.

In essence, property wrappers allow you to modify the behavior of property declarations. For example, consider the following SwiftUI view:

```
struct Toggle: View {  
    @Binding var isOn: Bool  
    // ...  
    var body: some View {  
        if isOn {  
            // ...  
        }  
    }  
}
```

In the code above, `@Binding` is a property wrapper. You can use the `isOn` property just like you'd use a regular `Bool` property: you can both get and set the value. Yet the behavior is different: the memory is stored outside the `Toggle` value, opaque to the view. You can also modify the value without being in a mutating method. Property wrappers are heavily used throughout SwiftUI, mostly to manage state.

You can use property wrappers for properties of classes and structs, for local variables (but not global variables), and in function arguments. At the end of this section, we'll discuss some of the limitations of property wrappers in more detail.

Another use case for property wrappers is when using UIKit and AppKit, where it's common for a property to invalidate part of a view. Recall the code from earlier in this chapter, where we use a change observer to invalidate a view's layout when a certain property changes:

```
class MyView: UIView {  
    var pageSize: CGSize = CGSize(width: 800, height: 600) {  
        didSet {  
            self.setNeedsLayout()  
        }  
    }  
}
```

Using the `@Invalidate` property wrapper added in iOS 15 (and macOS 12), you can now write this code as follows:

```
class MyView: UIView {
    @Invalidate(.layout) var pageSize: CGSize =
        CGSize(width: 800, height: 600)
}
```

The `@Invalidate` property wrapper stores the size internally, and whenever it changes, it calls `setNeedsLayout` on your behalf. Once you have a lot of properties that all invalidate different parts of your view (layout, constraints, display, etc.), using this property wrapper can help significantly clean up your code. Note that you can still use `willSet` and `didSet` with property wrappers; they work just like change observers on regular properties.

Outside of views, property wrappers are useful as well. There are property wrappers written by the community to customize the way a type is encoded or decoded via [Codable](#), property wrappers around [UserDefaults](#), and property wrappers to make reactive programming simpler (for example, `@Published` from [Combine](#)).

Usage

Property wrappers are a syntactic feature: you could use `Binding` and `Invalidate` without property wrappers. Here's the `Binding` example from before, but rewritten without property wrappers:

```
struct Toggle: View {
    var isOn: Binding<Bool>
    // ...
    var body: some View {
        if isOn.wrappedValue {
            // ...
        }
    }
}
```

In fact, when you use property wrappers, the compiler transforms your code in a similar way. To illustrate this, let's build a simple property wrapper for storing values in a box:

```
@propertyWrapper
class Box<A> {
    var wrappedValue: A
```

```
init(wrappedValue: A) {
    self.wrappedValue = wrappedValue
}
}
```

The Box type is useful when you need a mutable variable that's shared (you can pass the same instance of a Box to multiple places) or when you need a mutable variable in a scope that doesn't allow mutation (for example, you can modify the value inside a Box even when you're inside a non-mutating method). The above definition allows us to use a Box like this:

```
struct Checkbox {
    @Box var isOn: Bool = false

    func didTap() {
        isOn.toggle()
    }
}
```

To understand what's happening, let's look at the same code *after* the transformation by the compiler:

```
struct Checkbox {
    private var _isOn: Box<Bool> = Box(wrappedValue: false)
    var isOn: Bool {
        get { _isOn.wrappedValue }
        nonmutating set { _isOn.wrappedValue = newValue }
    }

    func didTap() {
        isOn.toggle()
    }
}
```

For each property marked with a property wrapper, the compiler generates an actual stored property prefixed with an underscore. In addition, a computed property that accesses the `wrappedValue` of the underlying property wrapper is generated. If a property is initialized with a value (in the above example, `isOn` is initialized with the value `false`) the property wrapper is initialized with `.init(wrappedValue:)`.

When defining a property wrapper, you have to provide at least a getter for `wrappedValue`. The setter is optional, and depending on whether or not it's present, a setter is generated for the computed property. In the case above, there's a setter, so a setter is generated for the computed property as well. Because `Box` is a class, the generated setter is nonmutating. Like the setter, the `init(wrappedValue:)` is also optional, but because we provide it in `Box`, we can initialize our `Box<Bool>` with a `Bool`.

Projected Values

In SwiftUI, property wrappers such as `@State` and `@ObservedObject` are used to define the ownership and storage of a value. For example, the definition `@State var x: Int` gives you storage for a mutable `Int` variable. The storage itself is managed by SwiftUI. However, many components don't care about where something is stored; they just need a value they can manipulate. For example, a `Toggle` needs a `Bool` it can read to and write from, and a `TextField` needs a mutable `String`. These values are provided to SwiftUI using the `Binding` property wrapper, which is essentially a getter and a setter to a value stored outside the binding.

SwiftUI lets you create a binding from `@State` (or other property wrappers, such as `@ObservedObject`) using a special property wrappers feature called *projected value*. It works like this: when you implement the `projectedValue` property, an extra computed property — which has the same name as the definition but with a `$` prefix — gets synthesized. In other words, when you have a `foo` property defined using a property wrapper, writing `$foo` is the same as writing `foo.projectedValue`.

To see this in action, we can extend our `Box` so that we can also have references to part of the boxed value. We'll create a type that works almost the same as `Binding` in SwiftUI. For example, if we have a `Box` with a `Person` struct, we could have a reference to the `name` property. The `Box` is still the storage of the `Person` value, but the reference to the person's name allows us to read and write that part of the value. As a first step, we can define a `Reference` property wrapper that stores a way to get and set a value:

```
@propertyWrapper
class Reference<A> {
    private var _get: () -> A
    private var _set: (A) -> ()

    var wrappedValue: A {
        get { _get() }
```

```

    set { _set(newValue) }
}

init(get: @escaping () -> A, set: @escaping (A) -> ()) {
    _get = get
    _set = set
}
}

```

Now, as a second step, we can extend Box to have a `projectedValue`, in turn creating a `Reference<A>` from a `Box<A>`:

```

extension Box {
    var projectedValue: Reference<A> {
        Reference<A>(get: { self.wrappedValue }, set: { self.wrappedValue = $0 })
    }
}

```

Because we implemented `projectedValue`, we can now create a Box holding a Person value, create a reference out of it using the `$` prefix, and pass that reference to a separate function or initializer. Inside `PersonEditor`, any modifications to `person` will change the underlying value inside the Box:

```

struct Person {
    var name: String
}

struct PersonEditor {
    @Reference var person: Person
}

func makeEditor() -> PersonEditor {
    @Box var person = Person(name: "Chris")
    PersonEditor(person:$person)
}

```

Projected values are useful in combination with [key-path-based dynamic member lookup](#). For example, instead of passing a Person to a `PersonEditor`, we might want to pass just the name of a person to a `TextEditor`. In the example above, we can't just write `$person.name`, because `$person` has the type `Reference<Person>`, and not `Person`. We can fix this by adding dynamic member lookup to the `Reference` type:

```

@propertyWrapper
@dynamicMemberLookup
class Reference<A> {
    // ...

    subscript<B>(dynamicMember keyPath: WritableKeyPath<A, B>
        -> Reference<B> {
        Reference<B>(get: {
            self.wrappedValue[keyPath: keyPath]
        }) {
            self.wrappedValue[keyPath: keyPath] = $0
        }
    }
}

```

This allows us to create a `Reference<String>` by writing `$person.name`. The `$foo.prop1.prop2` syntax gets transformed into `foo.projectedValue[dynamicMember: \.prop1.prop2]`.

Enclosing Self

Some property wrappers only work when they have access to the enclosing object. For example, the `@Published` property wrapper from Apple's Combine framework accesses the `objectWillChange` property on the object that contains the `@Published` property. In the example below, the change to `a.city` emits an event through the `a.objectWillChange` publisher:

```

class Address: ObservableObject {
    // ...
    @Published var city: String = "Berlin"
}

let a = Address()
a.city = "New York"

```

Likewise, in the `@Invalidating` example from the introduction of this section, any change to a property that's marked as `@Invalidating` will invalidate the enclosing view. As an example of how this works, we'll reimplement a small part of `@Invalidating`. To get access to the enclosing instance, we need to use a non-official API. Once released, the

official API might look different from the description in this section. Instead of using wrappedValue, we need to implement a static subscript. This subscript receives the enclosing object as a parameter, as well as key paths to the two properties that make up the property wrapper. Here's an example:

```
@propertyWrapper
struct InvalidatingLayout<A> {
    private var _value: A
    //...
    static subscript<T: UIView>(
        _enclosingInstance object:T,
        wrapped _: ReferenceWritableKeyPath<T, A>,
        storage storage: ReferenceWritableKeyPath<T, Self>) -> A {
        get {
            object[keyPath: storage]._value
        }
        set {
            object[keyPath: storage]._value = newValue
            object.setNeedsLayout()
        }
    }
}
```

In the code above, the subscript is now used instead of the wrappedValue. Inside the subscript, we have access to the view (through object) and the property wrapper itself (by using the storage key path together with the enclosing object). We can use the property wrapper just like a regular property wrapper:

```
class AView: UIView {
    @InvalidatingLayout var x = 100
}
```

Given the above definition, this is what the compiler synthesizes for us:

```
class AView: UIView {
    var _x = InvalidatingLayout(wrappedValue: 100)
    var x: Int {
        get {
            InvalidatingLayout[_enclosingInstance: self, wrapped: \.x, storage: \._x]
        }
    }
}
```

```
    set {
        InvalidatingLayout[_enclosingInstance: self, wrapped: \.x, storage: \._x]
            = newValue
    }
}
}
```

Our definition of the `@InvalidateLayout` property isn't quite complete yet. In addition to the subscript, we provide an `init(wrappedValue:)` to allow the initialization with an initial value. We're also required to implement `wrappedValue`, even though we really can't come up with a sensible implementation for the setter (because we need the object). However, we can annotate the property as deprecated:

```
struct InvalidatingLayout<A> {
    // ...

    @available(*, unavailable,
        message: "@InvalidateLayout is only available on UIView subclasses")
    var wrappedValue: A {
        get { fatalError() }
        set { fatalError() }
    }

    init(wrappedValue: A) {
        _value = wrappedValue
    }

    // ...
}
```

The annotation to `wrappedValue` will ensure that our property wrapper can only be used with `UIView` subclasses; any other use will cause a compiler error.

In SwiftUI, you might assume that `@State` works by looking at the enclosing `self` as well. It doesn't, as the enclosing `self` isn't an object, but a value type. Instead, the memory for the state value is managed by SwiftUI internally depending on the position of the view in the view hierarchy. This behavior is implemented using introspection.

Property Wrapper Ins and Outs

Property wrappers work for properties of structs and classes, but they don't work inside enums. This makes sense because enums can't have storage outside a case, and a property wrapper always includes a synthesized stored property. You can also use property wrappers on local variables (for example, inside the body of a function), but not on global variables. Furthermore, properties or variables defined with a property wrapper cannot be marked as unowned, weak, lazy, or `@NSCopying`.

Starting with Swift 5.5, functions can take property wrapper arguments as well. For example, consider the following function:

```
func takesBox(@Box foo: String) {  
    // ...  
}
```

To call this function, you provide a `String` as the parameter — not a `Box<String>`. The compiler will turn the above function into something like this:

```
func takesBox(foo initialValue: String) {  
    var _foo: Box<String> = Box(wrappedValue: initialValue)  
    var foo: String {  
        get { _foo.wrappedValue }  
        nonmutating set { _foo.wrappedValue = newValue }  
    }  
}
```

Note that in the above example, the property wrapper isn't part of the API of `takesBox`. However, if the property wrapper has an initializer named `init(projectedValue:)`, the compiler will also generate a different version of the function that takes a `projectedValue`. For example, if we implemented `init(projectedValue:)` on `Box`, the second version of the `takesBox` function would look like this:

```
func takesBox($foo initialValue: Reference<String>) {  
    var _foo: Box<String> = Box(projectedValue: initialValue)  
    var foo: String {  
        get { _foo.wrappedValue }  
        nonmutating set { foo.wrappedValue = newValue }  
    }  
}
```

In other words, when a property wrapper with an `init(projectedValue:)` is used in a function, it becomes part of the function's API. Properties defined using a property wrapper in a struct are also part of the memberwise initializer. For example, consider the following struct definition:

```
struct Test {  
    @Box var name: String  
    @Reference var street: String  
}
```

The memberwise initializer generated by the compiler will be `init(name: String, street: Reference<String>)`. If the property wrapper has an `init(wrappedValue:)` initializer (like `@Box`), the memberwise initializer will accept a wrapped value for this property. If the property wrapper doesn't have this initializer (like `@Reference`), the corresponding argument in the memberwise initializer includes the wrapper. You can, of course, still write your own initializer to accept wrapped or non-wrapped values as you see fit.

Property wrappers can be nested as well. For example, to add a double box around a property, you can write: `@Box @Box var name: String`. However, this doesn't seem to work (yet) with property wrappers that use the enclosing self instance. Furthermore, property wrappers that rely on introspection (e.g. `@State` in SwiftUI) will typically only work if they're the outermost wrapper.

Key Paths

A key path is an uninvoked reference to a property, analogous to an unapplied method reference. *Key-path expressions* start with a backslash, e.g. `\String.count`. The backslash is necessary to distinguish a key path from a type property of the same name that may exist (suppose that `String` also had a static `count` property — then `String.count` would return the value of that property). Type inference works in key-path expressions too: you can omit the type name if the compiler can infer it from the context, which leaves `\.count`.

As the name suggests, a key path describes a *path* through a type hierarchy, starting at the root value. For example, given the following `Person` and `Address` types, `\Person.address.street` is a key path that resolves a person's street address:

```
struct Address {  
    var street: String
```

```
var city: String
var zipCode: Int
}

struct Person {
    let name: String
    var address: Address
}

let streetKeyPath = \Person.address.street
// Swift.WritableKeyPath<Person, Swift.String>
let nameKeyPath = \Person.name // Swift.KeyPath<Person, Swift.String>
```

Key paths can be composed of any combination of stored and computed properties, along with optional chaining operators. The compiler automatically generates a new [keyPath:] subscript for all types. You use this subscript to “invoke” the key path, i.e. to access the property described by it on a given instance. So "Hello"[keyPath: \.count] is equivalent to "Hello".count. Or, for our current example:

```
let simpsonResidence = Address(street: "1094 Evergreen Terrace",
    city: "Springfield", zipCode: 97475)
var lisa = Person(name: "Lisa Simpson", address: simpsonResidence)
lisa[keyPath: nameKeyPath] // Lisa Simpson
```

If you look at the types of our two key-path variables above, you'll notice that the type of nameKeyPath is KeyPath<Person, String> (i.e. a strongly typed key path that can be applied to a Person and yields a String), whereas streetKeyPath's type is WritableKeyPath. Because all properties that form this latter key path are mutable, the key path itself allows mutation of the underlying value:

```
lisa[keyPath: streetKeyPath] = "742 Evergreen Terrace"
```

Doing the same with nameKeyPath would produce an error because the underlying property isn't mutable.

Key paths can not only refer to properties though; we can also use them to traverse subscripts. For example, the following syntax can be used to extract the name of the second person value in an array:

```
var bart = Person(name: "Bart Simpson", address: simpsonResidence)
let people = [lisa, bart]
```

```
people[keyPath:\.[1].name] // Bart Simpson
```

The same syntax can also be used to include dictionary subscripts in key paths.

Key Paths Can Be Modeled with Functions

A key path that maps from a base type `Root` to a property of type `Value` is very similar to a function of type `(Root) -> Value` — or, for writable key paths, a *pair* of functions for both getting and setting a value. The major benefit key paths have over such functions (aside from the syntax) is that they’re *values*. You can test key paths for equality and use them as dictionary keys (they conform to `Hashable`), and you can be sure that a key path is stateless — unlike functions, which might capture mutable state. None of these things are possible with normal functions.

The compiler can automatically convert a key-path expression to a function. For example, the following code is shorthand for `people.map { $0.name }`:

```
people.map(\.name) // ["Lisa Simpson", "Bart Simpson"]
```

Note that this only works for key-path expressions. For example, the following code uses a key-path expression to define a key path, and it doesn’t compile:

```
/*show*/  
let keyPath = \Person.name  
people.map(keyPath)
```

With a key-path expression, the compiler has two options for the inferred type: `\Person.name` could either be `KeyPath<Person, String>` or `(Person) -> String`. The compiler will prefer the key path type, but if that doesn’t work, it’ll try the function type.

Key paths are also composable by appending one key path to another. Notice that the types must match: if you start with a key path from A to B, the key path you append must have a root type of B, and the resulting key path will then map from A to the appended key path’s value type, say C:

```
// KeyPath<Person, String> + KeyPath<String, Int> = KeyPath<Person, Int>  
let nameCountKeyPath = nameKeyPath.appending(path: \.count)  
// Swift.KeyPath<Person, Swift.Int>
```

Writable Key Paths

A writable key path is special; you can use it to read *or* write a value. Hence, it's equivalent to a pair of functions: one for getting the property ((Root) → Value), and another for setting the property ((inout Root, Value) → Void). Writable key paths capture a lot of code in a neat syntax. Compare `streetKeyPath` with the equivalent getter and setter pair:

```
let streetKeyPath = \Person.address.street
let getStreet: (Person) -> String = { person in
    return person.address.street
}
let setStreet: (inout Person, String) -> () = { person, newValue in
    person.address.street = newValue
}

// Setter usage
lisa[keyPath: streetKeyPath] = "1234 Evergreen Terrace"
setStreet(&lisa, "1234 Evergreen Terrace")
```

Writable key paths come in two forms: `WritableKeyPath` and `ReferenceWritableKeyPath`. The second type is used with values that have reference semantics (classes), and the first type is used with all other types. The difference in use is that a `WritableKeyPath` requires the root value to be mutable, whereas a `ReferenceWritableKeyPath` doesn't require this.

Writable key paths are used throughout frameworks like SwiftUI. For example, in SwiftUI, there's an “environment” value that's passed through the view hierarchy. This value is managed and propagated by the framework, but you can modify it using special functions that require a `WritableKeyPath` to mutate part of that value. As we saw in the section on [projected values](#), writable key paths are also very useful in combination with dynamic member lookup.

The Key Path Hierarchy

There are five different types for key paths, each adding more precision and functionality to the previous one:

- An AnyKeyPath is similar to a function of type (Any) -> Any?.
- A PartialKeyPath<Source> is similar to a function of type (Source) -> Any?.
- A KeyPath<Source, Target> is similar to a function of type (Source) -> Target.
- A WritableKeyPath<Source, Target> is similar to a *pair* of functions of type (Source) -> Target and (inout Source, Target) -> () .
- A ReferenceWritableKeyPath<Source, Target> is similar to a *pair* of functions of type (Source) -> Target and (Source, Target) -> () . The second function can update a Source value with Target, and it works only when Source is a reference type. The distinction between WritableKeyPath and ReferenceWritableKeyPath is necessary because the setter for the former has to take its argument as an inout parameter.

This hierarchy of key paths is currently implemented as a class hierarchy. Ideally, these would be protocols, but Swift's generics system lacks some features to make this feasible. The class hierarchy is intentionally kept closed to facilitate changing this in a future release without breaking existing code.

As we've seen before, key paths are different from functions: they conform to Hashable, and in the future, they'll probably conform to Codable as well. This is why we say AnyKeyPath is *similar* to a function of type (Any) -> Any. While we can convert a key path into its corresponding function(s), we can't always go in the other direction.

Key Paths Compared to Objective-C

In Foundation and Objective-C, key paths are modeled as strings (we'll call these Foundation key paths to distinguish them from Swift's key paths). Because Foundation key paths are strings, they don't have any type information attached to them. In that sense, they're similar to AnyKeyPath. If a Foundation key path is misspelled or not well formed, or if the types don't match, the program will probably crash. (The #keyPath directive in Swift helps a little with the misspelling; the compiler can check if a property with the specified name exists.) Swift's KeyPath, WritableKeypath, and ReferenceWritableKeyPath are correct by construction: they can't be misspelled, and they don't allow for type errors.

Many Cocoa APIs use (Foundation) key paths when a function might have been better. This is partially a historical artifact: anonymous functions (or blocks, as Objective-C calls them) are a relatively recent addition, and key paths have been around much

longer. Before blocks were added to Objective-C, it wasn't easy to represent something similar to the function `{ $0.address.street }` except by using a key path: "address.street".

Future Directions

Key paths are still under active discussion, and it's likely they'll become more capable in the future. One possible feature is serialization through the `Codable` protocol. This would allow us to persist key paths on disk, send them over the network, and so on. Once we have access to the structure of key paths, this enables introspection. For example, we could use the structure of a key path to construct well-typed database queries (there are open source projects that do this already, but they rely on internals that might change in the future). If types could automatically provide an array of key paths to their properties, this could serve as the basis for runtime reflection APIs.

At the moment, key paths are very slow compared to direct access of properties. This is a [known bug](#). In most cases, the clarity of key paths is more important than their speed, but when you're writing performance-sensitive code, you should be aware of this (for example, don't instantiate key paths in a tight loop).

Recap

Properties and variables are an essential part of Swift. While stored properties and computed properties share the same syntax when used, they do two very different things: stored properties are used to store data, whereas computed properties are more similar to functions.

To a large extent, property wrappers and key paths are syntactic sugar. In other words, they allow you to express things that you could already write before, just with a much shorter syntax. However, don't dismiss them for that reason. Clean syntax is important for writing clear code.

A property wrapper can make it harder to understand what's going on, just like any other abstraction. Yet a carefully designed property wrapper can clean up common patterns and is worth the extra abstraction. SwiftUI's use of property wrappers is a good example of this.

Structs and Classes

6

When we're designing our data types, Swift lets us choose between two alternatives that seem to be similar on the surface: structs and classes. Both can have stored and computed properties, and both can have methods defined on them. Furthermore, not only do both have initializers, but we can define extensions on them, and we can conform them to protocols. Sometimes our code even keeps compiling when we change the class keyword into struct or vice versa. However, the similarities on the surface are deceptive, as structs and classes have fundamentally different behaviors.

Structs are *value types*, whereas classes are *reference types*. Even if we don't think in these terms, we're all familiar with the behavior of values and references in our daily work. We'll try to leverage this implicit understanding in the next section to shine some light on the formal distinction between value types and reference types in general, and structs and classes specifically.

Value Types and Reference Types

Let's start by looking at one of the simplest types possible: integers. Consider the following code:

```
var a: Int = 3
var b = a
b += 1
```

What's the value of a now? It's probably safe to say that we all expect a to still hold the value 3, even though we've incremented b to 4. Anything else would be a big surprise. And this is indeed correct:

```
a // 3
b // 4
```

This behavior is the essence of value types: assignment copies the value. In other words, each value type variable holds its own independent value. If a type behaves that way, it's also said to have *value semantics*.

Looking at the definition of Int in the standard library, we can indeed see that it's a struct (and therefore has value semantics):

```
public struct Int: FixedWidthInteger, SignedInteger {  
    ...  
}
```

Before we proceed, let's take a step back and look at this behavior from a more low-level perspective.

What do we mean by the term “variable?” We can say that a variable is a name for a location in memory that contains a value of a certain type. In the example above, we use the name `a` to refer to a location in memory of type `Int` currently holding the value 3. The second variable, `b`, is a name for a different location in memory, equally of type `Int` and containing the value 3 after the initial assignment. The statement `b += 1` then takes the value stored in the memory location referred to as `b`, increments it by one, and writes it back to the same location in memory. Thus, `b` now contains the value 4. Since the increment statement only modifies the value of the `b` variable, `a` is unaffected by this statement.

Memory	
...	
Variable <code>a</code> : Int	3
Variable <code>b</code> : Int	4
...	

Figure 6.1: A value type variable is a name for a location in memory that directly contains the value.

Value types are characterized by this direct relationship between variable and value: the value (also referred to as the instance of the value type) resides directly at the location in memory behind the variable. This applies to simple value types like integers, but also to more complex types like custom structs with multiple properties (on a machine code level, this might not hold true due to compiler optimizations, but they're invisible to the developer so that our description is at least semantically accurate).

Next, let's look at a view class as an example of a typical reference type:

```
var view1 = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
var view2 = view1
view2.frame.origin = CGPoint(x: 50, y: 50)
view1.frame.origin // (50, 50)
view2.frame.origin // (50, 50)
```

Although we assigned a new origin to `view2.frame.origin`, we naturally expect the frame of `view1` to have changed as well. In fact, we expect `view1` and `view2` to be the same thing in a certain sense — they both represent the same view we see onscreen. That's the casual way of saying `UIView` is a reference type, and that the `view1` and `view2` variables contain references pointing to the same underlying `UIView` instance in memory.

We reassign the `view2` variable, like so:

```
view2 = UILabel()
```

When we do this, `view1` still references the previously created view, while `view2` now references the newly created label instance. In other words, the reassignment has changed the instance (or object) `view2` is pointing to.

This is the essence of reference types: variables don't contain the "thing" itself (e.g. the instance of `UIView` or `URLSession`), but a reference to it. Other variables can also contain a reference to the same underlying instance, and the instance can be mutated through any of its referencing variables. A type with these properties is also said to have *reference semantics*.

Compared to value types, there's one more level of indirection at play. While a value type variable contains the value itself, a reference type variable contains a reference pointing to the value somewhere else. This indirection allows us to share access to an object across different parts of our program.

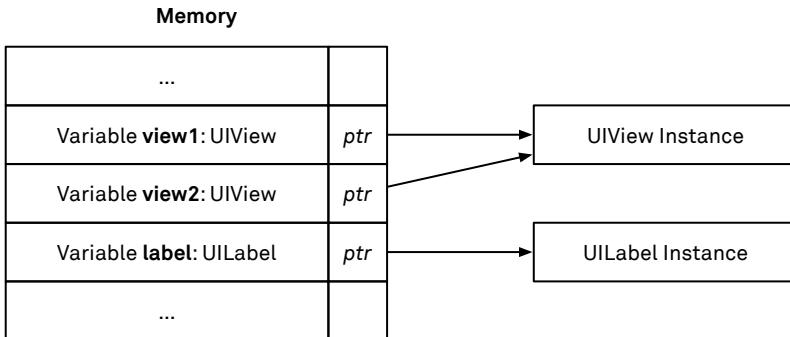


Figure 6.2: A reference type variable contains a pointer to the actual instance somewhere else in memory.

Let's look at the different behaviors of value types and reference types with an example of a custom type we define, starting with a class:

```

class ScoreClass {
    var home: Int
    var guest: Int
    init(home: Int, guest: Int) {
        self.home = home
        self.guest = guest
    }
}

var score1 = ScoreClass(home: 0, guest: 0)
var score2 = score1
score2.guest += 1
score1.guest // 1

```

Both variables, `score1` and `score2`, reference the same underlying instance of `Score`. Therefore, mutating the guest's score via `score2` changes the value we see when accessing the guest's score via `score1` as well. We can also pass `score2` to a function that performs the mutation:

```

func scoreGuest(_ score: ScoreClass) {
    score.guest += 1
}

```

```
scoreGuest(score1)
score1.guest//2
score2.guest//2
```

If we instead define the score type as a struct, the behavior changes:

```
struct ScoreStruct {
    var home: Int
    var guest: Int
    // Memberwise initializer synthesized by the compiler.
}
var score3 = ScoreStruct(home:0, guest: 0)
var score4 = score3
score4.guest += 1
score3.guest//0
```

As we saw with integers above, assigning a struct to another variable creates an independent copy of the value. Therefore, changing the guest's score via the `score4` variable has no effect on the guest's score in `score3`.

Using the struct version of `Score`, we can't define the same `scoreGuest` function as we did above for the class analogue. First, passing a value type as a function's parameter creates an independent copy of the value, just as assignment to a variable does. Second, the function parameter is immutable within the function (like a variable declared using `let`), so we cannot change its properties. To create a similar function, we'd have to use an `inout` parameter, which we'll cover in the next section.

We hope this initial overview of the behavior of structs and classes highlights their different natures despite their similarities in syntax and shared features. Through the rest of this chapter, we'll explore the tradeoffs between structs and classes. While classes are the more powerful tool, their capabilities come at a cost. On the other hand, structs are much more limiting, but these limitations can also be beneficial.

Mutation

Structs and classes are significantly different when it comes to controlling mutability. This might be unintuitive at first, but it'll make sense in light of the distinction between structs being value types and classes being reference types. As an example, we'll use the `ScoreClass` and `ScoreStruct` types from above again:

```
class ScoreClass {  
    var home: Int  
    var guest: Int  
    init(home: Int, guest: Int) {  
        self.home = home  
        self.guest = guest  
    }  
}  
  
struct ScoreStruct {  
    var home: Int  
    var guest: Int  
    // Memberwise initializer synthesized by the compiler.  
}
```

Both versions have their `home` and `guest` properties declared using the `var` keyword. If we create instances of both and store them in `var` variables, we can freely mutate the properties:

```
var scoreClass = ScoreClass(home: 0, guest: 0)  
var scoreStruct = ScoreStruct(home: 0, guest: 0)  
scoreClass.home += 1  
scoreStruct.guest += 1
```

There's an important difference between mutating the class and the struct versions though: struct mutation is always local to the variable we're mutating, i.e. only the value of the local variable, `scoreStruct`, is being changed. Mutating the class instance has potentially global effects: anyone who also holds a reference to the same instance will be affected by the change.

If we store the instances in `let` variables, we can still mutate the class instance, but not the struct instance:

```
let scoreClass = ScoreClass(home: 0, guest: 0)  
let scoreStruct = ScoreStruct(home: 0, guest: 0)  
scoreClass.home += 1 // works  
scoreStruct.guest += 1  
// Error: Left side of mutating operator isn't mutable:  
// 'scoreStruct' is a 'let' constant.
```

Declaring a variable with `let` means its value cannot be changed after initialization. Since the value of the `scoreClass` variable is the reference to the instance of `ScoreClass`, this only means we can't assign another reference to the `scoreClass` variable. However, to mutate a property of the `ScoreClass` instance we've created, we don't need to mutate the value of `scoreClass`. We just use the reference in `scoreClass` to get to the instance, which is where we can change the properties since they were declared as `var` in the class.

In the case of structs, this works very differently. Since structs are value types, the `scoreStruct` variable doesn't just contain a reference to an instance somewhere else; rather it actually contains the `ScoreStruct` instance itself. Since the value of `let` variables cannot be changed after initial assignment, we can't change a property anymore, even if it's declared with `var` in the struct. The reason is that changing a property within a struct is semantically equivalent to assigning a whole new struct instance to the variable. So the example from above:

```
scoreStruct.guest += 1
```

is equivalent to:

```
scoreStruct = ScoreStruct(home: scoreStruct.home,  
                         guest: scoreStruct.guest + 1)
```

This applies not only to mutating a direct property of the struct instance, but also to mutating any nested property. For example, assigning a new value to the `x`-coordinate of a rectangle's origin is semantically equivalent to assigning a whole new rectangle value to the variable:

```
var rect = CGRect(origin: .zero, size: CGSize(width: 100, height: 100))  
rect.origin.x = 10 // this is the same as...  
rect = CGRect(origin: CGPoint(x: 10, y: 0), size: rect.size)
```

What happens if we declare the properties with `let`, but we declare the `scoreClass` and `scoreStruct` variables with `var`?

```
class ScoreClass {  
    let home: Int  
    let guest: Int  
    init(home: Int, guest: Int) {  
        self.home = home
```

```
    self.guest = guest
}
}

struct ScoreStruct {
  let home: Int
  let guest: Int
}

var scoreClass = ScoreClass(home: 0, guest: 0)
var scoreStruct = ScoreStruct(home: 0, guest: 0)
scoreClass.home += 1
// Error: Left side of mutating operator isn't mutable:
// 'home' is a 'let' constant.
scoreStruct.guest += 1
// Error: Left side of mutating operator isn't mutable:
// 'guest' is a 'let' constant.
```

The mutation fails in the class case, even though `scoreClass` is declared with `var`. The reason is that `var` on the variable declaration only means that we can change the value of the variable. In the class case, the value of the variable is a reference to an instance, so we can change the reference:

```
scoreClass = ScoreClass(home: 2, guest: 1) // works
```

However, we can't change the `home` property on the instance `scoreClass` is referencing, because the property has been defined with `let`.

The mutation also fails in the struct case: since the properties are defined with `let`, we can no longer use them to change the value in `scoreStruct`, even though `scoreStruct` is a `var`. However, we can still assign a new struct to the `someStruct` variable:

```
scoreStruct = ScoreStruct(home: 2, guest: 1) // works
```

Lastly, if we define the properties and the variables with `let`, the compiler no longer allows any kind of mutation: we can't assign new instances to the `someClass` or `someStruct` variables, and we can't mutate the instance properties either.

We recommend using `var` properties in structs as a default. This allows the mutability of struct instances to be controlled by using `var` or `let` on the variable level, which gives you more flexibility. In contrast to classes, using `var` properties in structs doesn't introduce

potentially global mutable state, because mutating a struct property really just creates a copy of the struct with a changed field. `let` should be used sparingly and intentionally for properties that really shouldn't be mutated after initialization (e.g. because mutating a single property would bring the struct into an invalid state), even if the instance is stored in a `var` variable.

The key to understanding all the different combinations of `let` and `var` properties and variables is to remember two points:

- The value of a class variable is the reference to the instance, while the value of a struct variable is the struct instance itself.
- Changing a property of a struct, even through multiple levels of nesting, is like assigning a whole new struct instance to the variable.

Mutating Methods

Normal methods on structs defined with the `func` keyword can't mutate any of the struct's properties. This is because the `self` parameter that's implicitly passed into every method is immutable by default. We have to explicitly say `mutating func` to create a method that allows mutation:

```
extension ScoreStruct {  
    mutating func scoreGuest() {  
        self.guest += 1  
    }  
}  
  
var scoreStruct2 = ScoreStruct(home: 0, guest: 0)  
scoreStruct2.scoreGuest()  
scoreStruct2.guest // 1
```

Within a mutating method, we can consider `self` to be a `var`, so we can change properties on `self` as long as they're declared using `var` as well.

The compiler uses the presence of the `mutating` keyword as a marker to decide which methods can't be called on `let` constants. We can only call mutating methods on variables declared with `var`, since calling a mutating method is like assigning a new value to the variable (in fact, assigning a completely new value to `self` is also permitted in a mutating method). If we try to call a mutating method on a `let` variable, the compiler

shows an error, even if that method does not in fact mutate `self` — the mutating annotation is enough to disallow the call.

Property and subscript setters are implicitly mutating. In the rare case where you want to implement a computed property with a non-mutating setter (e.g. because your struct is a wrapper for a global resource and the setter only mutates global state), you can annotate the setter with `nonmutating set`. The compiler allows you to call such a setter on a `let` constant.

Classes don't have and don't need mutating methods: as we've seen above, we can mutate the properties of a class instance even through a variable that's declared with `let`. Similarly, `self` behaves like a `let` variable inside a class's methods. We can't reassign `self`, but we can use it to mutate properties on the instance `self` is referencing, as long as these properties are declared with `var`.

inout Parameters

We mentioned above that a mutating method on a struct has access to a mutable `self`, and therefore can change any `var` property on `self`. `inout` parameters allow us to write functions that can mutate any one of their parameters in place, not just `self`. As an example, let's write the mutating `scoreGuest` method as a free function:

```
func scoreGuest(_ score: ScoreStruct) {  
    score.guest += 1  
    // Error: Left side of mutating operator isn't mutable:  
    // 'score' is a 'let' constant.  
}
```

By default, function parameters, like `let` variables, are immutable. Of course, we can copy a parameter into a local `var`, but mutating this variable will have no effect on the original value that got passed in. To solve this problem, we add the `inout` keyword to the parameter's type:

```
func scoreGuest(_ score: inout ScoreStruct) {  
    score.guest += 1  
}  
  
var scoreStruct3 = ScoreStruct(home: 0, guest: 0)
```

```
scoreGuest(&scoreStruct3)
scoreStruct3.guest // 1
```

To call the `scoreGuest` function with an `inout` parameter, we have to do two things: first, the variable we're passing as an `inout` parameter has to be defined as `var`, and second, we have to prefix the variable name with `&` when passing it to the function. The required ampersand character makes it very clear at the call site that the function can now change the value of this variable.

Although the ampersand may remind you of the *address-of* operator in C or Objective-C, or of passing by reference in C++, that isn't what's happening in this case. `inout` parameters are passed as copies just as a regular parameter would be, but they're copied back when the function returns. In other words, when the function mutates an `inout` parameter multiple times, the caller will only see a single change as the new value is copied back. By the same logic, even if the function doesn't mutate its `inout` parameter at all, the caller will still see a mutation (i.e. any `willSet` and `didSet` observers will fire). As we explained in the [Functions](#) chapter, the compiler might optimize the copy-in-and-out to pass by reference if it can do so safely.

Lifecycle

Structs and classes are very different when it comes to lifecycle management. Structs are much simpler in this regard, because they can't have multiple owners; their lifetime is tied to the lifetime of the variable containing the struct. When the variable goes out of scope, its memory is freed and the struct disappears.

In contrast, an instance of a class can be referenced by multiple owners, which requires a more sophisticated memory management model. Swift uses [Automatic Reference Counting](#) (ARC) to keep track of the number of references to a particular instance. When the reference count goes down to zero (because all variables holding a reference have gone out of scope or have been set to `nil`), the Swift runtime calls the object's `deinit` and frees the memory. Therefore, classes can be used to model shared entities that perform cleanup work when they're eventually freed. Examples of this include file handles (which have to close their underlying file descriptors) and view controllers that might need to unregister a notification observer.

Reference Cycles

A reference cycle is when two or more objects reference each other strongly in a way that prevents them from ever being deallocated (short of the developer explicitly breaking the cycle). This creates memory leaks and prevents the execution of potential cleanup tasks.

Since structs are simple values, it's impossible to create reference cycles between them (as there are no references to structs). That's an advantage on one hand and a limitation on the other: it's one less thing to worry about, but it also means we can't model cyclical data structures using structs. For classes, the reverse applies: since the same instance can have multiple owners, we can use classes to model cyclical data structures, but we have to be careful to not create reference cycles.

Reference cycles can take many forms — from two objects referencing each other strongly, to complex cycles consisting of many objects and closures closing over objects. Let's first look at a simple example involving a window and its root view:

```
// First version
class Window {
    var rootView: View?
}

class View {
    var window: Window
    init(window: Window) {
        self.window = window
    }
}

var window: Window? = Window() // refcount: 1
window = nil // refcount: 0, deallocating
```

After the first line, the reference count is one. The moment we set the variable to `nil`, the reference count of our `Window` instance is zero, and the instance gets deallocated. However, if we also create a view and assign it to the window's `rootView` property, the reference count never comes down to zero again. Let's trace the reference counts line by line.

First, the window is created. The reference count for the window is now one:

```
var window: Window? = Window() // window: 1
```

Next, the view gets created and holds a strong reference to the window, so the window's reference count is now two, and the view's reference count is one:

```
var view: View? = View(window: window!) // window: 2, view: 1
```

Assigning the view as the window's `rootView` increases the view's reference count by one. Now, both the view and the window have a reference count of two:

```
window?.rootView = view // window: 2, view: 2
```

After setting both variables to `nil`, they still have a reference count of one:

```
view = nil // window: 2, view: 1  
window = nil // window: 1, view: 1
```

Even though they're no longer accessible from a variable, they strongly reference each other. This is called a *reference cycle*, and when dealing with graph-like data structures, we need to be very aware of the potential of creating memory leaks through cycles. Because of the reference cycle, these two objects will never be deallocated during the lifetime of the program.

Weak References

To break the reference cycle, we need to make one of the references weak or unowned. Assigning an object to a weak variable doesn't change its reference count. Weak references in Swift are always *zeroing*: the variable will automatically be set to `nil` once the referred object gets deallocated. This is why weak references must always be optionals.

To fix the example above, we'll make the window's `rootView` property weak, which means it won't strongly reference the view and thus will automatically become `nil` once the view is deallocated. To see what's going on, we can add some print statements to the classes' deinitializers. `deinit` gets called just before a class deallocates:

```
// Second version  
class Window {  
    weak var rootView: View?
```

```
deinit {
    print("Deinit Window")
}

class View {
    var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}
```

In the code below, we again create a window and a view. As before, the view strongly references the window; but because the window's rootView is declared as weak, the window doesn't strongly reference the view anymore. This way, we have no reference cycle, and both objects get deallocated when we set the variables to nil:

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
window = nil
view = nil
/*
Deinit View
Deinit Window
*/
```

Weak references are very useful when working with delegates, as is common in Cocoa. The delegating object (e.g. a table view) needs a reference to its delegate, but it shouldn't own the delegate because that would likely create a reference cycle. Therefore, delegate references are usually weak, and another object (e.g. a view controller) is responsible for making sure the delegate stays around for as long as needed.

Unowned References

Sometimes, though, we want a non-strong reference that isn't optional. For example, maybe we know our views will always have a window (so the property shouldn't be

optional), but we don't want a view to strongly reference the window. For these situations, there's the `unowned` keyword:

```
// Third version
class Window {
    var rootView: View?
    deinit {
        print("Deinit Window")
    }
}

class View {
    unowned var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}
```

In the code below, we can see that both objects get deallocated, as in the previous example with a weak reference:

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
view = nil
window = nil
/*
Deinit Window
Deinit View
*/
```

With unowned references, we're responsible for ensuring that the “referencee” outlives the “referencer.” In this example, we have to be sure the window outlives the view. If the window is deallocated before the view is deallocated, and if the unowned variable is accessed, the program will crash.

Note that this isn't the same as undefined behavior. The Swift runtime keeps a second reference count in the object to keep track of unowned references. When all strong

references are gone, the object will release all of its resources (for example, any references to other objects). However, the memory of the object itself will still be there until all unowned references are gone too. The memory is marked as invalid (sometimes also called zombie memory), and any time we try to access an unowned reference, a runtime error will occur.

This safeguard can be circumvented by using `unowned(unsafe)`. If we access an invalid reference that's marked as `unowned(unsafe)`, we get undefined behavior.

Closures and Reference Cycles

Classes aren't the only kind of reference type in Swift. There are also actors, which we'll cover in the [Concurrency](#) chapter, and functions (which also include closure expressions and methods). If a closure captures a variable holding a reference type, the closure will maintain a strong reference to it. Next to the previous example, this is the other primary way of introducing reference cycles into your code.

The usual pattern is like this: object A references object B, but object B stores a closure that references object A (in practice, the reference cycle can involve multiple intermediate objects and closures). As an example, we add an optional `onRotate` callback to the `window` class from above:

```
class Window {  
    weak var rootView: View?  
    var onRotate: (() -> ())? = nil  
}
```

If we configure the `onRotate` callback and use the view in there, we've introduced a reference cycle:

```
var window: Window? = Window()  
var view: View? = View(window: window!)  
window?.onRotate = {  
    print("We now also need to update the view: \(String(describing: view))")  
}
```

The view references the window, the window references the callback, and the callback references the view:

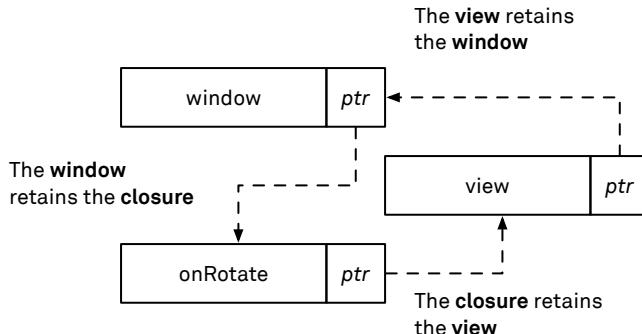


Figure 6.3: A retain cycle between the view, window, and callback

There are three places where we could break this reference cycle (each corresponding to an arrow in the diagram above):

- We could make the view's reference to the window weak. Unfortunately, the window would be deallocated immediately, because there are no other references keeping it alive.
- We might want to mark the `onRotate` property as weak, but Swift doesn't allow marking function properties as weak.
- We could make sure the closure doesn't strongly reference the view by using a capture list that captures the view weakly. This is the only correct option in this example.

```
window?.onRotate = { [weak view] in
    print("We now also need to update the view: \(String(describing: view))")
}
```

Capture lists can do more than just marking variables as weak or unowned. For example, if we wanted to have a weak variable that refers to the window, we could initialize it in the capture list, or we could even define completely unrelated variables, like so:

```
window?.onRotate = { [weak view, weak myWindow=window, x=5*5] in
    print("We now also need to update the view: \(String(describing: view))")
    print("window: \(String(describing: myWindow)), x: \(x)")
}
```

This is almost the same as defining the variable just above the closure, except that with capture lists, the scope of the variable is just the scope of the closure; it's not available outside of the closure.

Choosing between Unowned and Weak References

Should you prefer unowned or weak references in your own APIs? Ultimately, the answer to this question boils down to the lifetimes of the objects involved. If the objects have independent lifetimes — that is, if you can't make any assumptions about which object will outlive the other — a weak reference is the only safe choice.

On the other hand, if you can guarantee that the non-strongly referenced object has the same lifetime as its counterpart or will always outlive it, an unowned reference is often more convenient. This is because it doesn't have to be optional, and the variable can be declared with `let`, whereas weak references must always be optional vars. Same-lifetime situations are very common, especially when the two objects have a parent-child relationship. When the parent controls the child's lifetime with a strong reference and you can guarantee that no other objects know about the child, the child's back reference to its parent can always be unowned.

Unowned references also have less overhead than weak references, so accessing a property or calling a method on an unowned reference will be slightly faster. That said, this should only be a factor in very performance-critical code paths.

The downside of preferring unowned references is, of course, that your program may crash if you make a mistake in your lifetime assumptions. Personally, we often find ourselves preferring weak even when unowned could be used because the former forces us to explicitly check if the reference is still valid at every point of use. Especially when refactoring code, it's easy to break previous lifetime assumptions and introduce crashing bugs.

But there's also an argument to be made for always using the modifier that captures the lifetime characteristics you expect your code to have in order to make them explicit. If you or someone else later changes the code in a way that invalidates those assumptions, a hard crash is arguably the sensible way to alert you to the problem — assuming you find the bug during testing.

Deciding between Structs and Classes

When designing a type, we have to think about whether ownership of a particular instance of this type has to be shared among different parts of our program, or if multiple instances can be used interchangeably as long as they represent the same value. To share ownership of a particular instance, we have to use a class. Otherwise, we can use a struct.

For example, an instance of `URL` cannot be shared, because `URL` is a struct. Every time we assign a `URL` to a variable or pass one to a function, the compiler will make a copy. However, that's not a problem, because we consider two `URL` instances to be interchangeable if they represent the same URL. The same applies to other structs like integers, Booleans, and strings: we don't care whether two integers or two strings are backed by the same piece of memory; we care whether they represent the same value.

In contrast, we don't regard two instances of `UIView` as interchangeable. Even if all their properties are the same, they still represent different "objects" onscreen in different places of the view hierarchy. Therefore, `UIView` is modeled as a class so that we can pass a reference to a particular instance around to multiple parts of our program: a particular view is referenced by its superview, but also by its child views as their superview. In addition, we can store additional references to the view, e.g. in a view controller. The same view instance can be manipulated through all references, and these changes are automatically reflected through all references.

That being said, when we're designing a type that doesn't need shared ownership, we don't have to use a struct. We can also model it as a class, potentially providing an immutable API so that the type essentially has value semantics. In that sense, we can get away with only using classes without having to drastically alter the way we design our program. Of course, we'd lose some compile-time enforcement around mutability, and we might incur the cost of additional reference counting operations, but we could make it work.

On the other hand, if we wouldn't have classes (or references in general) at our disposal, we'd lose the entire concept of shared ownership and we'd have to rearchitect our program from top to bottom (assuming we've relied on classes before). So while we can model a struct as a class with some tradeoffs, the reverse isn't necessarily the case.

Structs are a tool in our toolbox that are purposefully less capable than classes. In return, structs offer simplicity: no references, no lifecycle, no subtypes. That means we don't

have to worry about reference cycles, side effects and race conditions through shared references, and inheritance rules — to name just a few examples.

In addition, structs promise better performance, especially for small values. For example, if `Int` were a class, an array of `Ints` would take up a lot more memory to store the references (pointers) to the actual instances, along with the additional overhead each instance requires (e.g. to store its reference count). Even more importantly, looping over this array would be much slower because the code would have to follow the additional level of indirection for each element and thus potentially be unable to make effective use of CPU caches, especially if the `Int` instances were allocated at wildly different locations in memory.

Classes with Value Semantics

Above, we outlined that structs have value semantics (i.e. each variable contains an independent value) and that classes have reference semantics (i.e. multiple variables can all point to the same underlying class instance). While that's true, we can write immutable classes that behave more like a value type, and we can write structs that don't really behave like a value type — at least on first sight.

When writing a class, we can lock it down to the point where its reference semantics no longer have an impact on its behavior. First, we declare all properties as `let`, making them immutable. Then, we make the class `final` to disallow subclassing, in order to prevent potential subclasses from reintroducing any mutable behavior:

```
final class ScoreClass {  
    let home: Int  
    let guest: Int  
    init(home: Int, guest: Int) {  
        self.home = home  
        self.guest = guest  
    }  
}  
  
let score1 = ScoreClass(home: 0, guest: 0)  
let score2 = score1
```

The `score1` and `score2` variables still contain references to the same underlying `ScoreClass` instance — that's how classes work after all. However, for all practical intents

and purposes, we can use `score1` and `score2` as if they contained independent values, because the underlying instance is completely immutable anyway.

An example of this is the `NSArray` class in Foundation. `NSArray` itself doesn't expose any mutating APIs, so its instances can essentially be used as if they were values. The reality is somewhat more complicated, since `NSArray` has a mutable subclass, `NSMutableArray`, and we can't make assumptions that we're really dealing with an `NSArray` instance if we haven't created it ourselves. That's why we declared our class as `final` above, and that's also why it's recommended to make a copy of an `NSArray` you receive from an API you don't control before you do anything else with it.

Structs with Reference Semantics

The reverse — structs containing reference type properties — also exhibits surprising behavior. Let's extend the `ScoreStruct` type to include a computed property, `pretty`, that provides a nicely formatted string for the current score:

```
struct ScoreStruct {
    var home: Int
    var guest: Int
    let scoreFormatter: NumberFormatter

    init(home: Int, guest: Int) {
        self.home = home
        self.guest = guest
        scoreFormatter = NumberFormatter()
        scoreFormatter.minimumIntegerDigits = 2
    }

    var pretty: String {
        let h = scoreFormatter.string(from: home as NSNumber)!
        let g = scoreFormatter.string(from: guest as NSNumber)!
        return "\(h) - \(g)"
    }
}

let score1 = ScoreStruct(home: 2, guest: 1)
score1.pretty // 02 - 01
```

In the initializer, we create a number formatter that's configured to show at least two integer digits, even if the score is less than 10. We use this formatter in the `pretty` property to produce the formatted output.

Now, let's make a copy of `score1` and then reconfigure the number formatter on this copy:

```
let score2 = score1
score2.scoreFormatter.minimumIntegerDigits = 3
```

Although we made the change on `score2`, the output of `score1.pretty` has changed as well:

```
score1.pretty // 002 - 001
```

The reason for this is that `NumberFormatter` is a class, i.e. the `scoreFormatter` property in our struct contains a reference to a number formatter instance. When we assigned `score1` to the new `score2` variable, a copy of `score1` was made. However, a copy of a struct is a copy of all its properties' values, and the value of `scoreFormatter` is just a reference. Therefore the `ScoreStruct` value in `score2` contains a reference to the same underlying number formatter instance as `score1` does.

Technically, `ScoreStruct` still has value semantics: when you assign an instance to another variable or pass it as a function parameter, the program makes a copy of the entire value. However, it depends what we consider to be the value. If we deliberately want to store a reference as one of the struct's properties, i.e. we're considering the reference itself as the value, then the struct above exhibits exactly the intended behavior. But we probably wanted the struct to include the number formatter instance itself, so that copies have their own formatters. In this case, the behavior of the struct above isn't correct.

To prevent the unexpected behavior in the example above, we could either change the type to be a class (so that the user of this type doesn't expect value semantics) or make the number formatter a private implementation detail so that it cannot be changed. The latter isn't a perfect solution though: we can still (accidentally) expose other public methods on the type that will mutate the number formatter internally.

We recommend being very careful about storing references within structs, since doing so will often result in unexpected behavior. However, there are cases where storing a reference is intentional and exactly what you need, mostly as an implementation detail

for performance optimizations. We'll look at an example of this in the next section, which covers copy-on-write.

Copy-On-Write Optimization

Value types require a lot of copying, since assigning a value or passing it on as a function parameter creates a copy. While the compiler tries to be smart about this and avoid copies when it can prove it's safe to do so, there's another optimization the author of a value type can make, and that's to implement the type using a technique called *copy-on-write*. This is especially important for types that can hold large amounts of data, like the standard library's collection types (Array, Dictionary, Set, and String). They are all implemented using copy-on-write.

Copy-on-write means that the data in a struct is initially shared among multiple variables; the copying of the data is deferred until an instance mutates its data. Since arrays are implemented using copy-on-write, if we create an array and assign it to another variable, the array's data hasn't actually been copied yet:

```
var x = [1, 2, 3]
var y = x
```

Internally, the array values in `x` and `y` contain a reference to the same memory buffer. This buffer is where the actual elements of the array are stored. However, the moment we mutate `x` (or `y` for that matter), the array detects that it's sharing its buffer with one or more other variables and makes a copy of the buffer before applying the mutation. This means we can mutate both variables independently, yet the potentially expensive copy of the elements only happens when it has to:

```
x.append(5)
y.removeLast()
x // [1, 2, 3, 5]
y // [1, 2]
```

Copy-on-write behavior isn't something we get for free for our own types; we have to implement it ourselves, just as the standard library implements it for its collection types. Implementing copy-on-write for a custom struct is only necessary in rare circumstances though, since the standard library already provides the most common types that deal with large amounts of data. Even if we define a struct that can contain a lot of data, we'll often use the built-in collection types to represent this data internally, and as a result, we benefit from their copy-on-write optimizations.

Nevertheless, understanding how copy-on-write can be implemented is helpful in understanding the behavior of Swift's collection types in general, along with that of some edge cases we should be aware of.

Copy-On-Write Tradeoffs

Before we look at the implementation of copy-on-write, we want to note that copy-on-write has its own tradeoffs. One advantage of value types is that they don't incur the overhead of reference counting. However, copy-on-write structs rely on storing a reference internally, and the internal reference count has to be incremented for every copy of a struct that gets created. So we're really giving up an advantage of value types — no need for reference counting — to mitigate against the potential cost of another property of value types — copy semantics.

Incrementing or decrementing a reference count is a relatively slow operation (compared to, say, copying a few bytes to another location on the stack) because such an operation must be thread-safe and therefore incurs locking overhead. Since all the variable-size types from the standard library — arrays, dictionaries, sets, and strings — rely on copy-on-write internally, all structs containing properties of these types also incur reference counting costs on each copy — potentially even multiple times when the type contains several of these properties (one exception to this is that of small strings up to 15 UTF-8 code units in size, for which Swift implements an [optimization](#) that avoids allocating a backing buffer altogether).

A practical example for this comes from the [SwiftNIO](#) project: an HTTP request used to be modeled as a struct in SwiftNIO, and it contained multiple properties, like the HTTP method, headers, etc. When such a struct was copied, not only did all its fields have to be copied, but the reference counts for all internal arrays, dictionaries, and strings had to be incremented too. This overhead resulted in significantly worse performance when passing around a value of this type (which was a very common operation), in comparison to passing around an HTTP request modeled as a class (since a reference to a class is smaller than all fields of the HTTP request struct, and only one reference count has to be updated).

Below, we'll look at how we can use the copy-on-write technique to combine the best of both worlds in this particular case: value semantics, and the performance advantage of using a class. [Johannes Weiss](#) of the SwiftNIO team also gave a great [talk about this at dotSwift 2019](#).

Implementing Copy-On-Write

We start with an extremely simplified version of an HTTP request struct:

```
struct HTTPRequest {  
    var path: String  
    var headers: [String: String]  
    // other fields omitted...  
}
```

To minimize the reference counting overhead outlined above, we'll first wrap all the properties in a private storage class:

```
struct HTTPRequest {  
    fileprivate class Storage {  
        var path: String  
        var headers: [String: String]  
        init(path: String, headers: [String: String]) {  
            self.path = path  
            self.headers = headers  
        }  
    }  
  
    private var storage: Storage  
  
    init(path: String, headers: [String: String]) {  
        storage = Storage(path: path, headers: headers)  
    }  
}
```

That way, our `HTTPRequest` struct only contains one property, `storage`, and it only requires one reference count of the internal storage instance to be incremented on copy. To expose the now private `path` and `headers` properties of the internal storage instance, we add computed properties to the struct:

```
extension HTTPRequest {  
    var path: String {  
        get { storage.path }  
        set {/* to do */}  
    }  
    var headers: [String: String] {
```

```
    get { storage.headers }
    set {/* to do */}
}
}
```

The important part is the implementation of the setters for these properties: we shouldn't just set the new value on the internal storage instance, because this object is potentially shared among multiple variables. Since storing the request's data in a class instance should be a private implementation detail, we have to make sure our class-based struct behaves exactly the same way as the original one. This means that mutating a property of an HTTP request variable should only change the value of that variable.

As a first step, we can create a copy of the internal storage class every time a setter is invoked. To make a copy, we add a `copy` method on `Storage`:

```
// Shadow Swift.print to capture print output for testing.
var printedLines: [String] = []
func print(_ value: Any) {
    var output = ""
    Swift.print(value, terminator: "", to: &output)
    printedLines.append(output)
    Swift.print(output)
}

extension HTTPRequest.Storage {
    func copy() -> HTTPRequest.Storage {
        print("Making a copy...") // For debugging
        return HTTPRequest.Storage(path: path, headers: headers)
    }
}
```

Then we can assign a copy of the current storage to the `storage` property before we set the new value:

```
extension HTTPRequest {
    var path: String {
        get {
            storage.path
        }
        set {

```

```
        storage = storage.copy()
        storage.path = newValue
    }
}
var headers: [String: String] {
    get {
        storage.headers
    }
    set {
        storage = storage.copy()
        storage.headers = newValue
    }
}
```

The `HTTPRequest` struct is now entirely backed by a class instance, but it still exhibits value semantics as if all its properties were properties on the struct itself:

```
let req1 = HTTPRequest(path: "/home", headers: [:])
var req2 = req1
req2.path = "/users"
assert(req1.path == "/home") // passes
```

However, the current implementation is still inefficient. We create a copy of the internal storage any time we make a change, no matter if any other variable references the same storage or not:

```
var req = HTTPRequest(path: "/home", headers: [:])
for x in 0..<5 {
    req.headers["X-RequestId"] = "\\(x)"
}
/*
Making a copy...
*/
```

Each time we mutate the request, another copy is made. All of these copies are unnecessary though; there's only the one `HTTPRequest` value in `req` that references the internal storage instance.

To provide efficient copy-on-write behavior, we need to know whether an object (the `Storage` instance in our case) is uniquely referenced, i.e. if it has a single owner. If it does, we can modify the object in place. Otherwise, we create a copy of the object before modifying it.

We can use the `isKnownUniquelyReferenced` function to find out if a reference has only one owner. If you pass an instance of a Swift class to this function, and if no one else has a strong reference to the object, the function returns `true`. If there are other strong references, it returns `false`.

There are a few subtle points to keep in mind when using `isKnownUniquelyReferenced`:

The function is thread-safe, but you have to make sure the variable that's being passed in isn't being accessed from another thread. (`isKnownUniquelyReferenced` isn't special in this regard; this limitation applies to every `inout` argument in Swift.) In other words, `isKnownUniquelyReferenced` doesn't protect against race conditions — this code isn't safe because both queues mutate the same variable concurrently.

```
var numbers = [1, 2, 3]
queue1.async { numbers.append(4) }
queue2.async { numbers.append(5) }
```

`isKnownUniquelyReferenced` uses an `inout` argument because that's the only way in Swift to refer to a *variable* in the context of a function argument. If the argument were passed normally, the compiler would always make a copy when the function is called, which means the object being tested could never be *uniquely* referenced inside the function body.

Unowned and weak references aren't considered, i.e. we have to make sure that no such references to the instance in question exist.

`isKnownUniquelyReferenced` doesn't work for Objective-C classes. To work around this limitation, we can box an Objective-C class instance within a Swift class.

Using this knowledge, we can now write a variant of `HTTPRequest` that checks if storage is uniquely referenced before mutating it. To avoid writing these checks in every property setter, we'll wrap the logic in a `storageForWriting` property:

```
extension HTTPRequest {
    private var storageForWriting: HTTPRequest.Storage {
        mutating get {
            if !isKnownUniquelyReferenced(&storage) {
                self.storage = storage.copy()
            }
            return storage
        }
    }

    var path: String {
        get { storage.path }
        set { storageForWriting.path = newValue }
    }

    var headers: [String: String] {
        get { storage.headers }
        set { storageForWriting.headers = newValue }
    }
}
```

To test our code, let's write the loop again:

```
var req = HTTPRequest(path: "/home", headers: [:])
var copy = req
for x in 0..<5 {
    req.headers["X-RequestId"] = "\\(x)"
} // Making a copy...
```

The debug statement only gets printed once: when we mutate `req` for the first time. In subsequent iterations, the uniqueness is detected and no copy is made. Combined with optimizations done by the compiler, copy-on-write avoids most of the unnecessary copies of value types.

willSet Defeats Copy-On-Write

Here's a performance trap to be aware of: the presence of a willSet observer on a property for a copy-on-write type defeats the copy-on-write-optimization. This is because willSet makes the variable newValue available within its body, forcing the compiler to create a temporary copy of the value. As a result, the value will no longer be uniquely referenced — any mutation of the property will trigger a copy.

To see this, we'll define three `HTTPRequest` properties and attach different change observers to them:

```
struct Wrapper {  
  var reqWithNoObservers = HTTPRequest(path: "/", headers: [:])  
  
  var reqWithWillSet = HTTPRequest(path: "/", headers: [:]) {  
    willSet {  
      print("willSet")  
    }  
  }  
  
  var reqWithDidSet = HTTPRequest(path: "/", headers: [:]) {  
    didSet {  
      print("didSet")  
    }  
  }  
}
```

Note that the properties are independent values, each with their own uniquely referenced storage. So mutating these values should *not* trigger copy-on-write, and that's exactly what we're seeing for the undecorated and `didSet` properties:

```
var wrapper = Wrapper()  
wrapper.reqWithNoObservers.path = "/about"  
wrapper.reqWithDidSet.path = "/forum"  
// didSet
```

The `willSet`-annotated property, however, *does* trigger copy-on-write on mutation:

```
wrapper.reqWithWillSet.path = "/blog"  
/*  
Making a copy...
```

```
willSet
```

```
*/
```

To understand this behavior, we have to consider the order of operations. Before `willSet` is called, a copy of the existing value is made. This copy is then mutated, `willSet` is called with `newValue` available in the body, and finally, the property's storage changes. For `didSet`, the behavior is similar: a copy is made before the mutation, and this copy is available through `oldValue`. However, in the example above, the compiler was smart enough to detect that we didn't use `oldValue` in our `didSet`, and it optimized the copy away. For `willSet`, it doesn't do this.

Consider a version of Swift with this optimization enabled for `willSet`. Just using `newValue` or not would change the order of operations: if you don't use `newValue`, the compiler would need to call `willSet` *before* the mutation, and any side effects inside `willSet` would happen before the side effects of the mutation. Whereas now, the side effects of the mutation always happen before `willSet`. Changing this behavior now would possibly break all programs that rely on this order of operations. For `didSet`, the Swift team was able to change the semantics.

The `willSet` semantics could become a performance problem in SwiftUI code because the `@Published` property wrapper used by `ObservableObject` uses `willSet` under the hood. For example, consider an array that acts as the source of truth for a long list view:

```
class ViewModel: ObservableObject {  
    @Published var cities: [String]  
}
```

Every mutation of the `cities` property will create a copy of the underlying array's storage.

Recap

In this chapter, we saw how structs (value types) and classes (reference types) have fundamentally different behavior despite their shared features. A value type variable simply contains the value, and each assignment to another variable or passing of it to a function creates a copy of the value. Meanwhile, a reference type variable contains a reference to the actual value. Assigning it to another variable or passing it to a function creates a copy of the reference and not the underlying value itself.

We discussed how mutability can be controlled by `let` and `var`, how the `mutating` keyword works, and how `inout` parameters are used. Finally, we showed how the copy-on-write optimization (that's used by many types in the standard library) works and how you can implement it for your own structs.

Enums

7

Structs and classes, which we discussed in the previous chapter, are examples of *record types*. A record is composed of zero or more *fields* (properties), with each field having its own type. Tuples also fall into this category: a tuple is effectively a lightweight anonymous struct with fewer capabilities. Records are such an obvious concept that we take them for granted. Almost all programming languages allow you to define composite types of this kind (early versions of BASIC and the original Lisp are perhaps the best-known exceptions). Even assembly programmers have always used the concept of records to structure data in memory, albeit without language support.

Swift's enumerations, or enums, belong to a fundamentally different category that's sometimes referred to as *tagged unions*, *variant types*, or *sum types*. In spite of sum types being a concept equally as powerful as records, support for them is much less widespread in mainstream programming languages. Sum types are commonplace in functional languages, though, and have become popular in newer languages such as Rust. In our opinion, enums are one of Swift's best features.

Overview

An enum consists of zero or more *cases*, with each case having an optional tuple-style list of associated values. In this chapter, we'll sometimes use the singular term "associated value" when we talk about a single case's associated value(s). A case can have multiple associated values, but you can think of these values as a single tuple.

Here's an enum for representing the alignment of a paragraph. The cases don't have associated values:

```
enum TextAlignement {  
    case left  
    case center  
    case right  
}
```

We saw in the [Optionals](#) chapter that Optional is a generic enum with two cases — none and some. The some case has an associated value for the boxed value:

```
@frozen enum Optional<Wrapped> {  
    /// The absence of a value.  
    case none
```

```
/// The presence of a value, stored as `Wrapped`.  
case some(Wrapped)  
}
```

(Ignore the `@frozen` attribute for now. We'll discuss it in the [Frozen and Non-Frozen Enums](#) section later on.)

The `Result` type, whose purpose is to represent the success or failure of an operation, has a similar shape but adds a second associated value (and corresponding generic parameter) for the failure case, enabling it to capture detailed error information:

```
@frozen enum Result<Success, Failure: Error> {  
    /// A success, storing a `Success` value.  
    case success(Success)  
    /// A failure, storing a `Failure` value.  
    case failure(Failure)  
}
```

We discuss `Result` in detail in the [Error Handling](#) chapter, and we'll also use it in many examples in this chapter.

You create an enum value by specifying one of its cases, plus values for the case's associated values, if it has any:

```
let alignment = TextAlign.left  
let download: Result<String, NetworkError> = .success("<p>Hello world!</p>")
```

Notice that in the second line, we have to provide the full type annotation, including all generic parameters. An expression like `Result.success(htmlText)` produces an error unless the compiler can infer the concrete type of the other generic parameter, `Failure`, from context. Having specified the complete type once, we can then rely on type inference using the leading-dot syntax. (The definition of `NetworkError` isn't shown here.)

Enums Are Value Types

Enums are value types, just like structs are. They have almost all the same capabilities structs have:

- Enums can have methods, computed properties, and subscripts.
- Methods can be declared mutating or non-mutating.
- You can write extensions for enums.
- Enums can conform to protocols.

However, enums cannot have stored properties. An enum's state is fully represented by its case plus the case's associated value. Think of the associated values as the stored properties for a particular case.

Mutating methods on enums work the same way they do on structs. We saw in the [Structs and Classes](#) chapter that inside a mutating method, `self` is passed inout and is hence mutable. Because enums don't have stored properties and there's no way to mutate a case's associated value directly, we mutate an enum by assigning a new value directly to `self`.

Enums don't require explicit initializers because the usual way to initialize an enum variable is to assign it a case. However, it's possible to add additional "convenience" initializers in the type definition or in an extension. For example, using the Locale API from Foundation, we can add an initializer to our `TextAlignment` enum that sets a default text alignment for a given locale:

```
extension TextAlignment {  
    init(defaultFor locale: Locale) {  
        guard let language = locale.languageCode else {  
            // Default value if language is n/a.  
            self = .left  
            return  
        }  
        switch Locale.characterDirection(forLanguage: language) {  
            case .rightToLeft:  
                self = .right  
                // Left is the default for everything else.  
            case .leftToRight, .topToBottom, .bottomToTop, .unknown:  
                self = .left  
            @unknown default:  
                self = .left  
        }  
    }  
}
```

```
let english = Locale(identifier: "en_AU")
TextAlignment(defaultFor: english) // left
let arabic = Locale(identifier: "ar_EG")
TextAlignment(defaultFor: arabic) // right
```

(We'll cover the `@unknown` default case in the [Frozen and Non-Frozen Enums](#) section.)

Sum Types and Product Types

An enum value contains exactly *one of* its cases (plus values for the case's associated values, if any). In fact, enums were [called “oneof”](#) and later [“union”](#) back in the early days of Swift (before the first public release). More concretely, a `Result` value contains either a success value *or* a failure value, but never both (and never none). In contrast, an instance of a record type contains values for *all of* its fields: a `(String, Int)` tuple contains a string *and* an integer. (Note that we talk about compound records with more than one field here; `UInt8` is a struct too, and you might say that it constrains instances to “one of 0...255.” But that's not what we mean.)

This ability to model “*or*” relationships is fairly unique, and it's what makes enums so useful. It allows us to write safer and more expressive code that takes full advantage of strong types in situations that often can't be expressed as cleanly with structs, tuples, or classes.

We say “fairly unique” because protocols and subclassing can be used for the same purpose, albeit with very different tradeoffs and applications. A variable of a protocol type (also called an [existential](#)) can be *one of* any type that conforms to the protocol. Similarly, an object of type `UIView` on iOS can also refer to any one of `UIView`'s direct or indirect subclasses, such as `UILabel` or `UIButton`. When working with such an object, we can either use the common interface defined on the base type (equivalent to calling methods defined on an enum), or attempt to downcast the instance to a concrete subtype to access data that's unique to that subtype (equivalent to switching over an enum).

The difference lies in which approach is more common — either dynamic dispatch through the common interface for protocols and classes, or switching for enums — and also in the particular capabilities and limitations the constructs have. For example, the list of cases of an enum is fixed and can't be extended retroactively, whereas you can always conform one more type to a protocol or add another subclass (though subclassing across module boundaries is restricted unless you explicitly declare a class as open).

Whether this freedom is desirable or even required depends on the problem to be solved. As value types, enums are also generally more lightweight and better suited for modeling “plain old values.”

There’s a neat correspondence between the two categories of types (“or” and “and”) and the mathematical concepts of addition and multiplication. Knowing about it isn’t essential to be a good Swift programmer, but we find it a helpful line of thinking when designing custom types.

There are many possible definitions for the term “type.” Here’s one: a type is the set of all possible values, or inhabitants, its instances can assume. `Bool` has two inhabitants, `false` and `true`. `UInt8` has 2^8 (256) inhabitants. `Int64` has 2^{64} (about 18.4 quintillion) inhabitants. Types such as `String` have infinitely many inhabitants — you can always create another string by adding one more character (at least until you’ve filled up your computer’s memory).

Now consider a tuple of two Boolean fields: `(Bool, Bool)`. How many inhabitants does this type have? The answer is four: `(false, false)`, `(true, false)`, `(false, true)`, and `(true, true)`. It’s impossible to construct any other value of this type except these four. What if we add another `Bool`, making it `(Bool, Bool, Bool)`? The number of inhabitants doubles to eight since each of the previous four inhabitants can be combined with `false` and `true`, respectively. This works not only with `Bool`s, of course. A `(Bool, UInt8)` pair has $2 \times 256 = 512$ inhabitants because each of the 256 `UInt8` inhabitants can be paired with one of the two Boolean values.

Generally speaking, the number of inhabitants of a tuple (or struct, or class) is equal to the *product* of the inhabitants of its members. For this reason, structs, classes, and tuples are also called **product types**.

Compare this to enums. Here’s an enum with three cases:

```
enum PrimaryColor {  
    case red  
    case yellow  
    case blue  
}
```

This type has three inhabitants — one per case. It’s impossible to construct any other `PrimaryColor` value than `.red`, `.yellow`, or `.blue`. What happens if we add associated values into the mix? Let’s add a fourth case that allows us to specify a grayscale value between 0 (black) and 255 (white):

```
enum ExtendedColor {  
    case red  
    case yellow  
    case blue  
    case gray(brightness: UInt8)  
}
```

The `.gray` case alone has 256 possible values, resulting in $3 + 256 = 259$ inhabitants for the entire enum. Generally speaking, the number of inhabitants of an enum is equal to the *sum* of the inhabitants of its cases. This is why enums are also called **sum types**.

Adding a field to a struct multiplies the number of possible states, often enormously. Adding a case to an enum only adds one additional inhabitant (or, if the case has an associated value, it adds the payload's inhabitants). This is a useful property for writing safe code, and the [Designing with Enums](#) section later in this chapter covers how to take advantage of this property in our code.

Pattern Matching

To do something useful with an enum value, we commonly have to inspect its case and extract the associated value. Take optionals as an example: every operation involving optionals — be it if let binding, optional chaining, or calling `Optional.map` — is shorthand for unwrapping the associated value of the `some` case and processing it further. If the inspected value is `none`, the operation usually aborts.

The most common way to inspect an enum is with a `switch` statement, which lets us compare a value against multiple candidates in a single statement. As an added bonus, `switch` has a convenient syntax for comparing a value against a particular case and extracting the associated values in one go; this mechanism is called **pattern matching**. Pattern matching isn't exclusive to `switch` statements, but they are its most prominent use case.

Pattern matching is useful because it lets us decompose a data structure by its *shape* instead of just its contents. The ability to combine pure matching with value binding makes it especially powerful.

Each case in a `switch` statement begins with one or more *patterns* the input value is matched with. A pattern describes the structure of a value. For instance, the pattern `.success((42, _))` in the following example matches the `success` case of an enum where

the associated value is a pair whose first element is equal to 42. The underscore is the wildcard pattern — the pair’s second element can be any value. In addition to plain matching, we can extract parts of a composite value and bind them to variables. The pattern `.failure(let error)` matches the failure case *and* binds the associated value to a new local constant, `error`:

```
let result: Result<(Int, String), Error> = ...
switch result {
case .success((42, _)):
    print("Found the magic number!")
case .success(_):
    print("Found another number")
case .failure(let error):
    print("Error: \(error)")
}
```

Let’s look at [the pattern types Swift supports](#).

Wildcard pattern — The underscore, `_`, matches any value and ignores it. Wildcards are often used for ignoring one part of an associated value while matching another. We saw an example of this above with `.success((42, _))`. In switch statements, `case _` is equivalent to the default keyword: both match any value and only make sense as the last case of the switch.

Tuple pattern — This matches tuples with a comma-separated list of zero or more subpatterns. As an example, `(let x, 0, _)` matches a tuple with three elements — where the second element is 0 — and binds the first element to `x`. The tuple pattern itself only matches the structure of a tuple, i.e. comma-separated values enclosed in parentheses. The tuple contents are subpatterns that are matched separately (in this example, a value-binding pattern, an expression pattern, and a wildcard pattern). Tuple patterns are useful for switching over multiple values in a single switch statement.

Enum case pattern — This matches the specified enum case. The pattern can include subpatterns for the associated values, be it for equality checks (`.success(42)`) or for value binding (`.failure(let error)`). To ignore an associated value, use an underscore or omit the pattern altogether, e.g. `.success(_)` and `.success` are equivalent.

Enum case patterns are the only way to extract an enum’s associated value or match against a `case` while ignoring the associated value. (For comparing against a specific case with a specific associated value, you can also use `==` in an if statement, assuming the enum is Equatable.)

Value-binding pattern — This binds part or all of a matched value to a new constant or variable. The syntax is `let someldentifier` or `var someldentifier`. The scope of the new variable is the case block in which it appears.

As a shorthand for multiple value bindings in a single pattern, you can prefix the pattern with a single `let` instead of repeating `let` for each binding. The patterns `let (x, y)` and `(let x, let y)` are equivalent. Notice the subtle difference when using value binding and equality matching in a single pattern: the pattern `(let x, y)` binds a tuple's first element to a new constant but compares the tuple's second element to the existing variable, `y`.

To combine value binding with other conditions the bound values must satisfy, you can extend a value-binding pattern with a `where` clause. For example,

`.success(let httpStatus) where 200..<300 ~= httpStatus` only matches success values with an associated value that falls into the specified range. Crucially, the `where` clause is evaluated *after* the value-binding step, so we can use the bound variables in the `where` clause. (For more on the pattern matching operator `~=`, see the Expression Patterns section below.)

If you include multiple patterns in a single case, all patterns must use the same names and types in their value bindings. Suppose you want to switch over the following enum:

```
enum Shape {  
    case line(from: Point, to: Point)  
    case rectangle(origin: Point, width: Double, height: Double)  
    case circle(center: Point, radius: Double)  
}
```

Notice that each of the cases' associated values contains the shape's origin point, but the other parameters vary depending on the kind of shape. Nonetheless, it's possible to extract the origin point of a shape with a single `case` statement containing three patterns:

```
switch shape {  
    case .line(let origin, _),  
        .rectangle(let origin, _, _),  
        .circle(let origin, _):  
            print("Origin point:", origin)  
}
```

You cannot include other value bindings in this case, e.g. for a circle's radius, because the compiler guarantees that each bound variable contains a valid value when one of the

patterns matches. Therefore, the compiler must be able to assign a valid value to each variable, and it cannot do this for the radius if shape turns out to be a line or rectangle.

Optional pattern — This provides syntactic sugar for matching and unwrapping optional values by using the familiar question mark syntax. The pattern `let value?` is equivalent to `.some(let value)`, i.e. it matches when the optional is non-nil and binds the unwrapped value to a constant.

As we saw in the [Optionals](#) chapter, we can also use `nil` to match an optional's `none` case. This shorthand requires no special compiler magic. It works as a normal expression pattern (see below) because the standard library includes [an overload of the `~ =` operator](#) for comparing optionals against `nil`.

Type-casting pattern — The pattern `is SomeType` matches if the value's runtime type is the same as the specified type or a subclass of that type. `let value as SomeType` performs the same check and additionally casts the matched value to the specified type, whereas `is` just checks the type:

```
let input: Any = ...
switch input {
    case let integer as Int: ... // integer has type Int.
    case let string as String: ... // string has type String.
    default: fatalError("Unexpected runtime type: \(type(of: input))")
}
```

Expression pattern — This matches against an expression by feeding the input value and the pattern to the pattern matching operator, `~ =`, which is defined in the standard library. The default implementation of `~ =` for Equatable types forwards to `==`; this is how simple equality checks work in patterns.

The standard library also provides overloads of `~ =` for ranges. This makes possible a nice syntax for checking if a value falls inside a range, especially when combined with one-sided ranges. The following switch statement tests whether a number is positive, negative, or zero:

```
let randomNumber = Int8.random(in: .min...(max))
switch randomNumber {
    case ..<0: print("\(randomNumber) is negative")
    case 0: print("\(randomNumber) is zero")
    case 1...: print("\(randomNumber) is positive")
```

```
default: fatalError("Can never happen")
}
```

Note that the compiler forces us to include a default case because it cannot determine that the three concrete cases cover all possible inputs (even though they do), and switch statements must always be exhaustive. We'll talk more about exhaustiveness checking in the [Designing with Enums](#) section.

We can extend the pattern matching system by overloading the `~=` operator for our custom types. The function implementing `~=` must have the following shape:

```
func ~= (pattern: ???, value: ???) -> Bool
```

The argument types can be chosen freely (they don't even have to be the same). The compiler will pick the most specific overload that works with the types of the input values. For each expression pattern the compiler encounters, it evaluates the expression pattern `~= value`, where `value` is the value we're switching over, and `pattern` is the pattern in the `case` statement. The match succeeds if the expression returns true.

We should note that, apart from in toy programs, we never found it necessary to extend pattern matching in this way. The standard library covers the basics pretty well, and anything that goes beyond the basics suffers too much from the inability to combine custom `~=`-based pattern matching with value-binding and wildcard patterns.

Pattern Matching in Other Contexts

Pattern matching is the only way to extract the associated value from an enum. But pattern matching isn't exclusive to enums, nor is it exclusive to switch statements. In fact, an assignment such as `let x = 1` can be seen as a value-binding pattern on the left side of the assignment operator matching the expression on the right side. Other pattern matching examples include:

Destructuring tuples in assignments, e.g. `let (word, pi) = ("Hello", 3.1415)` — and in loops, e.g. `for (key, value) in dictionary { ... }`. Notice that the `for` loop doesn't use `let` to indicate value binding. By default, all identifiers are value bindings in this case. `for` loops also support `where` clauses. For example,
`for n in 1...10 where n.isMultiple(of: 3) { ... }` only executes the loop body for 3, 6, and 9.

Tuple patterns can be nested to destructure values in nested tuples. Example:
for (num, (key: k, value: v)) in dictionary.enumerated() { ... }.

Using wildcards to ignore values we're not interested in. For example, for `_` in `1...3` executes a loop three times without creating a variable for the loop counter, and `_ = someFunction()` suppresses the compiler's “unused result” warning when we want to execute a function for its side effects.

Catching errors in a catch clause: do { ... } catch let error as NetworkError { ... }. Refer to the [Error Handling](#) chapter for more on this.

if case and guard case statements are similar to a switch statement that only has a single case. These are occasionally useful because they require fewer lines than a switch, though we prefer the latter in many situations in order to take advantage of the compiler's exhaustiveness checks.

The syntax of if/guard case [let] is often a big hurdle for newcomers to Swift. We think this is because it uses the assignment operator = for what's fundamentally a comparison operation and only optionally includes value bindings. For example, the following tests if an enum is a specific case but ignores the associated value:

```
let color: ExtendedColor = ...
if case .gray = color {
    print("Some shade of gray")
}
```

You can think of the assignment operator as “perform a pattern match of the value on the right-hand side with the pattern on the left-hand side.” It becomes clearer when you include a value binding, which uses the same syntax, only adding let or var:

```
if case .gray(let brightness) = color {
    print("Gray with brightness \(brightness)")
}
```

This isn't so different from the familiar if let `x = x` syntax that's used for optionals. In fact, Swift creator Chris Lattner [regrets](#) that the Swift developers chose to add if case [let] at all. If the if let syntax for optionals used a true optional pattern, including the question mark (if let `x? = x`), the language could just accept any valid pattern in if conditions and if case wouldn't be necessary.

for case and while case loops work in a way that's similar to if case. They allow you to only execute the loop when the pattern match succeeds. Refer to the [Optionals](#) chapter for examples.

Lastly, argument lists in closure expressions sometimes look like patterns because they also support a kind of tuple destructuring. For example, we can map over a dictionary and use a (key, value) parameter list inside the transformation closure, even though the function passed into map is specified to have a single Element parameter (Dictionary.Element is a (Key, Value) tuple type):

```
dictionary.map { (key, value) in  
    ...  
}
```

Here, (key, value) looks like a tuple but isn't — it's a function parameter list with two items. The fact that we can unpack the tuple inside the parameter list is thanks to special handling in the compiler that's unrelated to pattern matching. Without this feature, we'd have to use a single-item parameter list such as { element in ... } and then destructure element (which now is a real tuple) into key and value in a separate line.

Designing with Enums

Because enums belong to a different category of types than structs and classes do, they lend themselves to different design patterns. And since true sum types are a relatively uncommon (if rapidly growing) feature among mainstream programming languages, chances are you aren't as used to working with them as you are with traditional object-oriented approaches.

So, let's look at some of these patterns we can use in our code to take full advantage of enums. We've split them into six main points:

0. Switching exhaustively
1. Making illegal states impossible
2. Modeling state with enums
3. Choosing between enums and structs
4. Drawing parallels between enums and protocols
5. Modeling recursive data structures with enums

Switching Exhaustively

For the most part, switch is just a more convenient syntax for an if case statement with multiple else if case conditions. Syntax differences aside, there's one important distinction: **a switch statement must be exhaustive**, i.e. its cases must cover all possible input values. The compiler enforces this.

Exhaustiveness checking is an important tool for writing safe code and for keeping code correct as programs change. Every time you add a case to an existing enum, the compiler can alert you to all the places where you switch over this enum and need to handle the new case. Exhaustiveness checking isn't performed for if statements, nor does it work in switch statements that include a default case — such a switch can never not be exhaustive, since default matches any value.

For this reason, we recommend you **avoid default cases in switch statements** if at all possible. You can't avoid them completely because the compiler isn't always smart enough to determine if a set of cases is in fact exhaustive. We saw an example of this above when we switched over an Int8 and our range patterns covered all possible values. The compiler only ever errs on the side of safety, i.e. it'll never report a non-exhaustive set of patterns as exhaustive (barring bugs in the compiler's implementation).

False negatives aren't a problem when switching over enums though. Exhaustiveness checks are completely reliable for the following types:

- Bool
- Enums, as long as any associated values can be checked exhaustively or you match them with patterns that match any value (wildcard or value-binding pattern)
- Tuples, as long as their member types can be checked exhaustively

Let's look at an example. Here we switch over the Shape enum we defined above:

```
let shape: Shape = ...
switch shape {
  case let .line(from, to) where from.y == to.y:
    print("Horizontal line")
  case let .line(from, to) where from.x == to.x:
    print("Vertical line")
  case .line(_, _):
    print("Oblique line")
```

```
case .rectangle, .circle:  
    print("Rectangle or circle")  
}
```

We include two `where` clauses to treat horizontal (equal `y` coordinates) and vertical (equal `x` coordinates) lines as special cases. These two cases aren't enough to cover the `.line` case exhaustively, so we need another case that catches all the remaining lines. Although we're not interested in distinguishing between `.rectangle` and `.circle` here, we prefer listing the remaining cases explicitly over using a default case, as this allows us to take advantage of exhaustiveness checking.

By the way, the compiler also verifies that each pattern in a `switch` carries its weight. The compiler will emit a warning if it can prove that a pattern will never match because it's already covered in full by one or more preceding patterns.

Exhaustiveness checking has the most benefits if an enum and the code that uses it evolve in sync, i.e. every time a case is added to an enum, the code that switches over that enum can be updated at the same time. This is usually true if you have access to the source code of your program's dependencies and the program and its dependencies are compiled together. Things become more complicated when a library is distributed in binary form and the program using the library must be prepared to work with a newer version of the library than was known when the program was compiled. In this situation, it can be necessary to always include a default case, even in otherwise exhaustive switches. We'll come back to this in the [Frozen and Non-Frozen Enums](#) section later in this chapter.

Making Illegal States Impossible

There are many good reasons for using a statically typed programming language such as Swift. Performance is one: the more the compiler knows about the types of the variables in a program, the faster the code it can generate is (in general).

Another equally important reason is that types can guide developers as to how APIs should be used. If you pass a value of the wrong type to a function, the compiler will complain immediately. We might call this *compiler-driven development* — seeing the compiler not as an enemy you have to fight, but as a tool that, by using type information, leads you almost magically to a correct solution:

- A function with carefully chosen input and output types allows less space for misuse because the types establish an “upper bound” for the function’s behavior. For example, if you’re implementing a function that takes a non-optional object as a parameter, you can be sure the object will never be nil inside the function’s body. How well this works depends on how finely we can constrain our types to only accept valid values. Enums are often the perfect tool to define the range of permissible values precisely.
- Statically checked types prevent certain categories of errors outright; code that doesn’t compile because it violates the constraints set by the type system will never have to be dealt with at runtime.
- Types act as documentation that never goes out of sync. Unlike comments, which people might forget to update when code is modified, the types are an integral part of the program and are always up to date.

Not every aspect can be expressed in the type system, of course. For example, Swift provides no support for conveying that a function is pure (i.e. it has no side effects) or what its performance characteristics are. That’s why we still need documentation, and developers must take care not to violate documented guarantees as they update existing code. But it should be obvious that the amount of help you can get from the compiler grows with the capabilities of the type system. (We should note that it’s certainly possible for a programming language to overdo it. Giving the compiler more information requires more work from the developer that, while often helpful, may sometimes get in the way of solving the actual task. Plus, the more finely tuned your types are for a specific use case, the more code you have to write to transform values between types. We don’t think Swift has reached this point, but there’s no free lunch.)

Here’s our recommendation for designing custom types so as to maximize the help you get from the compiler: **use types to make illegal states unrepresentable**. Earlier, we saw in the Sum Types and Product Types section that adding a case to an enum adds exactly one possible value to the type. You can’t get more fine-grained than that, which makes enums so useful for this purpose.

The canonical example is `Optional`, which, through its `none` case, adds a single inhabitant to the wrapped type. This is precisely what’s needed to represent the absence of a value without resorting to sentinel values. We discussed the problem with sentinel values in the Optionals chapter.

Let’s look at an API that’s harder to use than it should be because it doesn’t follow the above guideline. A common pattern for asynchronous operations (such as performing a network request) in Apple’s iOS SDK is to pass a completion handler (a callback function)

to the method you're calling. The method will then call the handler when the task completes, passing the result of the operation along. Because most asynchronous operations can fail, the result can usually be either some value indicating success, e.g. the server's response, or an error. In the future, we'll probably see fewer methods that take callbacks, and we'll instead see more `async` methods. But for now, they're still common.

Consider [the geocoding API in Apple's Core Location framework](#). You pass it an address string and a callback function. The geocoder contacts a server that returns matching placemark objects for the address. The geocoder then calls the completion handler with the placemarks or an error:

```
class CLGeocoder {  
    func geocodeAddressString(_ addressString: String,  
        completionHandler: @escaping ([CLPlacemark]?, Error?) -> Void)  
    // ...  
}
```

Observe the type of the completion handler, `([CLPlacemark]?, Error?) -> Void`. Its two parameters are both optionals. This means there are four possible states this function can communicate back to the caller: `(.some, .none)`, `(.none, .some)`, `(.some, .some)`, or `(.none, .none)`. (This is a simplified view; the `.some` states really have infinitely many possible values, but we're only concerned with whether they're nil or non-nil.) The problem is that, of the four legal states, only the first two make sense in practice. What's a developer supposed to do if they receive an array of placemarks *and* an error? Even worse, what if both values come back nil? The compiler can't help you here because the types are less precise than they should be.

Now Apple probably took care when implementing this method to never return one of these invalid states, so they'll never occur in practice. But users of the API can't be sure of that, and even if it's true today, there's no guarantee it'll still be true in the next SDK release.

The geocoding API would be much more developer friendly if it replaced the two optionals with a `Result<[CLPlacemark], Error>` value:

```
extension CLGeocoder {
    func geocodeAddressString(_ addressString: String,
        completionHandler: @escaping (Result<[CLPlacemark], Error>) -> Void) {
        // ...
    }
}
```

The `Result` type represents either a success or a failure, but never both and never none. By using a type that makes the invalid states unrepresentable, the API becomes easier to use, and a whole range of potential bugs simply cannot happen because the compiler doesn't allow them. Many of Apple's iOS APIs don't take full advantage of Swift's type system because they're written in Objective-C, which has no equivalent concept to enums with associated values. But that doesn't mean we can't do better in Swift.

Starting with iOS 15 and macOS 12, `CLGeocoder` also provides an `async` API that either returns a `[CLPlacemark]` or throws an error. This new API is even more precise than our rewritten example above, because we know it'll always either return a value or throw an error, whereas an API that takes a completion handler could call the handler zero, one, or more times.

When writing a function, think carefully about the parameter and return types. The tighter you can constrain the types to the set of valid input and output values, the more help the compiler can give you (when implementing the function) and users of the API (when calling it).

By the way, there's another interesting state in the geocoding API that we've ignored until now: what if the returned array of placemarks is empty? The documentation seems to say this should never happen, i.e. if the server can't find a match for the input string, the function will return an error. But there's another possible interpretation: an empty array could signal that the request itself was successful (no network error, etc.) but no match was found. Just by looking at the types, we can't be sure which interpretation is correct. If we wanted to encode the first interpretation in the type system, we'd need an array type that provided a compile-time guarantee never to be empty. The standard library doesn't provide this, but we could write our own. The basic idea is to write a struct that uses a separate property for the array's first element (called `head`) and a standard array (which can be empty) for the remaining elements (the `tail`):

```
struct NonEmptyArray<Element> {
    private var head: Element
```

```
    private var tail: [Element]  
}
```

Since `head` is non-optional, it's impossible to create a `NonEmptyArray` value that doesn't contain at least one element. A complete implementation of `NonEmptyArray` should conform to all the same protocols `Array` adopts, most notably `Collection`. This would make it as convenient to use as a normal array — sometimes even more so, because we can overload some `Collection` APIs such as `first` and `last` to return non-optional values. If you'd like to pursue this, check out the [NonEmpty library](#) written by Brandon Williams and Stephen Celis. It's an implementation of this pattern that's generic over the wrapped collection type (so you could also have a non-empty string, for example). And for an in-depth discussion of Swift's collection protocols, see the [Collection Protocols](#) chapter.

Modeling State with Enums

We can apply this goal of making illegal values unrepresentable to another major aspect of application design: how to model state in our programs. A program's *state* is the contents of all variables at a given point in time, plus (implicitly) its current execution state, i.e. which threads are running and which instruction they're executing. The state "remembers" things like what mode an application is in, what data it's displaying, which user interaction it's currently processing, and so on. All but the most trivial programs are *stateful*: what happens next when a particular instruction is executed depends on the current state the system is in. (HTTP is an example of a *stateless* protocol, meaning that servers must process HTTP requests without considering previous requests from the same client. Web developers have to use features like cookies to remember state across multiple requests. But even though HTTP is stateless, a program processing HTTP requests would still be stateful so as to maintain its internal state.)

As a program runs, it changes its state in reaction to external events such as user interactions or incoming data from the network. This can take place implicitly without the developer thinking much about it — after all, state mutations happen all the time. But as an application grows more complex, it's a good idea to make a conscious effort to define the possible states the program (or one of its subsystems) can be in, as well as the legal transitions between states. The set of states a system can be in is also known as its *state space*.

Try to make your program's state space as small as possible. The smaller the state space, the easier your job as a developer becomes — a smaller state space reduces the number of cases your code has to deal with. Because enums model a finite number of

states, they're ideal for modeling state and transitions between states. And because each state, or enum case, carries its own data (in the form of associated values), it's easy to make illegal state combinations unrepresentable, as we saw in the previous section.

(We should note that it's likely your program's state space is technically infinitely large, especially if you accept user input in the form of text or uploaded images, etc. These data types naturally have a [nearly] infinite number of inhabitants. But just like in the previous section, where we only cared if a value was nil or non-nil, this is usually not a problem. The essential parts of most systems' state are generally finite and often few in number; otherwise, we couldn't model them in code.)

Let's look at an example. Suppose we're writing a chat app. When the user opens a chat channel, the app should display a spinner while it's loading the list of messages from the network. When the network request completes, the UI transitions either to displaying the messages or to showing an error if the request failed. Let's first consider how we'd model the app's state in the traditional way without enums (technically we're still using enums because we'll be using optionals, but you get the idea). We could use three variables — a Boolean that we set to true while the network request is in progress, and two optionals for the list of messages and the error, respectively:

```
struct StateStruct {  
    var isLoading: Bool  
    var messages: [Message]?  
    var error: Error?  
}  
  
// Set initial state.  
var structState = StateStruct(isLoading: true, messages: nil, error: nil)
```

Both `messages` and `error` should be `nil` while loading, and then one of them is assigned a value when the network request completes. They should never both be non-nil at the same time, nor should `isLoading` be true while any of them is non-nil.

Recall our discussion on [sum types vs. product types](#) and how to determine the number of inhabitants a type has. The `StateStruct` struct is a product type that has $2 \times 2 \times 2 = 8$ possible states: any combination of true or false for the Boolean and `none` or `some` for either of the two optionals (again, we're ignoring the infinitely many states of the `some` cases because they're not relevant for this discussion). This is a problem because our app only needs to handle three of those eight states: loading, displaying a list of messages, or displaying an error. The other five are invalid combinations that should

never occur if we wrote our program correctly, but we can't expect the compiler to alert us if we create an invalid state.

Now, let's model our state as a custom enum with the three states loading, loaded, and failed:

```
enum StateEnum {  
    case loading  
    case loaded([Message])  
    case failed(Error)  
}  
  
// Set initial state.  
var enumState = StateEnum.loading
```

You'll notice right away that setting the initial state becomes much cleaner because we don't have to concern ourselves with properties that aren't relevant to the initial state. Furthermore, we completely eliminated the chance to end up in an invalid state. Because each state carries its own associated data, the associated values for loaded and failed don't have to be optional. As a result, it's impossible to transition to the failed state unless we actually have an Error value in our code. (Things are a little less clear for the loaded state because you can always assign an empty array, but that isn't something you're likely to do accidentally.) When our program is in a particular state, we can be certain that all necessary data for that state is also available. Our StateEnum enum acts as the basis for a state machine.

Enums are *not* full [finite-state machines](#) because they lack the ability to specify illegal state transitions — for instance, in our simple example, it shouldn't be possible to transition from loaded to failed or vice versa. In practice, being unable to instantiate a state unless you have valid values for all associated data is almost as good, though: in a well-designed program, it's unlikely you'll find many places in the code where all associated data for a state is available, but transitioning to that state would still be an invalid operation.

Every time our code needs to access some state-dependent data (e.g. the messages array), we're now forced to switch over the state enum to extract the associated values. This can sometimes feel inconvenient because the switch syntax is quite heavy-handed. But it's an important safety feature because it forces us to always handle every possible

state — at least if we observe the guideline not to use default cases in our switch statements.

By the way, you may have noticed that the struct we started out with and the enum we replaced it with aren't the only ways to model this piece of state. In fact, the StateStruct.isLoading property is redundant because in our design, isLoading should be true if and only if messages and error are both nil. We could make isLoading a computed property without losing anything:

```
struct StateStruct2 {
    var messages: [Message]?
    var error: Error?

    var isLoading: Bool {
        get { return messages == nil && error == nil }
        set {
            messages = nil
            error = nil
        }
    }
}
```

This reduces the number of possible states from eight to four, leaving only a single invalid state (when messages and error are non-nil) — not perfect, but better than what we started with. It's often difficult to notice redundant properties like this, but this is where the connection between inhabitants of a type and algebra can really help us. If, as in this example, we determine that our custom type has $2 \times 2 \times 2$ inhabitants, but only three of them are valid, it's easy to see that one of the factors is redundant: 2×2 is enough room for three states, so it must be possible to eliminate one component.

The pattern of having two mutually exclusive optional values might also remind you of the example we used in the previous section where we replaced `([CLPlacemark]?, Error?)` with `Result<[CLPlacemark], Error>`. Applying the same pattern to our example would produce `Result<[Message], Error>`, but note that the two situations aren't completely identical; the chat app requires a third state, “loading,” where messages and error are both nil. Nesting the Result in an optional accomplishes this (recall that wrapping a type in an optional always adds exactly one inhabitant), resulting in this alternative representation of our state:

```
/// nil means "loading."
typealias State2 = Result<[Message], Error>?
```

This is equivalent to our custom enum, i.e. it's a type with three states and the same payloads for the states. (`Result<[Message]?, Error>` would be another equivalent variant.) But semantically, it's arguably a weaker solution, because it's not immediately clear that `nil` stands for the “loading” state.

Our example only models the state of a single subsystem of our application as an enum. But you can push this pattern much further and model the state of your entire program as a single enum — usually one that has lots of nested enums and structs that break the state down for individual subsystems. The idea is to have a single variable that captures the program’s state in its entirety. All state changes go through this one variable that you can then observe (e.g. by using `didSet`) to update your app’s UI when a state change occurs. This design also makes it easy to write the entire app state to disk and read it back on the next launch, essentially giving you state restoration for free. If you’d like to learn more about this, check out our book [*App Architecture*](#), which Chris and Florian wrote together with Matt Gallagher.

Although you can model your entire app state as an enum, what’s nice about the enums-as-state pattern is you don’t have to go all in to benefit from it. You can start small by converting a single subsystem (e.g. one screen) and see how it works. Then, gradually work your way up the hierarchy by wrapping your state enums for subsystems in a new enum that has one case per subsystem.

To summarize, an enum is a great match for modeling state. It can largely prevent invalid states, and keeping the entire state for a subsystem (or even an entire program) in a single variable makes state transitions a lot less error-prone. Furthermore, exhaustive switching allows the compiler to point out code paths that need updating as you add new states or change their associated values.

Choosing between Enums and Structs

Earlier in this chapter, we discussed how enums and structs have very different properties: an enum value represents exactly one of its cases (plus its associated values), whereas a struct value represents the values of all its properties. Despite these differences, it’s not uncommon to encounter problems that can be solved with either an enum or a struct.

Using an example inspired by [a blog post by Matt Diephouse](#), we’ll create a data type for an analytics event using an enum and a struct. Here’s the enum variant:

```
enum AnalyticsEvent {
    case loginFailed(reason: LoginFailureReason)
    case loginSucceeded
    ... // more cases.
}
```

This enum is then extended with several computed properties that switch over the enum and return the data required by users of the type, i.e. the actual strings and dictionaries that should be sent to the server:

```
extension AnalyticsEvent {
    var name: String {
        switch self {
            case .loginSucceeded:
                return "loginSucceeded"
            case .loginFailed:
                return "loginFailed"
            // ... more cases.
        }
    }

    var metadata: [String: String] {
        switch self {
            // ...
        }
    }
}
```

Alternatively, we could model the same analytics event as a struct, storing its name and metadata in two properties. We provide static methods (which correspond to the enum cases from above) to create instances for specific events:

```
struct AnalyticsEvent {
    let name: String
    let metadata: [String : String]

    private init(name: String, metadata: [String: String] = [:]) {
        self.name = name
        self.metadata = metadata
    }
}
```

```

static func loginFailed(reason: LoginFailureReason) -> AnalyticsEvent {
    return AnalyticsEvent(
        name: "loginFailed"
        metadata: ["reason" : String(describing: reason)]
    )
}

static let loginSucceeded = AnalyticsEvent(name: "loginSucceeded")
// ...
}

```

Since we declared the initializer as private, the public interface is identical to the enum variant: where the enum exposes cases like `.loginFailed(reason:)` or `.loginSucceeded`, the struct exposes static methods or properties. The name and metadata properties are available in both variants, either as computed properties (in the enum) or as stored properties (in the struct).

However, each version of the `AnalyticsEvent` type has its distinct characteristics that can become advantages or drawbacks, depending on what your requirements are:

- If we make the struct's initializer internal or public, the struct can be extended in other files or even other modules with additional static methods or properties, thereby adding new analytics events to the API. This isn't possible with the enum variant: you can't add new cases to an enum retroactively.
- The enum models the data more precisely; it can only represent one of its predefined cases, whereas the struct can potentially represent an infinite number of values in its two properties. The precision and safety of the enum come in handy if you want to perform additional processing on the events, like coalescing sequences of events.
- The struct can have private “cases” (i.e. static methods or static properties that aren't visible to all clients), whereas the enum's cases always have the same visibility as the enum itself.
- You can switch exhaustively over the enum, making sure not to miss a type of event. But due to the strictness of switching over an enum, adding an additional event type to the enum is a potentially source-breaking change for users of this API, whereas you can add static methods for new event types to the struct without affecting other code.

Drawing Parallels between Enums and Protocols

At first glance, enums and protocols may not seem to have a lot in common. But there are in fact some interesting parallels between the two. In the [Sum Types and Product Types](#) section, we mentioned that enums aren't the only construct that can express "one of" relationships; protocols can be used for this purpose as well. In this section, we'll look at an example of this and discuss the differences between the two approaches.

Let's start with a type that we used earlier in this chapter — an enum to model a number of different shapes in a drawing app:

```
enum Shape {  
    case line(from: Point, to: Point)  
    case rectangle(origin: Point, width: Double, height: Double)  
    case circle(center: Point, radius: Double)  
}
```

A shape can be a line, a rectangle, or a circle. To render these shapes into a Core Graphics context, we add a `render` method in an extension. The implementation must switch over `self` and execute the appropriate drawing commands for each case:

```
extension Shape {  
    func render(into context: CGContext) {  
        switch self {  
            case let .line(from, to): // ...  
            case let .rectangle(origin, width, height): // ...  
            case let .circle(center, radius): // ...  
        }  
    }  
}
```

Alternatively, we can use a protocol to define a shape as any type that can render itself into a Core Graphics context:

```
protocol Shape {  
    func render(into context: CGContext)  
}
```

The shape types we expressed as enum cases above now become concrete types that conform to the `Shape` protocol. Each conforming type implements its own `render(into:)` method:

```

struct Line: Shape {
    var from: Point
    var to: Point

    func render(into context: CGContext) { /* ... */}
}

struct Rectangle: Shape {
    var origin: Point
    var width: Double
    var height: Double

    func render(into context: CGContext) { /* ... */}
}

// `Circle` type omitted.

```

While functionally equivalent, it's interesting to consider how these two approaches, using either enums or protocols, are organized and how they can be extended with new functionality. The enum-based implementation is grouped by methods: the CGContext-based rendering code for all types of shapes resides within a single switch statement in the `render(into:)` method. The protocol-based implementation, on the other hand, is grouped by "cases": each concrete shape type implements its own `render(into:)` method, which contains its specific rendering code.

This has important consequences in terms of extensibility: with the enum variant, we can easily add new render methods — e.g. to render into an SVG file — in an extension on `Shape` later on, even in a different module. However, we can't add new kinds of shapes to the enum unless we control the source code containing the enum declaration. And even if we can change the enum definition, adding a new case is a source-breaking change for all methods that switch over the enum.

On the other hand, we can easily add new kinds of shapes with the protocol variant: we just create a new struct and conform it to the `Shape` protocol. However, short of modifying the original `Shape` protocol, we can't add new kinds of render methods, because we can't add new protocol requirements outside the protocol declaration. (We can add new methods to the protocol in an extension, but as we'll see in the [Protocols](#) chapter, extension methods are often not suitable for adding new functionality to a protocol because they're not dynamically dispatched.)

It turns out that enums and protocols have complementary strengths and weaknesses in this scenario. Each solution is extensible in one dimension and lacks flexibility in the other. These differences in extensibility between enums and protocols are less important if the declaration of the API and its usage happen in the same module. However, if you're writing library code, you should consider which dimension of extensibility is more important: adding new cases or adding new methods.

If you're interested in this specific problem of extensibility across module boundaries, check out the [Swift Talk episodes](#) we recorded with Brandon Kase about this topic. In these episodes, we explore a technique that allows us to get extensibility along both dimensions at the same time.

Modeling Recursive Data Structures with Enums

Enums are perfect for modeling recursive data structures, i.e. data structures that “contain” themselves. Think of a tree structure: a tree has multiple branches, where each branch is another tree, which again branches into multiple subtrees, and so on until you reach the leaves. Many common data formats are tree structures, e.g. HTML, XML, and JSON.

As an example of a recursive data structure, let's implement a small subset of XML. For a full-fledged implementation, see [Swim](#) or [swift-html](#). We'll create a `Node` enum, which will be either a text node, or an element, or a fragment (multiple nodes). The either-or relation is a strong hint that a sum type, i.e. an enum, is a good fit for defining a type for this data structure:

```
enum Node: Hashable {
    case text(String)
    indirect case element(
        name: String,
        attributes: [String: String] = [:],
        children: Node = .fragment([]))
    case fragment([Node])
}
```

Notice the `indirect` keyword, which is required to make this compile. `indirect` tells the compiler to represent the `element` case as a reference, and that makes the recursion work. Without the `element` case being a reference, the enum would have an infinite size. We'll talk more about `indirect` in the next section.

The definition above is all we need to construct a simple tree of nodes, similar to the HTML code <h1>Hello World</h1>:

```
let header: Node = .element(name: "h1", children: .fragment([
    .text("Hello "),
    .element(name: "em", children: .text("World"))
]))
```

We'll now add a few convenience conformances and a method to make constructing Nodes easier. We can first make our type conform to ExpressibleByArrayLiteral to make it easier to construct fragments:

```
extension Node: ExpressibleByArrayLiteral {
    init(arrayLiteral elements: Node...) {
        self = .fragment(elements)
    }
}
```

Likewise, we can conform to ExpressibleByStringLiteral to make it easier to create .text nodes:

```
extension Node: ExpressibleByStringLiteral {
    init(stringLiteral value: String) {
        self = .text(value)
    }
}
```

Finally, let's make it easier to wrap nodes in elements by adding an extension method. The wrapped method returns self, but wrapped inside an element node:

```
extension Node {
    func wrapped(in elementName: String, attributes: [String: String] = [:])
        -> Node {
        .element(name: elementName, attributes: attributes, children: self)
    }
}
```

Given these three extensions, we can now write the example from earlier in a much shorter way:

```
let contents: Node = [
    "Hello ",
    ("World" as Node).wrapped(in: "em")
]
let headerAlt = contents.wrapped(in: "h1")
```

To make this feel even more at home in Swift, we could use result builders, giving us a syntax that looks like SwiftUI.

We mentioned at the beginning of this chapter that enums can also have mutating methods. For example, we could write a mutating variant of wrapped that changes self in place:

```
extension Node {
    mutating func wrap(in elementName: String, attributes: [String: String] = [:]) {
        self = .element(name: elementName, attributes: attributes, children: self)
    }
}

var greeting: Node = "Hello"
greeting.wrap(in: "strong")
```

Just like with structs, mutating doesn't change the value itself: it only changes what value the variable is pointing to.

Enums in Swift can also be used to model more abstract data structures: for example, it's straightforward to build a linked list, a binary tree, or even persistent data structures. When doing this, a word of warning applies: it's notoriously difficult to outperform Swift's built-in data structures — such as arrays, dictionaries, or sets — for most use cases.

Indirect

To understand why we had to make our recursive Node enum indirect, recall that enums are value types. And value types can't contain themselves because allowing this would create an infinite recursion when calculating the type's size. The compiler must be able to determine a fixed and finite size for each type. Treating the recursive case as a heap-allocated reference fixes this problem because the reference adds a level of indirection; the compiler knows that the storage size for any reference is always 8 bytes (on a 64-bit system).

Note that we didn't have to write `indirect` before the `fragment` case, even though it also uses `Node` recursively. This is because the associated value is an array, which has a constant size (the actual memory for the array is stored in a buffer).

The `indirect` syntax is only available for enums. If it weren't available, or if we wanted to model a recursive struct, we could replicate the same behavior by boxing the recursive value in a class, thus creating the indirection manually. For our `Node` enum, if we don't want to allow top-level fragments, we could've also provided an alternative definition. This doesn't need to be marked as `indirect`, because it relies on the array's indirection:

```
enum NodeAlt {  
    case text(String)  
    case element(name: String, attributes: [String: String], children: [Node])  
}
```

We can also add `indirect` to the enum declaration itself, i.e. `indirect enum Node { ... }`. This is a shorter syntax for enabling indirection for all cases that have an associated value (`indirect` only ever applies to the associated values and never to the tag bits the enum uses to distinguish its cases). If an `indirect` case has multiple associated values, the reference is put around the combined associated values.

The size of an enum value is the size of the largest case, plus the size required to store the tag (which of the cases the value represents). For example, the following enum's size is 17, which is the size of the largest case (16), plus a byte to store the tag (1):

```
enum TwoInts {  
    case nothing  
    case int(Int, Int)  
}  
  
MemoryLayout<TwoInts>.size // 17
```

If we mark the `int` case as `indirect`, the size changes to 8, not 9. Even though the size of the reference is 8 bytes, the reference has some unused bits to spare where the tag bits can be stored inline.

Sometimes, when you have an enum with a large case, you might want to mark that specific case as `indirect` to reduce the enum's size. Of course, this comes at the price of indirection.

In theory, the compiler could infer whether or not an enum should be marked as indirect. In some other languages (e.g. Haskell), this is done for us. The Swift designers chose not to do this, because they want to give us explicit control, for example by marking a large case as indirect. In addition, the compiler cannot reliably infer this: for a generic enum, whether or not it should be indirect can be dependent on the generic parameters.

Raw Values

It's sometimes desirable to associate each case of an enum with a number or some other value. Enums in C or Objective-C work like this by default — they're really just integers under the hood. Swift enums aren't interchangeable with arbitrary integers, but we can optionally declare a one-to-one mapping between an enum's cases and so-called *raw values*. This can be useful for interoperating with C APIs or for encoding an enum's value in a data format such as JSON (the [Codable](#) system, which we'll discuss in the [Encoding and Decoding](#) chapter, uses the raw value for synthesizing a [Codable](#) conformance for enums with raw values).

Giving an enum raw values involves adding a raw value type, separated from the type name by a colon, to the type declaration. We then assign a raw value to each case using assignment syntax. Here's an example of an enum for HTTP status codes with a raw value type of `Int`:

```
enum HTTPStatus: Int {  
    case ok = 200  
    case created = 201  
    // ...  
    case movedPermanently = 301  
    // ...  
    case notFound = 404  
    // ...  
}
```

Each case must have a unique raw value. If we don't provide values for one or more cases, the compiler will try to choose sensible defaults. In this example, we could have omitted the explicit raw value assignment for the `created` case; the compiler would've picked the same value, 201, by incrementing the previous case's raw value.

The RawRepresentable Protocol

A raw-representable type gains two new APIs: a `rawValue` property, and a failable initializer (`init?(rawValue:)`). These are declared in the `RawRepresentable` protocol (the compiler automatically implements this protocol for enums with raw values):

```
// A type that can be converted to and from an associated raw value.  
protocol RawRepresentable {  
    // The type of the raw values, such as Int or String.  
    associatedtype RawValue  
  
    init?(rawValue: RawValue)  
    var rawValue: RawValue { get }  
}
```

The initializer is failable because there may not be a valid value of the conforming type for every inhabitant of the `RawValue` type. For example, only a few dozen integers are valid HTTP status codes; for all other inputs, `HTTPStatus.init?(rawValue:)` must return `nil`:

```
HTTPStatus(rawValue: 404) // Optional(HTTPStatus.notFound)  
HTTPStatus(rawValue: 1000) // nil  
HTTPStatus.created.rawValue // 201
```

Manual RawRepresentable Conformance

The above syntax for assigning raw values to an enum only works for a limited set of types; the raw value type can be `String`, `Character`, or any integer or floating-point type. This covers a lot of use cases, but it doesn't mean these types are the only possible raw value types. Because the above syntax is just syntactic sugar for a `RawRepresentable` conformance, you always have the option of implementing the conformance manually if you need more flexibility.

The following example defines an enum that represents points in a logical coordinate system where `x` and `y` coordinates can assume values between `-1` (left/bottom) and `1` (right/top). This coordinate system is somewhat similar to the `anchorPoint` property of `CALayer` in Apple's Core Animation framework. We use a pair of integers as our raw value type, and since the convenience syntax doesn't support tuples, we implement `RawRepresentable` manually:

```

enum AnchorPoint {
    case center
    case topLeft
    case topRight
    case bottomLeft
    case bottomRight
}

extension AnchorPoint: RawRepresentable {
    typealias RawValue = (x: Int, y: Int)

    var rawValue: (x: Int, y: Int) {
        switch self {
            case .center: return (0, 0)
            case .topLeft: return (-1, 1)
            case .topRight: return (1, 1)
            case .bottomLeft: return (-1, -1)
            case .bottomRight: return (1, -1)
        }
    }

    init?(rawValue: (x: Int, y: Int)) {
        switch rawValue {
            case (0, 0): self = .center
            case (-1, 1): self = .topLeft
            case (1, 1): self = .topRight
            case (-1, -1): self = .bottomLeft
            case (1, -1): self = .bottomRight
            default: return nil
        }
    }
}

```

This is a little more code to write, but it isn't difficult. And it's exactly the code the compiler generates for us in its automatic RawRepresentable synthesis. It's no surprise then that the behavior for users of the enum is the same in both cases:

```

AnchorPoint.topLeft.rawValue // (x: -1, y: 1)
AnchorPoint(rawValue: (x: 0, y: 0)) // Optional(AnchorPoint.center)
AnchorPoint(rawValue: (x: 2, y: 1)) // nil

```

One thing to look out for when implementing `RawRepresentable` manually is the assignment of duplicate raw values. The automatic synthesis requires raw values to be unique — duplicates cause a compile error. But in a manual implementation, the compiler won't stop you from returning the same raw value for multiple enum cases. There may be a good reason to use duplicate raw values (e.g. when your enum uses multiple cases as synonyms for each other, perhaps for backward compatibility), but it should be the exception. Switching over an enum always matches against the enum's cases and never the raw values. In other words, you can't match one case with another case, even if they have the same raw value.

RawRepresentable for Structs and Classes

By the way, `RawRepresentable` isn't limited to enums; you can also conform a struct or class. `RawRepresentable` conformance is often a good choice for simple wrapper types that are introduced to preserve type safety. For instance, a program may use strings to represent user IDs internally. Instead of using `String` directly, it's a good idea to define a new `UserID` type to prevent accidental mixups with other string variables. It should still be possible to initialize a `UserID` with a string and to extract its string value; `RawRepresentable` is a good match for these requirements:

```
struct UserID: RawRepresentable {  
    var rawValue: String  
}
```

Here, the `rawValue` property satisfies one of the two protocol requirements, but where did the implementation for the second requirement, the initializer, go? It's provided by Swift's automatic memberwise initializer for structs. The compiler is smart enough to accept the (unavailable) `init(rawValue:)` as an implementation for the failable initializer the protocol requires. This has the nice side effect that we don't have to deal with optionals when creating a `UserID` from a string. If we wanted to perform validation of the input string (perhaps not all strings are valid user IDs), we'd have to provide our own implementation for `init?(rawValue:)`.

Internal Representation of Raw Values

Aside from the added `RawRepresentable` APIs and the different rules for automatic `Codable` synthesis, enums with raw values are really no different from all other enums. In particular, enums with raw values maintain their full type identity. Unlike in C, where you can assign arbitrary integer values to variables of an enum type, a Swift enum with

Int raw values doesn't "become" an integer. The only possible values an enum instance can have are the enum's cases, and the only way to get to the raw values is through the `rawValue` and `init?(rawValue:)` APIs.

Having raw values doesn't change an enum's in-memory representation, either. We can verify this by defining an enum with String raw values and looking at the type's size:

```
enum MenuItem: String {  
    case undo = "Undo"  
    case cut = "Cut"  
    case copy = "Copy"  
    case paste = "Paste"  
}  
  
MemoryLayout<MenuItem>.size // 1
```

The `MenuItem` type is only one byte. This tells us a `MenuItem` instance doesn't store the raw value internally — if it did, it'd have to be at least 16 bytes (the size of `String` on 64-bit platforms). The compiler-generated implementation for `rawValue` works like a computed property, similar to the implementation for `AnchorPoint` shown above.

Enumerating Enum Cases

In [Sum Types and Product Types](#) above, we talked about a type's inhabitants: the set of all possible values an instance of a type can have. It's often useful to operate on these values as a collection, e.g. to iterate over them or count them. The `Caselterable` protocol models this functionality by adding a static property (i.e. a property that's invoked on the type, not on an instance) named `allCases`:

```
/// A type that provides a collection of all of its values.  
protocol Caselterable {  
    associatedtype AllCases: Collection  
    where AllCases.Element == Self  
  
    static var allCases: AllCases { get }  
}
```

For enums without associated values, the compiler can automatically generate a `Caselterable` implementation; all we have to do is declare the conformance. Let's do this for our `MenuItem` type from the previous section:

```
enum MenuItem: String, Caselterable {
    case undo = "Undo"
    case cut = "Cut"
    case copy = "Copy"
    case paste = "Paste"
}
```

Because the `allCases` property is a `Collection`, it has all the usual properties and capabilities you know from arrays and other collections. In the following example, we use `allCases` to get a count of all menu items and to transform them into strings suitable for displaying in a user interface (for the sake of simplicity, we're using the raw values directly as menu item titles; a real app would use the raw values as keys into a lookup table where the localized titles are stored):

```
MenuItem.allCases
// [MenuItem.undo, MenuItem.cut, MenuItem.copy, MenuItem.paste]
MenuItem.allCases.count // 4
MenuItem.allCases.map { $0.rawValue } // ["Undo", "Cut", "Copy", "Paste"]
```

Similar to other compiler-synthesized protocol implementations, such as `Equatable` and `Hashable`, the biggest benefit of the autogenerated `Caselterable` conformance isn't the difficulty of the code itself (it's trivial to write a manual implementation), but the fact that the compiler-generated code will always be up to date — manual conformances have to be updated every time cases are added or removed, which is very easy to forget to do.

The `Caselterable` protocol doesn't prescribe a specific order of the values in the `allCases` collection, but the documentation for `Caselterable` guarantees that the synthesized conformance provides the cases in their declaration order.

Manual Caselterable Conformance

`Caselterable` is particularly useful for plain enums without associated values, and these are the only types covered by automatic compiler synthesis. This makes sense, because adding associated values to an enum makes the number of inhabitants the enum can have potentially infinite. However, as long as we can come up with a way to produce a collection of all inhabitants, we can always implement the conformance manually. In fact, we aren't even limited to enums. While the names `Caselterable` and `allCases` imply that this feature is intended mainly for enums (no other types have cases), the compiler will gladly accept a struct or class conforming to the protocol.

One of the simplest types to write a manual Caseltable implementation for is Bool:

```
extension Bool: Caseltable {  
    public static var allCases: [Bool] {  
        return [false, true]  
    }  
}  
  
Bool.allCases // [false, true]
```

Some integer types are also a good match. Note that the return type of allCases doesn't have to be an array — it can be any Collection. It'd be wasteful to generate an array of every possible integer when a range can represent the same collection with much less memory:

```
extension UInt8: Caseltable {  
    public static var allCases: ClosedRange<UInt8> {  
        return .min ... .max  
    }  
}  
  
UInt8.allCases.count // 256  
UInt8.allCases.prefix(3) + UInt8.allCases.suffix(3) // [0, 1, 2, 253, 254, 255]
```

By the same logic, if you want to write a Caseltable implementation for a type with a lot of inhabitants, or a type where the values are expensive to generate, consider returning a lazy collection so as not to perform unnecessary work upfront. We'll discuss lazy collections in the [Collection Protocols](#) chapter.

Note that both of these examples ignore the general rule not to conform types you don't own to protocols you don't own. Before you break this rule in production code, consider the tradeoffs associated with doing so. Refer to the [Protocols](#) chapter for more on this.

Frozen and Non-Frozen Enums

We stressed repeatedly throughout this chapter that one of the best qualities of enums is the ability to switch exhaustively over them. The compiler can only perform its exhaustiveness checks if it knows at compile time about all possible cases an enum can have. This is trivially true for enum declarations that are in the same module as the code that switches over them. It's also true if the enum declaration is in another library but

this library is compiled together with the client code — every time a case is added or removed, the enum declaration and the client code are recompiled, allowing the compiler to recheck all switch statements.

However, there are situations where we want to switch over an enum from a module that's linked into our program in binary form. The standard library is the most prominent example of this: even though the source code for the standard library is freely available, we usually use the binary that ships with our Swift distribution or operating system. The same is true for the other libraries that ship with Swift, including Foundation and Dispatch. Lastly, Apple and other companies want to ship Swift libraries in binary form.

To take an example of a standard library type, suppose we want to switch over a DecodingError instance in our code. DecodingError is an enum that, as of Swift 5.5, has four cases to indicate different error conditions:

```
let error: DecodingError = ...
// Exhaustive at compile time, but possibly not at runtime.
switch error {
    case .typeMismatch: ...
    case .valueNotFound: ...
    case .keyNotFound: ...
    case .dataCorrupted: ...
}
```

It's not unlikely that future Swift versions will add additional cases as the Codable system expands. But if we build an app that contains this code and ship it to customers, some of these customers may eventually run the executable on a newer OS with a newer Swift version that includes another DecodingError case. In this situation, our program would crash because it encountered a condition it couldn't handle.

An enum that may gain new cases in the future is called *non-frozen*. To make programs resilient against changes to non-frozen enums, code that switches over a non-frozen enum from another module must always include a default clause to handle future cases. In Swift 5.5, the compiler only emits a warning (not an error) if you omit the default case, but this is only a temporary situation to make migration of existing code easier. It'll become an error in a future release.

If you accept the compiler's fix-it for the warning, you'll notice that it adds an @unknown attribute to the default case:

```
switch error {  
    ...  
    case .dataCorrupted: ...  
    @unknown default:  
        // Handle unknown cases.  
    ...  
}
```

@unknown default behaves like a normal default clause at runtime, but it's also a signal to the compiler that the default case is only meant to handle enum cases that are unknown at compile time. If the default case matches a case that's known at compile time, we'll still get a warning. This means we can still benefit from exhaustiveness checking when we recompile our program in the future against a newer library interface. If a case got added to the library API since the last update, we'll get warnings to update all our switch statements to explicitly handle the new case. @unknown default gives you the best of both worlds: compile-time exhaustiveness checking and runtime safety.

The distinction between frozen and non-frozen enums is only enabled for modules that are compiled in *library evolution mode*, which is off by default and activated by the `-enable-library-evolution` compiler flag. Modules that have library evolution enabled are also called *resilient libraries*; they're designed to maintain a stable ABI while still allowing the library authors to make certain API changes. Adding an enum case is one example of such a change. The standard library is a resilient library, as are all frameworks in Apple's SDKs. Enums in resilient libraries — i.e. libraries that are designed between releases — are non-frozen by default. There's also an attribute, `@frozen`, for marking a particular enum declaration as frozen. By using this attribute, the developers of the library promise to never add another case to that enum — doing so would break binary compatibility.

Examples of frozen enums in the standard library include `Optional` and `Result`; if they weren't frozen, switching over them would always require a default clause, which would be a major annoyance.

Enums in non-resilient libraries (which covers all modules you compile and link directly into your program, such as Swift Package Manager dependencies that are open source) are always considered frozen and thus don't require an `@unknown default` case. This is fine because the binary interfaces of these modules will never run out of sync.

Tips and Tricks

We'll close the chapter with some tips and tricks.

Try to avoid nested switch statements. You can use a tuple to switch over multiple values at once. For example, say you want to set a variable based on the value of two Booleans. Switching over the Booleans one after another requires repeating the inner switch, and this becomes ugly quickly:

```
let isImportant: Bool = ...
let isUrgent: Bool = ...
```

```
let priority: Int
switch isImportant {
case true:
    switch isUrgent {
        case true: priority = 3
        case false: priority = 2
    }
case false:
    switch isUrgent {
        case true: priority = 1
        case false: priority = 0
    }
}
```

Putting the two Booleans into a tuple and switching over that is shorter and more readable:

```
let priority2: Int
switch (isImportant, isUrgent) {
case (true, true): priority2 = 3
case (true, false): priority2 = 2
case (false, true): priority2 = 1
case (false, false): priority2 = 0
}
```

Take advantage of definite initialization checks. Take another look at the previous code sample. This pattern of declaring, but not initializing, a let constant before a switch, and then initializing it in every case of the switch, takes advantage of the compiler's definite initialization checks. The compiler verifies that a variable has been

fully initialized before it's first used — it'll emit an error if we forget to initialize in one or more code paths. This style is much safer than the naive alternative of making priority a var and assigning to it twice (once at the declaration site and then again inside the switch).

Like if, switch is a statement, not an expression — though we often wish it was the latter. Swift has no convenient syntax for setting a variable as a result of switching over an enum. Declaring a constant before the switch statement and assigning to it in each case is the best we can do.

Avoid naming enum cases none or some. These are attractive names for an enum case; we recommend you avoid them, though, due to the potential clash with Optional's case names in pattern matching contexts. So, this is a problematic enum definition:

```
enum Selection {  
    case none  
    case some  
    case all  
}
```

Suppose we have a variable of type Selection? (i.e. an optional) and want to match it against a pattern:

```
var optionalSelection: Selection? = ...
```

```
if case .some = optionalSelection {  
    // Some items selected? Or?  
}
```

Does this only match Selection.some, or does it match Optional.some, i.e. any non-nil value? The answer is the latter, but this is easy to get wrong, especially considering that Swift likes to promote non-optional values to optionals implicitly. (The compiler does issue a warning for if case .none = optionalSelection to point out the ambiguity, but it doesn't for if case .some =....)

Use backticks for case names that are reserved words. If you use certain keywords as case names (e.g. default), the type checker will complain because it can't parse the code. You can wrap the word in backticks to use it anyway:

```
enum Strategy {  
    case custom
```

```
    case `default` // requires backticks.  
}
```

What's nice about this is that the backticks aren't required in places where the type checker can disambiguate what you mean. This is perfectly valid:

```
let strategy = Strategy.default
```

Enum cases can be used like factory methods. If an enum case has an associated value, the case name alone forms a function with the signature (AssocValue) → Enum. Take this enum to represent a color in one of two color spaces — RGB or grayscale:

```
enum OpaqueColor {  
    case rgb(red: Float, green: Float, blue: Float)  
    case gray(intensity: Float)  
}
```

OpaqueColor.rgb is a function that takes three Floats and produces an OpaqueColor:

```
OpaqueColor.rgb // (Float, Float, Float) -> OpaqueColor
```

We can also pass these functions to higher-order functions such as map. Here, we create a gradient of grayscale colors from black to white by passing the enum case directly to map as a factory method:

```
let gradient = stride(from: 0.0, through: 1.0, by: 0.25).map(OpaqueColor.gray)  
/*  
[OpaqueColor.gray(intensity: 0.0), OpaqueColor.gray(intensity: 0.25),  
OpaqueColor.gray(intensity: 0.5), OpaqueColor.gray(intensity: 0.75),  
OpaqueColor.gray(intensity: 1.0)]  
*/
```

Enum cases can even satisfy protocol requirements that have the same shape. Here, the protocol's static method requirement maps directly to the enum case of the same name, so the conformance comes for free:

```
protocol ColorProtocol {  
    static func rgb(red: Float, green: Float, blue: Float) -> Self  
}
```

```
// No code required.  
extension OpaqueColor: ColorProtocol {}
```

Don't use associated values to fake stored properties. Use a struct instead. Enums can't have stored properties. This may sound like a significant limitation, but it isn't. If you think about it, adding a stored property of type T is really no different than adding an associated value of the same type to every case. For instance, let's add an alpha channel to our OpaqueColor type from above by giving each case one more associated value:

```
enum AlphaColor {  
    case rgba(red: Float, green: Float, blue: Float, alpha: Float)  
    case gray(intensity: Float, alpha: Float)  
}
```

This works, but extracting the alpha amount from an AlphaColor instance isn't very convenient now — we'd have to switch over the instance and extract the value from each case, even though we know every AlphaColor has an alpha component. We could wrap this logic in a computed property, but a better solution may be to avoid the problem in the first place — let's wrap the original OpaqueColor enum in a struct and make alpha a stored property of the struct:

```
struct Color {  
    var color: OpaqueColor  
    var alpha: Float  
}
```

This is a general pattern: when you see an enum where each case has the same piece of data in its payload, consider wrapping the enum in a struct and pulling the common property out. This changes the shape of the resulting type, but it doesn't change its fundamental nature. It's the same as factoring out a common factor in a math equation: $a \times b + a \times c = a \times (b + c)$. This correspondence with algebra is why the umbrella term for sum and product types is “algebraic data type.”

Don't overdo it with associated value components. We used associated values with multiple tuple-style components — such as OpaqueColor.rgb(red:green:blue:) — quite heavily in this chapter. This is convenient for short examples, but in production code, writing a custom struct for each case is often a better choice. Compare the two versions of the Shape type we used above in the [Pattern Matching](#) section. First, here's the original tuple style:

```
enum Shape {
    case line(from: Point, to: Point)
    case rectangle(origin: Point, width: Double, height: Double)
    case circle(center: Point, radius: Double)
}
```

And here's the alternative with one custom struct per case:

```
struct Line {
    var from: Point
    var to: Point
}

struct Rectangle {
    var origin: Point
    var width: Double
    var height: Double
}

struct Circle {
    var center: Point
    var radius: Double
}

enum Shape2 {
    case line(Line)
    case rectangle(Rectangle)
    case circle(Circle)
}
```

The latter example requires writing a little more code initially, but it cleans up the enum declaration, as well as patterns in switch statements. In addition, the structs have an identity of their own; we can extend them and conform them to protocols.

Use caseless enums as namespaces. Aside from the implicit namespaces formed by modules, Swift doesn't have namespaces built in. We can use enums as "fake" namespaces though. Since type definitions can be nested, outer types act as namespaces for all declarations they contain. As we saw in the [Optionals](#) chapter, enums that have no cases — such as `Never` — can't be instantiated. This makes an empty caseless enum the best option for defining a custom namespace. The standard library does this too — for example, with the Unicode "namespace":

```
/// A namespace for Unicode utilities.  
public enum Unicode {  
    public struct Scalar {  
        internal var _value: UInt32  
        // ...  
    }  
    // ...  
}
```

Unfortunately, caseless enums aren't a perfect workaround for the lack of proper namespaces: protocols can't be nested inside other declarations, which is why a related standard library protocol is named `UnicodeCodec` rather than `Unicode.Codec`.

Recap

Enums are sum types. When defining custom types, enums are an important tool for avoiding the combinatorial explosion of unwanted states characteristic of a design based purely on product types. Thinking carefully about a type's inhabitants helps us make better design decisions. An enum, or a combination of nested enums and structs, is often the best choice if you need a type that's precisely tailored to the problem you're trying to solve — for example, to model your program's state.

Enums lend themselves to different design patterns than the more familiar record types do. Your goal should be to make illegal program states impossible to represent in your types. This reduces the set of states your code must be prepared to handle, and it enables the compiler to guide you when writing new code. Whenever possible, take advantage of the compiler's exhaustiveness checks.

Strings

8

All modern programming languages have support for Unicode strings, but that often only means that the native string type can store Unicode data — it's not a promise that simple operations, like getting the length of a string, will return “sensible” results. In fact, most languages, and in turn, most string manipulation code written in those languages, exhibit a certain level of denial about Unicode’s inherent complexity. This can lead to some unpleasant bugs.

Swift’s string implementation goes to heroic efforts to be as Unicode-correct as possible. A String in Swift is a collection of Character values, where a Character is what a human reader of a text would perceive as a single character, regardless of how many Unicode scalars it’s composed of. As a result, all standard Collection operations — like count or prefix(5) — work on the level of user-perceived characters.

This is great for correctness, but it comes at a price, mostly in terms of unfamiliarity; if you’re used to manipulating strings with integer indices in other languages, Swift’s design will seem unwieldy at first, leaving you wondering: Why can’t I write str[999] to access a string’s one-thousandth character? Why doesn’t str[idx+1] get the next character? Why can’t I loop over a range of Character values such as "a"..."z"? It also has performance implications: String does *not* support random access, i.e. jumping to an arbitrary character isn’t an $O(1)$ operation. It can’t be — when characters have variable width, the string doesn’t know where the n^{th} character is stored without looking at all the characters that come before it.

In this chapter, we’ll discuss the string architecture in detail, along with some techniques for getting the most out of Swift strings in terms of functionality and performance. But we’ll start with an overview of the required Unicode terminology.

Unicode

Things used to be so simple. ASCII strings were a sequence of integers between 0 and 127. If you stored them in an 8-bit byte, you even had a bit to spare! Since every character was of a fixed size, ASCII strings could be random access.

But ASCII wasn’t enough if you were writing in anything other than English or for a non-U.S. audience; other countries and languages needed other characters (even English-speaking Britain needed a £ sign). Most of them needed more characters than would fit into seven bits. ISO 8859 takes the extra bit and defines 16 different encodings above the ASCII range, such as Part 1 (ISO 8859-1, aka Latin-1), covering several Western European languages; and Part 5, covering languages that use the Cyrillic alphabet.

This is still limiting, though: if you want to use ISO 8859 to write in Turkish about Ancient Greek, you're out of luck, since you'd need to pick either Part 7 (Latin/Greek) or Part 9 (Turkish). And eight bits is still not enough to encode many languages. For example, Part 6 (Latin/Arabic) doesn't include the characters needed to write Arabic-script languages such as Urdu or Persian. Meanwhile, Vietnamese — which is based on the Latin alphabet but with a large number of diacritic combinations — only fits into eight bits by replacing a handful of ASCII characters from the lower half. And this isn't even an option for other East Asian languages.

When you run out of room with a fixed-width encoding, you have a choice: either increase the size, or switch to variable-width encoding. Initially, Unicode was defined as a 2-byte fixed-width format, now called UCS-2. This was before reality set in and it was accepted that even two bytes (i.e. ~65,000 code points) wouldn't be sufficient, while four would be horribly inefficient for most purposes. So today, Unicode is a variable-width format, and it's variable in two different senses:

- A single character (also known as an *extended grapheme cluster*) consists of one or more Unicode *scalars*.
- A scalar is encoded by one or more *code units*.

To understand why, we need to clarify what these terms mean.

The basic building block of Unicode is the *code point*: an integer value in the Unicode code space, which ranges from 0 to 0x10FFFF (in decimal notation: 1,114,111). Every character or other unit of script that's part of Unicode is assigned a unique code point. In Unicode 14 (published in September 2021), only about 145,000 of the 1.1 million available code points are currently in use, so there's a lot of room for more emoji. Code points are commonly written in hex notation with a "U+" prefix. For example, the euro sign is at code point U+20AC (or 8364 in decimal).

Unicode *scalars* are almost, but not quite, the same as code points. They're all the code points *except* the 2,048 *surrogate code points* in the range 0xD800 to 0xDFFF (which are used by the UTF-16 encoding to represent code points greater than 65,535). Scalars are represented in Swift string literals as "\u{xxxx}", where xxxx represents hex digits. So the euro sign can be written in Swift as either "\u{20AC}" or "\u{U+20AC}". The corresponding Swift type is `Unicode.Scalar`, which is a wrapper around a `UInt32` value.

The same Unicode data (i.e. a sequence of scalars) can be encoded with different encodings, with UTF-8 and UTF-16 being the most common ones. The smallest entity in an encoding is called a *code unit*. The UTF-8 encoding has 8-bit-wide code units, and UTF-16 has 16-bit-wide code units. UTF-8 has the added benefit of being backward

compatible with 8-bit ASCII — a feature that's helped it overtake ASCII as the most popular encoding on the web and in file formats. Code units are different from code points or scalars because a single scalar is often encoded with multiple code units. Since there are more than a million potential code points, UTF-8 takes one to four code units (one to four bytes) to encode a single scalar, whereas UTF-16 takes either one or two code units (two or four bytes). Swift represents UTF-8 and UTF-16 code units as `UInt8` and `UInt16` values, respectively (aliased as `Unicode.UTF8.CodeUnit` and `Unicode.UTF16.CodeUnit`).

To represent each scalar by a single code unit, you'd need a 21-bit encoding scheme, which usually gets rounded up to 32-bit and is called UTF-32. This is what `Unicode.Scalar` does in Swift. But even that wouldn't get you a fixed-width encoding: Unicode is still a variable-width format when it comes to "characters." What a user might consider "a single character" — as displayed on the screen — might require multiple scalars composed together. The Unicode term for such a user-perceived character is an *(extended) grapheme cluster*.

The rules for how scalars form grapheme clusters determine how text is segmented. For example, if you hit the backspace key on your keyboard, you expect your text editor to delete exactly one grapheme cluster, even if that "character" is composed of multiple Unicode scalars, each of which may use a varying number of code units in the text's representation in memory. Grapheme clusters are represented in Swift by the `Character` type, which can encode an arbitrary number of scalars, as long as they form a single user-perceived character.

The following diagram shows three different views of the string "AB • ". Depending on how you look at it, the string is made up of four `Characters`, five Unicode scalars, or twelve UTF-8 code units:

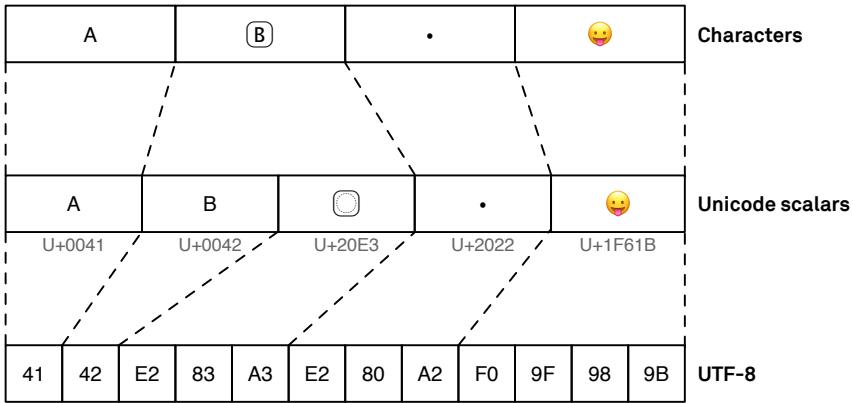


Figure 8.1: Different views of the same string

We'll see more examples, as well as how Swift deals with the arising complexity, in the next section.

Grapheme Clusters and Canonical Equivalence

Combining Marks

A quick way to see how String handles Unicode data is to look at the two different ways to write é. Unicode defines U+00E9, *Latin small letter e with acute*, as a single value. But you can also write it as the plain letter e, followed by U+0301, *combining acute accent*. In both cases, what's displayed is é, and users have the expectation that two strings displayed as “résumé” would not only be equal to each other but also have a “length” of six characters, no matter which technique was used to produce the é in either one. They'd be what the Unicode specification describes as *canonically equivalent*.

And in Swift, this is exactly the behavior you get:

```
let single = "Pok\u{00E9}mon" // Pok\u00e9mon
let double = "Poke\u{0301}mon" // Pok\u00e9mon
```

They both display identically:

```
(single, double) // ("Pokémon", "Pokémon")
```

And both have the same character count:

```
single.count // 7  
double.count // 7
```

Consequently, they also compare equal:

```
single == double // true
```

Only if you drop down to a view of the underlying representation can you see that they're different:

```
single.unicodeScalars.count // 7  
double.unicodeScalars.count // 8
```

Compare this with `NSString` in Foundation: the two strings aren't equal, and the `length` property — which many Objective-C programmers probably use to count the number of characters to be displayed on the screen — gives different results:

```
let nssingle = single as NSString  
nssingle.length // 7  
let nsdouble = double as NSString  
nsdouble.length // 8  
nssingle == nsdouble // false
```

Here, `==` is defined as the version for comparing two `NSObject`s:

```
extension NSObject: Equatable {  
    static func ==(lhs: NSObject, rhs: NSObject) -> Bool {  
        return lhs.isEqual(rhs)  
    }  
}
```

In the case of `NSString`, `==` will do a literal comparison on the level of UTF-16 code units rather than one accounting for equivalent but differently composed characters. Most string APIs in other languages work this way too. If you really want to perform a canonical comparison of two `NSStrings`, you must use `NSString.compare(_:)`.

Of course, there's one big benefit to just comparing code units: it's faster! This is an effect that can still be achieved with Swift strings via the `utf8` view:

```
single.utf8.elementsEqual(double.utf8) // false
```

Why does Unicode support multiple representations of the same character at all? The existence of precomposed characters is what enables the opening range of Unicode code points to be compatible with Latin-1, which already had characters like é and ñ. While they might be a pain to deal with, it makes conversion between the two encodings quick and simple.

And ditching precomposed forms wouldn't have helped anyway, because composition doesn't just stop at pairs; you can compose more than one diacritic together. For example, Yoruba has the character ɸ, which could be written three different ways: by composing ó with a dot, or by composing ø with an acute, or by composing o with both an acute and a dot. And for that last one, the two diacritics can be in either order! So these are all equal:

```
let chars: [Character] = [
    "\u{1ECD}\u{300}", // ɸ
    "\u{F2}\u{323}", // ɸ
    "\u{6F}\u{323}\u{300}", // ɸ
    "\u{6F}\u{300}\u{323}" // ɸ
]
let allEqual = chars.dropFirst().allSatisfy { $0 == chars.first } // true
```

In fact, some diacritics can be added ad infinitum. A famous internet meme illustrates this nicely:

```
let zalgo = "soon"
```

```
zalgo.count // 4
zalgo.utf8.count // 68
```

In the above, `zalgo.count` (correctly) returns 4, while `zalgo.utf8.count` returns 68. And if your code doesn't work correctly with internet memes, then what good is it, really?

Unicode's grapheme-breaking rules even affect you when all strings you deal with are pure ASCII: CR+LF, the character pair of carriage return and line feed that's commonly used as a line break on Windows, is a single grapheme:

```
// CR+LF is a single Character.  
let crlf = "\r\n"  
crlf.count // 1
```

Emoji

Strings containing emoji can also be surprising in various other programming languages. Many emoji are assigned Unicode scalars that don't fit in a single UTF-16 code unit. Languages that represent strings as collections of UTF-16 code units, such as Java or C#, would say that the string "😂" is two "characters" long. Swift handles this case correctly:

```
let oneEmoji = "😂" // U+1F602  
oneEmoji.count // 1
```

Notice that the important thing is how the string is exposed to the program, and *not* how it's stored in memory. Swift uses UTF-8 as its internal encoding, but that's an implementation detail. The public API is based on grapheme clusters.

Other emoji are composed of multiple scalars. An emoji flag is a combination of two regional indicator symbols that correspond to an ISO country code. Swift treats the flag correctly as one Character:

```
let flags = "🇧🇷 🇺🇸"  
flags.count // 2
```

To inspect the Unicode scalars a string is composed of, use the `unicodeScalars` view. Here, we format the scalar values as hex numbers in the common format for code points:

```
flags.unicodeScalars.map {  
    "U+\($0.value, radix: 16, uppercase: true)"  
}  
// ["U+1F1E7", "U+1F1F7", "U+1F1F3", "U+1F1FF"]
```

Skin tones combine a base character such as 🤵 with one of five skin tone modifiers (e.g. 🤤, or the *type-4 skin tone modifier*) to yield the final emoji (以人为例). Again, Swift handles this correctly:

```
let skinTone = "以人为例" // 🤵 + 🤤 + 🤤
```

Emoji depicting families and couples, such as 🤰 and 🤵, present another challenge to the Unicode standards body. Due to the countless possible combinations of gender and the number of people in a group, providing a separate code point for each variation is problematic. Combine this with a distinct skin tone for each person and it becomes impossible. Unicode solves this by specifying that these emoji are actually sequences of multiple emoji combined with the invisible *zero-width joiner* (ZWJ) character (U+200D). So the family 🤰 is really *man* 🤵 + ZWJ + *woman* 🤵 + ZWJ + *girl* 🤵 + ZWJ + *boy* 🤵. The ZWJ serves as an indicator to the operating system that it should use a single glyph if available.

You can verify that this is really what's going on:

```
let family1 = "以人为例"
let family2 = "以人为例\ufe0f以人为例\ufe0f以人为例\ufe0f以人为例\ufe0f"
family1 == family2 // true
```

And once again, Swift is smart enough to treat such a sequence as a single Character:

```
family1.count // 1
family2.count // 1
```

Emoji for professions are ZWJ sequences too. For example, the *female firefighter* 🚒 is composed of *woman* 🤵 + ZWJ + *fire engine* 🚒, and the *male health worker* 🚒 is a sequence of *man* 🤵 + ZWJ + *staff of Aesculapius* ✌.

Rendering these sequences into a single glyph is the task of the operating system. On Apple platforms in 2022, the OS includes glyphs for the subset of sequences the Unicode standard lists as “recommended for general interchange” (RGI), i.e. the ones “most likely to be widely supported across multiple platforms.” When no glyph is available for a syntactically valid sequence, the text-rendering system falls back to rendering each component as a separate glyph. Notice that this can cause a mismatch “in the other direction” between user-perceived characters and what Swift sees as a grapheme cluster;

all examples up until now were concerned with programming languages *overcounting* characters, but here we see the reverse. As an example, family sequences containing skin tones are currently not part of the RGI subset. But even if the operating system renders such a sequence as multiple glyphs, Swift still counts it as a single Character because the Unicode text segmentation rules aren't concerned with rendering:

```
// Family with skin tones is rendered as multiple glyphs
// on most platforms in 2022.

let family3 = "👨\u{200D}👩\u{200D}👧\u{200D}👲\u{200D}👳\u{200D}"
// But Swift still counts it as a single Character.
family3.count // 1
```

No matter how carefully a string API is designed, text is so complicated that it may never catch all edge cases.

Swift uses the grapheme-breaking algorithm of the operating system's [ICU](#) library. As a result, your programs will automatically adopt new Unicode rules as users update their OSes. This means you can't rely on users seeing the same behavior you see during development. For example, when you deploy server-side Swift code on Linux, the code might behave differently because the Linux distribution might ship with a different ICU version than your development machine does.

In the examples we discussed in this section, we treated the length of a string as a proxy for all sorts of things that can go wrong when a language doesn't take the full complexity of Unicode into account. Just think of the gibberish a simple task such as reversing a string containing composed character sequences can produce in a programming language that doesn't process strings by grapheme clusters. This isn't a new problem, but the emoji explosion has made it much more likely that bugs caused by sloppy text handling will come to the surface, even if you have a predominantly English-speaking user base. And the magnitude of errors has increased as well: whereas a decade ago, a botched accented character would cause an off-by-one error, messing up a modern emoji can easily cause results to be off by 10 or more "characters." For example, a four-person family emoji is 11 (UTF-16) or 25 (UTF-8) code units long:

```
family1.count // 1
family1.utf16.count // 11
family1.utf8.count // 25
```

It's not that other languages don't have Unicode-correct APIs at all — most do. For instance, `NSString` has the `enumerateSubstrings` method that can be used to walk through a string by grapheme clusters. But defaults matter, and Swift's priority is to do the correct thing by default. And if you ever need to drop down to a lower level of abstraction, `String` provides views that let you operate directly on Unicode scalars or code units. We'll say more about those below.

Strings and Collections

As we've seen, `String` is a collection of `Character` values. In Swift's first three years of existence, `String` went back and forth between conforming and not conforming to the `Collection` protocol. The argument for *not* adding the conformance was that programmers would expect all generic collection-processing algorithms to be completely safe and Unicode-correct, which wouldn't necessarily be true for all edge cases.

As a simple example, you might assume that if you concatenate two collections, the resulting collection's length would be the sum of the lengths of the two source collections. But this doesn't hold for strings if a suffix of the first string forms a grapheme cluster with a prefix of the second string:

```
let flagLetterJ = "🇯 " // 2 characters
let flagLetterP = "🇵 " // 2 characters
let flag = flagLetterJ + flagLetterP // 🇵🇯
flag.count // 1 character
flag.count == flagLetterJ.count + flagLetterP.count // false
```

To this end, `String` itself wasn't made a `Collection` in Swift 2 and 3; rather, a collection-of-characters view was moved to a property, `characters`, which put it on a footing similar to the other collection views: `unicodeScalars`, `utf8`, and `utf16`. Picking a specific view prompted you to acknowledge you were moving into a collection-processing mode and that you should consider the consequences of the algorithm you were about to run.

In practice, the gain in correctness for a few edge cases that are rarely relevant in real code (unless you're writing a text editor) turned out to not be worth the loss in usability and learnability caused by this change. So `String` was made a `Collection` again in Swift 4.

Bidirectional, Not Random Access

However, for reasons that should be clear from the examples we've seen so far in this chapter, String is *not* a random-access collection. How could it be, when knowing where the n^{th} character of a particular string is involves evaluating just how many Unicode scalars precede that character? For this reason, String conforms only to BidirectionalCollection. You can start at either end of the string, moving forward or backward, and the code will look at the composition of the adjacent characters and skip over the correct number of bytes. However, you need to iterate up and down one character at a time.

Keep the performance implications of this in mind when writing string-processing code. Algorithms that depend on random access to maintain their performance guarantees aren't a good match for Unicode strings. Consider this String extension for generating a list of a string's prefixes, which works by generating an integer range from zero to the string's length and then mapping over the range to create the prefix for each length:

```
extension String {  
    var allPrefixes1: [Substring] {  
        return (0...count).map(prefix)  
    }  
}  
  
let hello = "Hello"  
hello.allPrefixes1 // ["", "H", "He", "Hel", "Hell", "Hello"]
```

As simple as this code looks, it's very inefficient. It first walks over the string once to calculate the length, which is fine. But then, each of the $n + 1$ calls to prefix is another $O(n)$ operation, because prefix always starts at the beginning and has to work its way through the string to count the desired number of characters. Running a linear process inside another linear loop means this algorithm is accidentally $O(n^2)$ — as the length of the string increases, the time this algorithm takes increases quadratically.

If possible, an efficient string algorithm should walk over a string only once and then operate on string indices to denote the substrings it's interested in. Here's a better version of the same algorithm:

```
extension String {  
    var allPrefixes2: [Substring] {
```

```
        return [""] + indices.map { index in self[...index] }
    }
}

hello.allPrefixes2 // [ "", "H", "He", "Hel", "Hell", "Hello" ]
```

This code also has to iterate over the string once to generate the `indices` collection. But once that's done, the subscripting operation inside `map` is $O(1)$. This makes the whole algorithm $O(n)$.

Range-Replaceable, Not Mutable

`String` also conforms to `RangeReplaceableCollection`. Here's an example of how you'd replace part of a string by first identifying the appropriate range in terms of string indices and then calling `replaceSubrange`. The replacement string can have a different length or could even be empty (which would be equivalent to calling `removeSubrange`):

```
var greeting = "Hello, world!"
if let comma = greeting.firstIndex(of: ",") {
    greeting[..<comma] // Hello
    greeting.replaceSubrange(comma..., with: " again.")
}
greeting // Hello again.
```

As always, keep in mind that results may be surprising if parts of the replacement string form new grapheme clusters with adjacent characters in the original string.

One collection-like feature that strings do *not* provide is that of `MutableCollection`. This protocol adds one feature to a collection — that of the single-element subscript set — in addition to `get`. This isn't to say strings aren't mutable — as we just saw, they have several mutation methods. But what you can't do is replace a single character using the subscript operator. The reason comes back to variable-length characters. Most people can probably intuit that a single-element subscript update would happen in constant time, as it does for `Array`. But since a character in a string may be of variable width, updating a single character could take linear time in proportion to the length of the string: changing the width of a single element would require shuffling all the later elements up or down in memory. Moreover, indices that come after the replaced index would become invalid through the shuffling, which is equally unintuitive. For these reasons, you have to use `replaceSubrange`, even if the range you pass in is only a single character.

String Indices

Most programming languages use integers for subscripting strings, e.g. `str[5]` would return the sixth “character” of `str` (for whatever that language’s idea of a “character” is). Swift doesn’t allow this. Why? The answer should sound familiar to you by now: subscripting is supposed to take constant time (intuitively, as well as per the requirements of the Collection protocol), and looking up the n^{th} Character is impossible without looking at all the bytes that come before it.

`String.Index`, which is the index type used by `String` and its views, is an opaque value that essentially stores a byte offset into the string’s in-memory representation (usually UTF-8). It’s still an $O(n)$ operation if you want to compute the index for the n^{th} character and have to start at the beginning of the string, but once you have a valid index, subscripting the string with it will only take $O(1)$ time. And crucially, finding the next index after an existing index is also fast because you can start at the existing index’s byte offset — you don’t need to go back to the beginning again. This is why iterating over the characters in a string in order (forward or backward) is efficient.

String index manipulation is based on the same Collection APIs you’d use with any other collection. It’s easy to miss this equivalence since the collections we use the most — arrays — use integer indices, and we usually use simple arithmetic to manipulate those. The `index(after:)` method returns the index of the next character:

```
let s = "abcdef"
let second = s.index(after: s.startIndex)
s[second] // b
```

You can automate iterating over multiple characters in one go via the `index(_:offsetBy:)` method:

```
// Advance 4 more characters.
let sixth = s.index(second, offsetBy: 4)
s[sixth] // f
```

If there’s a risk of advancing past the end of the string, you can add a `limitedBy:` parameter. The method returns `nil` if it hits the limit before reaching the target index:

```
let safelidx = s.index(s.startIndex, offsetBy: 400, limitedBy: s.endIndex)
safelidx // nil
```

This is undoubtedly more code than simple integer indices would require, but again, that's the point. If Swift allowed integer subscripting of strings, the temptation to accidentally write horribly inefficient code (e.g. by using integer subscripting inside a loop) would be too big.

Nevertheless, to someone used to dealing with fixed-width characters, working with strings in Swift seems challenging at first — how will you navigate without integer indices? And indeed, some seemingly simple tasks, like extracting the first four characters of a string, can turn into monstrosities like this one:

```
s[..
```

But thankfully, being able to access the string via the Collection interface also means you have several helpful techniques at your disposal. Most of the methods that operate on `Array` also work on `String`. Using the `prefix` method, the same thing looks much clearer:

```
s.prefix(4) // abcd
```

(Note that both expressions return a `Substring`; you can convert it back into a `String` by wrapping it in a `String.init`. We'll talk more about substrings in the next section.)

As a slightly more complex example, extracting the month from a date string can be accomplished entirely without performing any subscripting operations on the string:

```
let date = "2019-09-01"  
date.split(separator: "-")[1] // 09  
date.dropFirst(5).prefix(2) // 09
```

For finding a specific character, you can use `firstIndex(of:)`:

```
var hello = "Hello!"  
if let idx = hello.firstIndex(of: "!") {  
    hello.insert(contentsOf: ", world", at: idx)  
}  
hello // Hello, world!
```

The `insert(contentsOf:at:)` method inserts another collection of the same element type (e.g. `Character` for strings) before a given index. This doesn't have to be another `String`; you could insert an array of characters into a string just as easily.

Having established that opaque indices are the way to work with strings, the Swift team recognizes there are valid use cases where you just want to do the “easy” thing. To that end, they [proposed](#) adding a subscripting syntax for integer offsets to all collection types, and not just strings. The core team ultimately returned the proposal for revision to incorporate feedback that came up during the review, but this revision never happened, perhaps because other things were more pressing.

String Parsing

Of course, there are also tasks that cannot be accomplished just by using the Collection APIs on a string: parsing a CSV file is a good example of this. We can’t naively split a line on comma characters, because commas can also appear within values that are wrapped in quotes. To solve tasks like this, we can iterate over the string, character by character, while keeping track of some state. Essentially, we’re writing a very simple parser:

```
func parse(csv: String) -> [[String]] {
    var result: [[String]] = [[]]
    var currentField = ""
    var inQuotes = false
    for c in csv {
        switch (c, inQuotes) {
            // ...
        }
    }
    return result
}
```

First, we create a `result` as an array of arrays of strings. Each line is represented by an array of strings, and the CSV string can contain many lines. The `currentField` variable acts as a buffer to collect the characters of one field while we iterate over the string. Finally, the `inQuotes` Boolean keeps track of whether or not we’re currently within a quoted string. It’s the only piece of state we need for this simple parser.

Now we have to fill in the cases for the `switch` statement:

- `(",", false)` — a comma outside of quotes ends the current field
- `("\\n", false)` — a newline outside of quotes ends the current line

- ("\"", _) — a quote toggles the `inQuotes` Boolean
- `default` — in all other cases, we append the current character to `currentField`

```
func parse(csv: String) -> [[String]] {  
    // ...  
    for c in csv {  
        switch (c, inQuotes) {  
            case ("\", false):  
                result[result.endIndex-1].append(currentField)  
                currentField.removeAll()  
            case ("\n", false):  
                result[result.endIndex-1].append(currentField)  
                currentField.removeAll()  
                result.append([])  
            case ("\\"", _):  
                inQuotes = !inQuotes  
            default:  
                currentField.append(c)  
        }  
    }  
    result[result.endIndex-1].append(currentField)  
    return result  
}
```

(We're creating a temporary tuple to switch over two values at once. You may remember this technique from the [Enums](#) chapter.)

After the `for` loop, we still have to append `currentField` one last time before returning the result, because the CSV string might not end on a newline.

Let's try the CSV parser with an example:

```
let csv = #"""  
    "Values in quotes","can contain , characters"  
    "Values without quotes work as well:",42  
    """#  
  
parse(csv: csv)  
/*  
[["Values in quotes", "can contain , characters"],
```

```
["Values without quotes work as well:", "42"]]  
*/
```

The string literal above is using the [extended delimiters syntax](#) (enclosing the string literal in # characters), which allows us to write quotes within the string literal without having to escape them.

Being able to write small parsers like this one significantly enhances your string-handling skills. In this way, tasks that are difficult or impossible to accomplish with Collection APIs or even regular expressions often become easier to write *and* read.

The CSV parser above isn't complete, but it's already useful. It's short because we don't have to track a lot of state; there's just a single Boolean variable. With a bit of extra work, we could ignore empty lines, ignore whitespace around quoted fields, and support escaping of quotes within quoted fields (by using two quote characters). Instead of using a single Boolean for tracking the parser's state, we'd then use an enum to distinguish all possible states unambiguously.

However, the more state we add to the parser, the easier it becomes to make mistakes in its implementation. Therefore, this approach of parsing within a single loop is only advisable for small parsers. If we have to keep track of more state, we'd have to change the strategy from writing everything in a single loop to breaking the parser up into multiple functions.

Substrings

Like all collections, String has a specific slice, or SubSequence type, named Substring. A substring is much like an ArraySlice: it's a view of a base string with different start and end indices. Substrings share the text storage of their base strings. This has the huge benefit that slicing a string is an inexpensive operation. Creating the firstWord variable in the following example requires no expensive copies or memory allocation:

```
let sentence = "The quick brown fox jumped over the lazy dog."  
let firstSpace = sentence.firstIndex(of: " ") ?? sentence.endIndex  
let firstWord = sentence[..  
type(of: firstWord) // Substring
```

Slicing being cheap is especially important in loops where you iterate over the entire (potentially long) string to extract its components. Tasks like finding all occurrences of a word in a text, or parsing CSV data, like we did above, come to mind. A useful string processing operation in this context is splitting. The `split` method is defined on `Collection` and returns an array of subsequences (i.e. `[Substring]`). Its most common variant is defined like so:

```
extension Collection where Element: Equatable {
    public func split(separator: Element, maxSplits: Int = Int.max,
                      omittingEmptySubsequences: Bool = true) -> [SubSequence]
}
```

You can use it like this:

```
let poem = """
Over the wintry
forest, winds howl in rage
with no leaves to blow.
"""

let lines = poem.split(separator: "\n")
// ["Over the wintry", "forest, winds howl in rage", "with no leaves to blow."]
type(of: lines) // Array<Substring>
```

This can serve a function similar to the `components(separatedBy:)` method `String` inherits from `NSString`, with added configurations for whether or not to drop empty components. Again, no copies of the input string are made. And since there's another variant of `split` that takes a closure, it can do more than just compare characters. Here's an example of a primitive word wrap algorithm, where the closure captures a count of the length of the line thus far:

```
extension String {
    func wrapped(after maxLength: Int = 70) -> String {
        var lineLength = 0
        let lines = self.split(omittingEmptySubsequences: false) { character in
            if character.isWhitespace && lineLength >= maxLength {
                lineLength = 0
                return true
            } else {
                lineLength += 1
                return false
            }
        }
    }
}
```

```
    }
    return lines.joined(separator: "\n")
}
}

sentence.wrapped(after: 15)
/*
The quick brown
fox jumped over
the lazy dog.
*/
```

Or, consider writing a version that takes a sequence of multiple separators:

```
extension Collection where Element: Equatable {
    func split<S: Sequence>(separators: S) -> [SubSequence]
        where Element == S.Element
    {
        return split { separators.contains($0) }
    }
}
```

This way, you can write the following:

```
"Hello, world!".split(separators: " , ") // ["Hello", "world"]
```

StringProtocol

Substring has almost the same interface as String. This is achieved through a common protocol named `StringProtocol`, which both types conform to. Since almost the entire string API is defined on `StringProtocol`, you can mostly work with a `Substring` as you would with a `String`. At some point, though, you'll have to turn your substrings back into `String` instances; like all slices, substrings are only intended for short-term storage so as to avoid expensive copies during an operation. When the operation is complete and you want to store the results or pass them on to another subsystem, you should create a new `String`. You can do this by initializing a `String` with a `Substring`, as we do in this example:

```
func lastWord(in input: String) -> String? {
    // Process the input, working on substrings.
```

```
let words = input.split(separators:[",", " "])
guard let lastWord = words.last else { return nil }
// Convert to String for return.
return String(lastWord)
}

lastWord(in: "one, two, three, four, five") // Optional("five")
```

The rationale for discouraging long-term storage of substrings is that a substring always holds on to the entire original string. A substring representing a single character of a huge string will hold the entire string in memory, even after the original string's lifetime would normally have ended. Long-term storage of substrings would therefore effectively cause memory leaks because the original strings have to be kept in memory even when they're no longer accessible.

By working with substrings during an operation and only creating new strings at the end, we defer copies until the last moment and make sure to only incur the cost of those copies that are actually necessary. In the example above, we split the entire (potentially long) string into substrings, but we only pay the cost for a single copy of one short substring at the end. (Ignore for a moment that this algorithm isn't efficient anyway; iterating backward from the end until we find the first separator would be the better approach.)

Encountering a function that only accepts a `Substring` when you want to pass a `String` is less common — most functions should take either a `String` or any `StringProtocol`-conforming type. But if you do need to pass a `Substring`, the quickest way is to subscript the string with the range operator ... without specifying any bounds:

```
// Substring that encompasses the entire string.
let substring = sentence[...]
```

You may be tempted to take full advantage of the existence of `StringProtocol` and convert all your APIs to take `StringProtocol` instances rather than plain `Strings`. But the advice of the Swift team is not to do that:

Our general advice is to stick with `String`. Most APIs would be simpler and clearer just using `String` rather than being made generic (which itself can come at a cost), and user conversion on the way in on the few occasions that's needed isn't much of a burden.

APIs that are extremely likely to be used with substrings, and at the same time aren't further generalizable to the Sequence or Collection level, are an exception to this rule. An example of this in the standard library is the joined method. The standard library provides an overload for sequences with StringProtocol-conforming elements:

```
extension Sequence where Element: StringProtocol {  
    /// Returns a new string by concatenating the elements of the sequence,  
    /// adding the given separator between each element.  
    public func joined(separator: String = "") -> String  
}
```

This lets you call joined directly on an array of substrings (which you got from a call to split, for example) without having to map over the array and copy every substring into a new string. This is more convenient and much faster.

The number-type initializers that take a string and convert it into a number also take StringProtocol values. Again, this is especially handy if you want to process an array of substrings:

```
let commaSeparatedNumbers = "1,2,3,4,5"  
let numbers = commaSeparatedNumbers.split(separator: ",")  
    .compactMap { Int($0) }  
numbers // [1, 2, 3, 4, 5]
```

Since substrings are intended to be short-lived, it's generally not advisable to return one from a function unless you're dealing with Sequence or Collection APIs that return slices. If you write a similar function that only makes sense for strings, having it return a substring tells readers it doesn't make a copy. However, functions that create new strings requiring memory allocations, such as uppercased(), should always return String instances.

If you want to extend String with new functionality, placing the extension on StringProtocol is a good idea to keep the API surface between String and Substring consistent. StringProtocol is explicitly designed to be used whenever you would've previously extended String. If you want to move existing extensions from String to StringProtocol, the only change you should have to make is to replace any passing of self into an API that takes a concrete String with String(self).

Keep in mind, though, that StringProtocol isn't intended as a conformance target for your own custom string types. The documentation explicitly warns against it:

Do not declare new conformances to `StringProtocol`. Only the `String` and `Substring` types of the standard library are valid conforming types.

Code Unit Views

Sometimes it's necessary to drop down to a lower level of abstraction and operate directly on Unicode scalars or code units instead of Swift characters (i.e. grapheme clusters). `String` provides three views for this: `unicodeScalars`, `utf8`, and `utf16`. Like `String`, these are bidirectional collections that support all the familiar operations. And like substrings, the views share the string's storage; they simply represent the underlying bytes in a different way.

There are a few common reasons why you'd need to work on one of the views. Firstly, maybe you actually need the code units, perhaps for rendering into a UTF-8-encoded webpage, or for interoperating with a non-Swift API that expects a particular encoding. Or maybe you need information about the string in a specific format.

For example, suppose you're writing a Twitter client. While the Twitter API expects strings to be UTF-8-encoded, Twitter's character-counting algorithm is based on NFC-normalized scalars (at least that's how it used to be — the algorithm has become more complicated in recent years, but we'll stick to the previous approach for the sake of this example). So if you want to show your users how many characters they have left, this is how you could do it:

```
let tweet = "Having ☕ in a cafe\u{301} in 🇺🇸 and enjoying the ☀️."  
let characterCount = tweet.precomposedStringWithCanonicalMapping  
    .unicodeScalars.count  
characterCount // 46
```

NFC normalization converts base letters and combining marks, such as the e plus accent in "cafe\u{301}", into their precomposed forms. The `precomposedStringWithCanonicalMapping` property is defined in Foundation.

UTF-8 is the de facto standard for storing text or sending it over the internet. Since the `utf8` view is a collection, you can use it to pass the raw UTF-8 bytes to any other API that accepts a sequence of bytes, such as the initializers for `Data` or `Array`:

```
let utf8Bytes = Data(tweet.utf8)
```

```
utf8Bytes.count // 62
```

The UTF-8 view on `String` also has the least overhead of all code unit views, because UTF-8 is the native in-memory format for Swift strings.

Note that the `utf8` collection doesn't include a trailing null byte. If you need a null-terminated representation, use the `withCString` method or the `utf8CString` property on `String`. The latter returns an array of bytes:

```
let nullTerminatedUTF8 = tweet.utf8CString  
nullTerminatedUTF8.count // 63
```

The `withCString` method calls a function you provide with a pointer to the null-terminated UTF-8 contents of the string. This is useful if you need to call a C API that expects a `char *`. In many cases, you don't even need the explicit `withCString` call, because the compiler can automatically convert a Swift `String` to a C string for a function call. For example, here's a call to the `strlen` function in the C standard library:

```
strlen(tweet) // 62
```

We'll see more examples of this in the [Interoperability](#) chapter. In most cases (if the string's underlying storage is already UTF-8), this conversion has almost no cost because Swift can pass a direct pointer into the string's storage to C (because the storage actually *does* include a trailing null byte). If the string has a different in-memory encoding, the compiler will automatically insert code to transcode the contents and copy said contents into a temporary buffer.

The `utf16` view has a special significance because Foundation APIs traditionally treat strings as collections of UTF-16 code units. While the `NSString` interface is transparently bridged to Swift `String`, thereby handling the transformations implicitly for you, other Foundation APIs, such as `NSRegularExpression` or `NSAttributedString`, often expect input in terms of UTF-16 data. We'll see an example of this in the [Strings and Foundation](#) section.

A second reason for using the code unit views is that operating on code units rather than fully composed characters can be faster. This is because the Unicode grapheme-breaking algorithm is fairly complex and requires additional lookahead to identify the start of the next grapheme cluster. Traversing the `String` as a `Character` collection has gotten much faster in recent years, however, so be sure to measure if the (relatively small) speedup is worth losing Unicode correctness. As soon as you drop down to one of the code unit views, you must be certain that your specific algorithm

works correctly in that context. For example, using the UTF-8 view to parse JSON would be alright because all special characters the parser is interested in (such as commas, quotes, or braces) are representable in a single code unit; it doesn't matter that some strings in the JSON data may contain complex emoji sequences. (As mentioned earlier, pay special attention to newlines. "\r\n" is a single Character, but it's two scalars or code units.) On the other hand, finding all occurrences of a word in a string wouldn't work correctly on a code unit view if you want the search algorithm to find different normalization forms of the search string.

One desirable feature *none* of the code unit views provide is random access. The consequence is that String and its views are a bad match for algorithms that require random access. The vast majority of string-processing tasks should work fine with sequential traversal, especially since an algorithm can always store substrings for fragments it wants to be able to revisit in constant time. If you absolutely need random access, you can always convert the string itself or one of its views into an array and work on that, as with Array(str) or Array(str.utf8). For maximum performance at the cost of safety, there's also withUTF8(str) { buffer in...}, which provides a temporary pointer into the string's storage.

Index Sharing

Strings and their views share the same index type, String.Index. This means you can use an index derived from a string to subscript one of the views. In the following example, we search the string for "é" (which consists of two scalars, the letter e and the combining accent). The resulting index refers to the first scalar in the Unicode scalar view:

```
let pokemon = "Poke\u{301}mon" // Pok  mon
if let index = pokemon.firstIndex(of: " ") {
    let scalar = pokemon.unicodeScalars[index] // e
    String(scalar) // e
}
```

This works great as long as you go down the abstraction ladder, from characters to scalars to UTF-8 or UTF-16 code units. Going the other way can have surprising results, because not every valid index in one of the code unit views is on a Character boundary:

```
let flag = "  " // [N] + [Z]
// This initializer creates an index at a UTF-16 offset.
let someUTF16Index = String.Index(utf16Offset: 2, in: flag)
```

```
flag[someUTF16Index] // Z
```

String.Index has a set of methods (`samePosition(in:)`) and failable initializers (`(String.Index.init?(_:within:))`) for converting indices between views. These return `nil` if the given index has no exact corresponding position in the specified view. For example, trying to convert the position of the combining accent in the scalars view to a valid index in the string fails because the combining character doesn't have its own position in the string:

```
if let accentIndex = pokemon.unicodeScalars.firstIndex(of: "\u{301}") {  
    accentIndex.samePosition(in: pokemon) // nil  
}
```

Strings and Foundation

Swift's String type has a very close relationship with its Foundation counterpart, `NSString`. Any `String` instance can be bridged to `NSString` using the `as` operator, and Objective-C APIs that take or return an `NSString` are automatically translated to use `String`. As of Swift 5.5, `String` still lacks a lot of functionality that `NSString` possesses, but since strings are such fundamental types, and since constantly having to cast to `NSString` would be annoying, `String` receives special treatment from the compiler: when you import Foundation, `NSString` members become directly accessible on `String` instances, making Swift strings significantly more capable than they'd otherwise be.

Having the additional features is undoubtedly a good thing, but it can make working with strings somewhat confusing. For one thing, if you forget to import Foundation, you may wonder why some methods aren't available. Foundation's history as an Objective-C framework also tends to make the `NSString` APIs feel a little bit out of place next to the standard library, if only because of different naming conventions. Last but not least, overlap between the feature sets of the two libraries sometimes means there are two APIs with completely different names that perform nearly identical tasks. If you're a long-time Cocoa developer and learned the `NSString` API before Swift came along, this is probably not a big deal, but it will be puzzling for newcomers.

We've already seen one example — the standard library's `split` method vs. `components(separatedBy:)` in Foundation — and there are numerous other mismatches: Foundation uses `ComparisonResult` enums for comparison predicates, while the standard library is designed around Boolean predicates; methods like `trimmingCharacters(in:)` and `components(separatedBy:)` take a `CharacterSet` as an

argument, which is an unfortunate misnomer in Swift (more on that later); and the extremely powerful `enumerateSubstrings(in:options:_:)` method, which can iterate over a string in chunks of grapheme clusters, words, sentences, or paragraphs, deals with strings and ranges where a corresponding standard library API would use substrings. (The standard library could also expose the same functionality as a lazy sequence, which would be very cool.) Foundation APIs are often locale-aware, too, i.e. they can compare or enumerate strings based on localization-specific criteria. The standard library intentionally ignores this important aspect of text processing.

The following example enumerates the words in a string. The callback closure is called once for every word found:

```
let sentence = """
    The quick brown fox jumped \
    over the lazy dog.
"""

var words: [String] = []
sentence.enumerateSubstrings(in:sentence.startIndex..., options: .byWords) {
    (word, range, _, _) in
    guard let word = word else { return }
    words.append(word)
}
words
// ["The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]
```

To get an overview of all `NSString` members imported into `String`, check out [NSStringAPI.swift](#) in the Foundation source code.

Due to the mismatch between the native in-memory encodings of Swift strings (UTF-8) and `NSString` (UTF-16), there's an additional performance cost when a Swift string has to be bridged to an `NSString`. This means that passing a native Swift string to a Foundation API such as `enumerateSubstrings(in:options:using:)` may not be as quick as passing an `NSString` — the method may assume it's able to jump around the string in terms of UTF-16 offsets in constant time, but this would be a linear-time operation on a Swift string. To mitigate this effect, Swift implements [sophisticated index caching](#) to achieve [amortized constant time](#) characteristics.

Other String-Related Foundation APIs

Having said all that, native `NSString` APIs are generally pleasant to use with Swift strings because most of the bridging work is done for you. Some other Foundation APIs that deal with strings are a lot trickier to use because Apple hasn't (yet?) written special Swift overlays for them. Consider `NSRegularExpression`, the Foundation class that represents regular expressions for matching strings against search patterns. To successfully use this class from Swift, you have to be aware of the following:

- Although all `NSRegularExpression` APIs that originally take an `NSString` now take a Swift `.String`, the entire API is still based on `NSString`'s concept of a collection of UTF-16 code units. Foundation uses ranges of type `NSRange` to represent "substrings," and these invariably work in terms of UTF-16 code units.
- Less importantly, frequent bridging between `String` and `NSString` might come with unexpected performance costs. See [session 229 at WWDC 2018](#) for details on this.

For example, the `rangeOfFirstMatch(in:options:range:)` method returns the location of the first match of the regex as an `NSRange`, and not as a `Range<String.Index>`. On top of that, because Objective-C doesn't have optionals, the method returns the sentinel value `NSRange(location: NSNotFound, length: 0)` to represent "not found."

`NSRange` is a structure that contains two integer fields — `location` and `length`:

```
public struct NSRange {  
    public var location: Int  
    public var length: Int  
}
```

In the context of strings, the fields specify a string segment in terms of UTF-16 code units. Swift provides initializers for converting between `Range<String.Index>` and `NSRange`. This makes working with `NSRange` less error-prone than having to remember to drop down to the UTF-16 view all the time; however, it doesn't make the extra code required to translate back and forth any shorter. Here's an example of how you'd create a regular expression and match it against a search string:

```
extension NSRegularExpression {  
    func firstMatch(in input: String) -> Substring? {  
        // NSRegularExpression expects the "substring" in which  
        // to search as an NSRange.  
    }  
}
```

```

var inputRange = NSRange(input.startIndex..., in: input)
let utf16Length = inputRange.length
while true {
    let matchNSRange = self.rangeOfFirstMatch(in: input, range: inputRange)
    guard matchNSRange != NSRange(location: NSNotFound, length: 0) else {
        // Pattern not found.
        return nil
    }
    // Convert NSRange to Swift range.
    guard let matchRange = Range(matchNSRange, in: input) else {
        // Match doesn't fall on a Character boundary → start over.
        inputRange.location =
            matchNSRange.location + matchNSRange.length
        inputRange.length = utf16Length - inputRange.location
        continue
    }
    return input[matchRange]
}
}

let input = "My favorite numbers are -9, 27, and 81."
let regex = try! NSRegularExpression(pattern: "-?[0-9]+")
if let match = regex.firstMatch(in: input) {
    print("Found: \(match)")
} else {
    print("Not found")
}
// Found: -9

```

The `let regex = try!...` line is a good example of where force-unwrapping (or rather, its equivalent for error handling) is justified. We're initializing `NSRegularExpression` with a string literal. If this crashes during development because we passed in an invalid regex, we can easily fix it, but it'll never fail in production. See the [Optionals](#) chapter for more on this.

You may wonder why we need the loop in the code above. `NSRegularExpression` looks at strings in terms of Unicode scalars. This means it may find a match on a scalar that's *part of* a grapheme cluster (aka *Character*). Here's an example where we search for one of two scalars that make up a flag emoji:

```

let flag = "🇧🇷"

let regex2 = try! NSRegularExpression(pattern: "[B]")
regex2.rangeOfFirstMatch(in: flag, range: NSRange(flag.startIndex..., in: flag))
// {0, 2}

```

This is a valid match as far as `NSRegularExpression` is concerned, but it's probably not what a Swift programmer would expect or want. Moreover, because the match falls on part of a Character, it's not a valid range in a Swift string. When we can't convert the match's `NSRange` back to a `Range<String.Index>`, we ignore the match and start over from the match position.

In conclusion, we were able to write a nice and safe wrapper for the original API by:

- Hiding all uses of `NSRange`
- Translating between the different ways `String` and `Foundation` deal with Unicode
- Using optionals instead of sentinel values to represent “no match”
- Returning a substring instead of a range

Let's use the same approach for finding all matches in a string, and not just the first. The equivalent `NSRegularExpression` API is called `matches(in:options:range:)`, and it returns an array of `NSTextCheckingResult` objects. This is a fairly complex class, but for our purposes, it's enough to know that it provides the `NSRange` of the match it represents. The final code is actually a little simpler than `firstMatch(in:)` above because we can omit the loop:

```

extension NSRegularExpression {
    func matches(in input: String) -> [Substring] {
        let inputRange = NSRange(input.startIndex..., in: input)
        let matches = self.matches(in: input, range: inputRange)
        return matches.compactMap { match in
            guard match.range != NSRange(location: NSNotFound, length: 0) else {
                // NSTextChecking should never have a "nil" range.
                fatalError("Should never happen")
            }
            guard let matchRange = Range(match.range, in: input) else {
                // Match doesn't fall on a Character boundary.
                return nil
            }

```

```
        return input[matchRange]
    }
}
}

regex.matches(in: input) // [-9, "27", "81"]
```

We think you'll agree that it's a bit of work to make these Foundation APIs feel like idiomatic Swift. And having to use an API that deals with Unicode in a different way isn't just an inconvenience; it can easily introduce hard-to-find bugs. It's likely Swift will get native regular expression syntax in the not-too-distant future, but until then, this is the way to go.

Aside from `NSRegularExpression`, another Foundation class with similar impedance mismatches is `NSAttributedString`, which is used to represent rich text. Apple introduced a modern version of this named struct `AttributedString` that feels much more at home in Swift. We recommend you use it if your deployment target allows it.

Ranges of Characters

Speaking of ranges, you may have tried to iterate over a range of characters and found it doesn't work:

```
let lowercaseLetters = ("a" as Character)..."z"
for c in lowercaseLetters { // Error
    ...
}
```

(The cast to `Character` is important because the default type of a string literal is `String`; we need to tell the type checker we want a `Character` range.)

In the Built-In Collections chapter, we talked about the reason why this fails: `Character` doesn't conform to the `Strideable` protocol, which is a requirement for ranges to become *countable* and thus collections. The only thing you can do with a range of characters is compare other characters against it, i.e. check whether or not a character is inside the range:

```
lowercaseLetters.contains("A") // false
lowercaseLetters.contains("c") // true
```

```
lowercaseLetters.contains("é") // false
```

One type for which the notion of countable ranges does make sense is `Unicode.Scalar` — at least if you stick to ASCII or other well-ordered subsets of the Unicode catalog. Scalars have a well-defined order through their code point values, and there's always a finite number of scalars between any two bounds. `Unicode.Scalar` isn't `Strideable` by default, but we can add the conformance retroactively:

```
extension Unicode.Scalar: Strideable {
    public typealias Stride = Int

    public func distance(to other: Unicode.Scalar) -> Int {
        return Int(other.value) - Int(self.value)
    }

    public func advanced(by n: Int) -> Unicode.Scalar {
        return Unicode.Scalar(UInt32(Int(value) + n))!
    }
}
```

(We're ignoring the fact that the surrogate code points 0xD800 to 0xDFFF aren't valid Unicode scalar values. Constructing a range that overlaps this region is a programmer error.)

Conforming a type you don't own to a protocol you don't own can be problematic and is generally not recommended. Doing so can cause conflicts if other libraries you use add the same extension, or if the original vendor later adds the same conformance (with a potentially different implementation). It's often a better idea to create a wrapper type and add the protocol conformance to that type. We'll come back to this in the [Protocols](#) chapter.

Adding `Strideable` conformance to `Unicode.Scalar` allows us to use a range of Unicode scalars as a convenient way of generating an array of characters:

```
let lowercase = ("a" as Unicode.Scalar)..."z"
Array(lowercase.map(Character.init)
/*
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n",
```

```
"o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]  
*/
```

CharacterSet

Let's look at one last interesting Foundation type, and that's `CharacterSet`. A `CharacterSet` is an efficient way to store a set of Unicode code points. It's often used to check if a particular string only contains characters from a specific character subset, such as alphanumerics or decimalDigits. The name `CharacterSet`, imported from Objective-C, is unfortunate in Swift because `CharacterSet` isn't compatible with Swift's `Character` type. A better name would be `UnicodeScalarSet`.

We can illustrate this by creating a set from a couple of complex emoji. It seems as though the set only contains the two emoji we put in, but testing for membership with a third emoji is successful because the firefighter emoji is really a sequence of *woman* + ZWJ + *fire engine*:

```
let favoriteEmoji = CharacterSet("👩🚒".unicodeScalars)  
// Wrong! Or is it?  
  
favoriteEmoji.contains("🚒") // true
```

`CharacterSet` provides a number of factory initializers like `.alphanumerics` and `.whitespacesAndNewlines`. Most of these correspond to official [Unicode character categories](#) (each code point is assigned a category, such as “Letter” or “Nonspacing Mark”). The categories cover all scripts, and not just ASCII or Latin-1, so the number of members in these predefined sets is often huge. Since `CharacterSet` conforms to `SetAlgebra`, we can combine multiple character sets by using set operations such as unions or intersections.

Unicode Properties

Part of the functionality of `CharacterSet` has been integrated into `Unicode.Scalar` in Swift 5. We no longer need the Foundation type to test a scalar for membership in official Unicode categories, because these are now exposed directly as properties on `Unicode.Scalar`, such as `isEmoji` or `isWhitespace`. To avoid cluttering the main

Unicode.Scalar namespace, the Unicode properties are namespaced under the name properties:

```
("\u{1f600}" as Unicode.Scalar).properties.isEmoji // true  
("\u{ff1f}" as Unicode.Scalar).properties.isMath // true
```

Check out the documentation for [Unicode.Scalar.Properties](#) for the full list. Most of these properties are Boolean, but not all of them are: there are also things like age (the Unicode version when the scalar was introduced), name (the official Unicode name), numericValue (useful for fractions that have their own code point or numbers in non-Latin scripts), and generalCategory (an enum describing a scalar's "first-order, most usual categorization").

For example, listing the code point, name, and general category for each scalar that makes up a string requires nothing more than a little bit of string formatting:

```
"I'm a 🐱.".unicodeScalars.map { scalar -> String in  
    let codePoint = "U+\(String(scalar.value, radix: 16, uppercase: true))"  
    let name = scalar.properties.name ?? "(no name)"  
    return "\(codePoint): \(name) - \(scalar.properties.generalCategory)"  
}.joined(separator: "\n")  
/*  
U+49: LATIN CAPITAL LETTER I – uppercaseLetter  
U+2019: RIGHT SINGLE QUOTATION MARK – finalPunctuation  
U+6D: LATIN SMALL LETTER M – lowercaseLetter  
U+20: SPACE – spaceSeparator  
U+61: LATIN SMALL LETTER A – lowercaseLetter  
U+20: SPACE – spaceSeparator  
U+1F469: WOMAN – otherSymbol  
U+1F3FD: EMOJI MODIFIER FITZPATRICK TYPE-4 – modifierSymbol  
U+200D: ZERO WIDTH JOINER – format  
U+1F692: FIRE ENGINE – otherSymbol  
U+2E: FULL STOP – otherPunctuation  
*/
```

The Unicode scalar properties are fairly low level and intentionally use the Unicode standard's sometimes obscure terminology. It's often useful to have similar categories on the level of Characters, so Swift 5 also added a bunch of related properties to Character. Here are some examples:

```
Character("4").isNumber // true  
Character("$").isCurrencySymbol // true  
Character("\n").isNewline // true
```

Unlike the Unicode scalar properties, these categories on `Character` aren't an official part of the Unicode specification because Unicode only categorizes scalars, and not extended grapheme clusters. The standard library tries its best to provide sensible information about the nature of a character, but due to the large number of supported scripts and Unicode's ability to combine scalars in infinite combinations, these categories aren't exact and may not match what other tools or programming languages provide. They might also evolve over time.

Internal Structure of String and Character

Like the other collection types in the standard library, strings are copy-on-write collections with value semantics. A `String` instance stores a reference to a buffer, which holds the actual character data. When you make a copy of a string (through assignment or by passing it to a function) or create substrings, all the instances share the same buffer. The character data is only copied when an instance gets mutated while it's sharing its character buffer with one or more other instances. For more on copy-on-write, see the [Structs and Classes](#) chapter.

`String` uses UTF-8 as its in-memory representation for native Swift strings. You may be able to use this knowledge to your advantage if you require the best possible performance — traversing the UTF-8 view may be a bit faster than traversing the UTF-16 view or the Unicode scalars view. Also, UTF-8 is the natural format for most string processing, because the majority of source data from files or the web uses the UTF-8 encoding.

Strings received from Objective-C are backed by an `NSString`. In this case, the `NSString` acts directly as the Swift string's buffer to make the bridging efficient. An `NSString`-backed `String` will be converted to a native Swift string when it gets mutated.

For small strings of up to 15 UTF-8 code units (or 10 code units on 32-bit platforms) there's a special optimization that avoids allocating a backing buffer altogether. Since strings are 16 bytes wide, the code units of small strings can be stored inline. Although 15 UTF-8 code units might not sound like much, it's enough for a lot of strings. For example, in machine-readable formats like JSON, many keys and values (e.g. numbers

and Booleans) fit into this length, especially because they often only use ASCII characters.

This optimization is also leveraged for the internal representation of the Character type (simplified from the [standard library source](#)):

```
public struct Character {  
    internal var _str: String  
  
    internal init(unchecked str: String) {  
        self._str = str  
        // ...  
    }  
}
```

A character is represented internally as a string of length one. Since almost all characters fit into 15 UTF-8 bytes (long emoji sequences may be the most common exception), the small string optimization means that the vast majority of Character values don't require heap allocation or reference counting.

String Literals

Throughout this chapter, we've been using `String("Hello")` and "Hello" pretty much interchangeably, but they're different. `" "` is a string literal, just like the array literals covered in the [Collection Protocols](#) chapter. You can make your own types initializable from a string literal by conforming to `ExpressibleByStringLiteral`.

String literals are slightly more complicated than array literals because they're part of a hierarchy of three protocols: `ExpressibleByStringLiteral`, `ExpressibleByExtendedGraphemeClusterLiteral`, and `ExpressibleByUnicodeScalarLiteral`. Each defines an `init` for creating a type from each kind of literal, but unless you really need fine-grained logic based on whether or not the value is being created from a single scalar/cluster, you only need to implement the string version; the others are covered by default implementations that refer to the string literal initializer.

As an example of a custom type adopting the `ExpressibleByStringLiteral` protocol, we define a `SafeHTML` type. This is really just a wrapper around a string but with added type safety. When we use a value of this type, we can be sure that the potentially dangerous HTML tags it contains have been escaped to avoid posing a security risk:

```

extension String {
    var htmlEscaped: String {
        // Replace all opening and closing brackets.
        // A "real" implementation would be more sophisticated.
        return replacingOccurrences(of: "<", with: "&lt;")
            .replacingOccurrences(of: ">", with: "&gt;")
    }
}

struct SafeHTML {
    private(set) var value: String

    init(unsafe html: String) {
        self.value = html.htmlEscaped
    }
}

```

We could use this type to make sure the API of our views only accepts values that are properly escaped. The disadvantage is that we'll have to manually wrap many string literals in our code in `SafeHTML`. Luckily, we can conform to `ExpressibleByStringLiteral` to avoid this overhead:

```

extension SafeHTML: ExpressibleByStringLiteral {
    public init(stringLiteral value: StringLiteralType) {
        self.value = value
    }
}

```

This assumes string literals in our code are always safe (which is a fair assumption, since we typed them ourselves):

```

let safe: SafeHTML = "<p>Angle brackets in literals are not escaped</p>"
// SafeHTML(value: "<p>Angle brackets in literals are not escaped</p>")

```

Above, we have to explicitly specify the `SafeHTML` type, otherwise `safe` would be of type `String`. However, we can omit the explicit types in contexts where the compiler already knows the type, such as when dealing with property assignment or function calls.

String Interpolation

String interpolation, i.e. putting expressions into string literals (such as "a * b = \(\a * \b\)"), has existed in Swift since the beginning. Swift 5 introduced a public API to leverage string interpolation for custom types. We can use this API to improve the SafeHTML type from above. We often have to write string literals containing HTML tags with some user-input data in between:

```
let input = ... // received from user, unsafe!
let html = "<li>Username: \(input)</li>"
```

input must be escaped because it comes from an untrusted source, but the string literal segments should stay untouched, since we want to write HTML tags in there. We can achieve this by implementing a custom string interpolation rule for our SafeHTML type.

Swift's string interpolation API consists of two protocols:

ExpressibleByStringInterpolation and StringInterpolationProtocol. The former has to be adopted by the type that should be constructed by string interpolation. The latter can be adopted either by the same type or by a separate type and contains several methods to build up the interpolated value step by step.

ExpressibleByStringInterpolation inherits from ExpressibleByStringLiteral. We implemented the latter for the SafeHTML type above, so we can skip it here. We conform to ExpressibleByStringInterpolation by implementing an initializer that can construct a SafeHTML value from a StringInterpolationProtocol value. For this example, we'll use the same type, SafeHTML, to conform to StringInterpolationProtocol:

```
extension SafeHTML: ExpressibleByStringInterpolation {
    init(stringInterpolation: SafeHTML) {
        self.value = stringInterpolation.value
    }
}
```

The StringInterpolationProtocol protocol has three requirements: an initializer, an appendLiteral method, and one or more appendInterpolation methods. Swift has a default type conforming to this protocol, DefaultStringInterpolation, that handles the string interpolation we get for free from the standard library. We want to provide a custom type with an appendInterpolation method that escapes the interpolated value:

```

extension SafeHTML: StringInterpolationProtocol {
    init(literalCapacity: Int, interpolationCount: Int) {
        self.value = ""
        value.reserveCapacity(literalCapacity)
    }

    mutating func appendLiteral(_ literal: String) {
        value += literal
    }

    mutating func appendInterpolation<T>(_ x:T) {
        self.value += String(describing: x).htmlEscaped
    }
}

```

The initializer informs the interpolation type about the approximate capacity needed to store all literals combined, as well as the number of interpolations we have to expect. We could ignore these two parameters and just initialize value with an empty string. However, it's good practice to at least pass the literalCapacity value to `String.reserveCapacity`. This tells the string to allocate enough storage for the size we expect, avoiding multiple expensive reallocations along the way. It's not perfect because we don't know in advance how big the interpolated segments will be, but it's better than nothing. We could improve this further by adding an estimated size (e.g. 10 bytes) for each interpolation segment.

The `appendLiteral` method simply appends a string to the `value` property, since we consider string literals to be safe by default (just as with `ExpressibleByStringLiteral` above). The `appendInterpolation(_)` method takes an input parameter of any type and uses `String(describing:)` to turn it into a string. We escape this string before appending it to `value`.

Since the `appendInterpolation` method has no label on the parameter, we can use it just like we use Swift's default string interpolation:

```

let unsafeInput = "<script>alert('Oops!')</script>"
let safe: SafeHTML = "<li>Username: \\"unsafeInput"</li>"
safe
/*
SafeHTML(value: "<li>Username:
<script>alert('Oops!')</script></li>")
*/

```

The compiler translates the interpolated string into a series of `appendLiteral` and `appendInterpolation` calls on our custom `StringInterpolationProtocol` type, giving us the chance to store this data how we see fit. Once all the literal and interpolation segments have been processed, the resulting value is passed to the `init(stringInterpolation:)` initializer to create the `SafeHTML` value.

In our case, we chose to conform the same type to both `ExpressibleByStringInterpolation` and `StringInterpolationProtocol`, because they share the same structure (both only need one string property). However, the ability to use separate types is useful when the data structure you need for building up the string interpolation differs from the structure of the type being constructed by the interpolation.

We can do much more with string interpolation though. Essentially, the `\(...)` syntax is shorthand for an `appendInterpolation` method call, i.e. we can use multiple parameters and labels. We can make use of this behavior to add a “raw” interpolation that allows us to interpolate values without escaping them:

```
extension SafeHTML {  
    mutating func appendInterpolation<T>(raw x:T) {  
        self.value += String(describing:x)  
    }  
}  
  
let star = "<sup>*</sup>"  
let safe2:SafeHTML = "<li>Username\raw: star:\(unsafeInput)</li>"  
safe2  
/*  
SafeHTML(value: "<li>Username<sup>*</sup>:  
&lt;script&gt;alert('Oops!')&lt;/script&gt;</li>"  
*/
```

Custom String Descriptions

Functions like `print`, `String(describing:)`, and string interpolation are written to take any type, no matter what. Even without any customization, the results you get back might be acceptable because structs print their properties by default:

```
let safe:SafeHTML = "<p>Hello, World!</p>"  
print(safe)  
// prints out SafeHTML(value: "<p>Hello, World!</p>")
```

Then again, you might want something a little prettier, especially if your type contains private variables you don't want displayed. But never fear! It only takes a minute to conform your custom type to the `CustomStringConvertible` protocol, giving it a nicely formatted output when it's passed to `print`:

```
extension SafeHTML: CustomStringConvertible {  
    var description: String {  
        return value  
    }  
}
```

Now, if someone converts a `SafeHTML` value to a string through various means — using it with a streaming function like `print`, passing it to `String(describing:)`, or using it in some string interpolation — it'll just print its string value:

```
print(safe) // <p>Hello, World!</p>
```

There's also `CustomDebugStringConvertible`, which you can implement to provide a different output format for debugging purposes. It's used when someone calls `String(reflecting:)`:

```
extension SafeHTML: CustomDebugStringConvertible {  
    var debugDescription: String {  
        return "SafeHTML: \(value)"  
    }  
}
```

`String(reflecting:)` falls back to using `CustomStringConvertible` if `CustomDebugStringConvertible` isn't implemented, and vice versa. Similarly, `String(describing:)` falls back to `CustomDebugStringConvertible` if `CustomStringConvertible` isn't available. So often, `CustomDebugStringConvertible` isn't worth implementing if your type is simple. However, if your custom type is a container, it's probably polite to conform to `CustomDebugStringConvertible` to print the debug versions of the elements the type contains. And if you do anything out of the ordinary when printing for debugging purposes, be sure to implement `CustomStringConvertible` as well. But if your implementations for `description` and `debugDescription` are identical, you can pick one and omit the other.

By the way, `Array` always prints out the debug description of its elements, even when invoked via `String(describing:)`. The reason is that an array's description should never be presented to the user anyway, hence the output of `description` can be optimized for

debugging. And an array of empty strings would look wrong without the enclosing quotes, which `String.description` omits.

Given that conforming to `CustomStringConvertible` implies that a type has a pretty print output, you may be tempted to write something like the following generic function:

```
func doSomethingAttractive<T: CustomStringConvertible>(with value: T) {  
    // Print out something incorporating value, safe in the knowledge  
    // it will print out sensibly.  
}
```

But you're not supposed to use `CustomStringConvertible` in this manner. Instead of poking at types to establish whether or not they have a `description` property, you should use `String(describing:)` regardless and live with the ugly output if a type doesn't conform to the protocol. This will never fail for any type. And it's a good reason to implement `CustomStringConvertible` whenever you write more than a very simple type. It only takes a handful of lines.

LosslessStringConvertible

The last of the `...StringConvertible` protocols is `LosslessStringConvertible`. It builds on `CustomStringConvertible` and adds an initializer for converting from a string *back* to your custom type:

```
public protocol LosslessStringConvertible: CustomStringConvertible {  
    /// Instantiates an instance of the conforming type from a string  
    /// representation.  
    init?(_ description: String)  
}
```

As the name implies, the string description must be a lossless, value-preserving representation of the original value, which allows round trips without losing information. In the standard library, the number types, string types, and `Bool` conform to `LosslessStringConvertible`. The protocol is rarely used. If you need to serialize more than a single value as a string, Swift's `Codable` system is more flexible. See the [Encoding and Decoding](#) chapter for more on this.

Text Output Streams

The print and dump functions in the standard library log text to the [standard output](#). The default versions of these functions forward to overloads named `print(_:to:)` and `dump(_:to:)`. The `:to:` argument is the output target; it can be any type that conforms to the `TextOutputStream` protocol:

```
public func print<Target: TextOutputStream>
(_ items: Any..., separator: String = " ",
 terminator: String = "\n", to output: inout Target)
```

The standard library maintains an internal text output stream that writes everything that's streamed to it to the standard output. What else can you write to? Well, `String` is the only relevant type in the standard library that's an output stream (`Substring` and `DefaultStringInterpolation` are output streams too, but you usually wouldn't write to them):

```
var s = ""
let numbers = [1,2,3,4]
print(numbers, to: &s)
s // [1, 2, 3, 4]
```

This is useful if you want to reroute the output of the `print` and `dump` functions into a string. Incidentally, the standard library also harnesses output streams to allow Xcode to capture all `stdout` logging. Take a look at [this global variable declaration in the standard library](#):

```
public var _playgroundPrintHook: ((String) -> Void)?
```

If this is non-nil, `print` will use a special output stream that routes everything that's printed both to the standard output *and* to this function. The declaration is even public, so you can use this for your own shenanigans:

```
var printCapture = ""
_playgroundPrintHook = { text in
    printCapture += text
}
print("This is supposed to only go to stdout")
printCapture // This is supposed to only go to stdout
```

But don't rely on it! It's totally undocumented, and we don't know what functionality in Xcode will break when you reassign this.

We can also make our own output streams. The protocol has only one requirement: a write method that takes a string and writes it to the stream. For example, this output stream buffers writes to an array:

```
struct ArrayStream: TextOutputStream {
    var buffer: [String] = []
    mutating func write(_ string: String) {
        buffer.append(string)
    }
}

var stream = ArrayStream()
print("Hello", to: &stream)
print("World", to: &stream)
stream.buffer // ["", "Hello", "\n", "", "World", "\n"]
```

The documentation explicitly allows functions that write to an output stream to call write(_:) multiple times per writing operation. That's why the array buffer in the example above contains separate elements for line breaks, and even some empty strings. This is an implementation detail of the print function that may change in future releases.

Another possibility is to extend Data so that it takes a stream, writing it as UTF-8-encoded output:

```
extension Data: TextOutputStream {
    mutating public func write(_ string: String) {
        self.append(contentsOf: string.utf8)
    }
}

var utf8Data = Data()
var string = "café"
utf8Data.write(string)
Array(utf8Data) // [99, 97, 102, 195, 169]
```

The source of an output stream can be any type that conforms to the TextOutputStreamable protocol. This protocol requires a generic method, write(to:), which accepts any type that conforms to TextOutputStream and writes self to it. In the

standard library, String, Substring, Character, and Unicode.Scalar conform to TextOutputStreamable, but you can also add conformance to your own types.

As we've seen, internally, print is using some TextOutputStream-conforming wrapper on the standard output. You could write something similar for standard error, like this:

```
struct StdErr: TextOutputStream {
    mutating func write(_ string: String) {
        guard !string.isEmpty else { return }

        // Strings can be passed directly into C functions that take a
        // const char* — see the Interoperability chapter for more.
        fputs(string, stderr)
    }
}

var standarderror = StdErr()
print("oops!", to: &standarderror)
```

Recap

Strings in Swift are very different than their counterparts in almost all other mainstream programming languages. When you're used to strings effectively being arrays of code units, it'll take a while to switch your mindset to Swift's approach of prioritizing Unicode correctness over simplicity.

Ultimately, we think Swift makes the right choice. Unicode text is much more complicated than what those other languages pretend. In the long run, the time savings from avoided bugs you'd otherwise have written will probably outweigh the time it takes to unlearn integer indexing.

We're so used to random "character" access that we may not realize how rarely this feature is really needed in string-processing code. We hope the examples in this chapter convince you that simple in-order traversal is perfectly fine for most common operations. Forcing you to be explicit about which representation of a string you want to work on — grapheme clusters, Unicode scalars, UTF-8 code units, or UTF-16 code units — is another safety measure; readers of your code will be grateful for it.

When Chris Lattner outlined [the goals for Swift's string implementation](#) in July 2016, he ended with this:

Our goal is to be better at string processing than Perl!

Swift 5.5 still isn't quite there yet — too many desirable features are missing, including the transfer of more string APIs from Foundation to the standard library, native language support for regular expressions, and APIs for formatting and parsing strings. But the good news is that the Swift team has expressed interest in [tackling all these topics in the future](#).

Generics

9

Generic programming is a technique for writing reusable code while maintaining type safety. For example, the standard library uses generic programming to make the `sort` method take a custom comparator function while making sure the types of the comparator's parameters match up with the element type of the sequence being sorted. Likewise, an array is generic over the kind of elements it contains in order to provide a type-safe API for accessing and mutating the array's contents.

When we talk about generic programming in Swift, we usually mean programming with generics (signified by angle brackets in Swift's syntax, e.g. `Array<Int>`). However, it's helpful to see the broader context in which generics exist. Generics are a form of **polymorphism**. Polymorphism means using a single interface or name that works with multiple types.

There are at least four different concepts that can all be grouped under **polymorphic programming**:

- We can define multiple functions with the same name but different types. For example, in the Functions chapter, we defined three different functions named `sortDescriptor`, all of which had different parameter types. This is called **overloading**, or more technically, **ad hoc polymorphism**.
- When a function or method expects a class C, we can also pass in a subclass of C. This is called **subtype polymorphism**.
- When a function has a generic parameter (in angle brackets), we call it a **generic function** (and likewise for generic types). This is called **parametric polymorphism**. The generic parameters are also called **generics**.
- We can define protocols and make multiple types conform to them. This is another (more structured) form of **ad hoc polymorphism**, which we'll discuss in the Protocols chapter.

Which concept is used to solve a particular problem is often a matter of taste. In this chapter, we'll talk about the third technique, parametric polymorphism. Generics are often used together with protocols to specify constraints on generic parameters. We'll see examples of this in the Protocols chapter, but this chapter focuses on just generics.

Generic Types

The most generic function we can write is the identity function, i.e. the function that returns its input unchanged:

```
func identity<A>(_ value: A) -> A {  
    return value  
}
```

The identity function has a single **generic parameter**: for any A we choose, it has the type $(A) \rightarrow A$. However, the identity function has an unlimited number of **concrete types** (types with no generic parameters, such as Int, Bool, and String). For example, if we choose A to be Int, the concrete type is $(\text{Int}) \rightarrow \text{Int}$; if we choose A to be $(\text{String} \rightarrow \text{Bool})$, then its type is $((\text{String}) \rightarrow \text{Bool}) \rightarrow (\text{String}) \rightarrow \text{Bool}$; and so on.

Functions and methods aren't the only generic types. We can also have generic structs, classes, and enums. For example, here's the definition of Optional:

```
enum Optional<Wrapped> {  
    case none  
    case some(Wrapped)  
}
```

We say that Optional is a generic type. When we choose a value for Wrapped, we get a concrete type. For example, Optional<Int> or Optional<UIView> are both concrete types. We can think of Optional (without the angle brackets) as a *type constructor*: given a concrete type (e.g. Int), it constructs a different concrete type (e.g. Optional<Int>).

When we look at the standard library, we see there are many concrete types, but also many generic types (for example: Array, Dictionary, and Result). The Array type has a single generic parameter, Element. This means we can pick any concrete type and use it to create an array. We can also create our own generic types. For example, here's an enum that describes a binary tree with values at the nodes:

```
enum BinaryTree<Element> {  
    case leaf  
    indirect case node(Element, l: BinaryTree<Element>, r: BinaryTree<Element>)  
}
```

BinaryTree is a generic type with a single generic parameter, Element. To create a concrete type, we have to choose a concrete type for Element. For example, we can pick Int:

```
let tree: BinaryTree<Int> = .node(5, l: .leaf, r: .leaf)
```

When we want to turn a generic type into a concrete type, we have to choose exactly one concrete type for each generic parameter. You might already be familiar with this limitation when creating an Array. For example, when we create an empty array, we're required to provide an explicit type; otherwise, Swift doesn't know what concrete type to use for its elements:

```
// Type annotation is required.  
var emptyArray: [String] = []
```

You *can* put values with different concrete types in an array as long as the array's element type is a common supertype of all members. For an array of objects, the compiler automatically infers the element type to be their most specific common superclass:

```
// Type is inferred to be [UIView].  
let subviews = [UILabel(), UISwitch()]
```

In other situations, Swift forces you to explicitly acknowledge that you meant to create an array of Any:

```
// Type annotation is required.  
let multipleTypes: [Any] = [1, "foo", true]
```

We'll discuss Any in the section on [generic functions](#).

Extending Generic Types

The generic parameter Element is available anywhere within the scope of BinaryTree. For example, we can freely use Element as if it were a concrete type when writing an extension on BinaryTree. Let's add a convenience initializer that uses Element for its parameter:

```
extension BinaryTree {  
    init(_ value: Element) {  
        self = .node(value, l: .leaf, r: .leaf)  
    }  
}
```

And here's a computed property that collects all the values in the tree recursively and returns them as an array:

```
extension BinaryTree {
    var values: [Element] {
        switch self {
            case .leaf:
                return []
            case let .node(left, right):
                return left.values + [el] + right.values
        }
    }
}
```

When we call `values` on a `BinaryTree<Int>`, we'll get back an array of integers:

```
tree.values // [5]
```

We can also add generic methods. For example, we can add `map`, which has an additional generic parameter named `T` for the return type of the transformation function and the element type of the transformed tree. Because we define `map` in an extension on `BinaryTree`, we can still use the generic `Element` parameter:

```
extension BinaryTree {
    func map<T>(_ transform: (Element) -> T) -> BinaryTree<T> {
        switch self {
            case .leaf:
                return .leaf
            case let .node(left, right):
                return .node(transform(el),
                            l: left.map(transform),
                            r: right.map(transform))
        }
    }
}
```

Because neither `Element` nor `T` are constrained by a protocol, the caller can choose any concrete type they want for them. We can even pick the same concrete type for both generic parameters:

```
let incremented: BinaryTree<Int> = tree.map { $0 + 1 }
// node(6, l: BinaryTree<Swift.Int>.leaf, r: BinaryTree<Swift.Int>.leaf)
```

But we could also choose different types. In this example, `Element` is `Int`, but `T` is `String`:

```
let stringValues: [String] = tree.map { "\($0)" }.values // ["5"]
```

In Swift, many collections are generic (for example: Array, Set, and Dictionary). However, generics aren't just useful for collections; they're also used throughout the Swift standard library. For example:

- Optional uses a generic parameter to abstract over its wrapped type.
- Result has two generic parameters — one representing a successful value, and another representing an error.
- Unsafe[Mutable]Pointer is generic over the type of memory it points to.
- Key paths are generic over both their root type and the resulting value type.
- Ranges are generic over their element type (called Bound).

Generics vs. Any

Generics and Any can be used for similar purposes, yet they behave very differently. In languages without generics, Any is often used to achieve the same thing but with less type safety. This typically means using runtime programming, such as introspection and dynamic casts, to extract a concrete type from a variable of type Any. Generics can be used to solve many of the same problems, but with the added benefit of compile-time checking that avoids runtime overhead.

When reading code, generics can help us understand what a method or function is doing. For example, consider the type of reduce on arrays (in this example, we'll ignore the fact that reduce is really defined on the Sequence protocol): :

```
extension Array {  
    func reduce<Result>(_ initial: Result,  
                        _ combine: (Result, Element) -> Result)  
}
```

Without looking at the implementation, we can tell a lot from the type signature alone (assuming that reduce has a sensible implementation):

- First of all, we know that reduce is generic over Result, which is also the return type. (Result is the name of the generic parameter. Don't confuse it with enum Result from the standard library.)

- Looking at the input parameters, we can see that the function takes a value of type `Result`, along with a way to combine a `Result` and an `Element` into a new `Result`.
- Because the return type is `Result`, the return value of `reduce` must be either `initial` or the return value from calling `combine`. We know this because the method doesn't have the ability to construct arbitrary values of type `Result`.
- If the array is empty, we don't have an `Element` value, so the only thing that can be returned is `initial`.
- If the array isn't empty, the type leaves some implementation freedom: the method could return `initial` without looking at the elements (though that would border on being a non-sensible implementation), or the method could call `combine` either with one of the elements (e.g. the first or the last element) or sequentially with all of the elements.

Note that there's an infinite number of other possible implementations. For example, the implementation could call `combine` for only some of the elements. It could use runtime type introspection, mutate some global state, or make a network call. However, none of these things would qualify as a sensible implementation, and because `reduce` is defined in the standard library, we can be certain it has a sensible implementation.

Now, consider the same method, defined using `Any`:

```
extension Array {
  func reduce(_ initial: Any, _ combine: (Any, Any) -> Any) -> Any
}
```

This type signature carries much less information, even if we only consider sensible implementations. Just by looking at the type, we can't really tell the relation between the first parameter and the return value. Likewise, it's unclear in which order the arguments are passed to the combining function. It's not even clear that the combining function is used to combine a result and an element.

In a way, the more generic a function or method is, the less it can do. Recall the identity function: it's so generic that there's only one sensible implementation:

```
func identity<A>(_ value: A) -> A {
  return value
}
```

In our experience, generic types are a great help when reading code. More specifically, when we see a very generic function or method, such as `reduce` or `map`, we don't have to guess what it does: the number of possible implementations is limited by the type.

Designing with Generics

Generics can be very useful during the design of your program, as they can help factor out shared functionality and reduce boilerplate. In this section, we'll refactor a non-generic piece of networking code, pulling out the common functionality by using generics.

We'll write an extension on `URLSession`, `loadUser`, that fetches the current user's profile from a web service and parses it into the `User` data type. First, we construct the URL and start a network request. Next, we decode the downloaded data using the `Codable` infrastructure (which we'll discuss in the [Encoding and Decoding](#) chapter):

```
extension URLSession {
    func loadUser() async throws -> User {
        let userURL = webserviceURL.appendingPathComponent("/profile")
        let (data, _) = try await data(from: userURL)
        return try JSONDecoder().decode(User.self, from: data)
    }
}
```

If we wanted to reuse the same function to load a different type (e.g. `BlogPost`), the implementation would be almost the same. We'd duplicate the code and change three things: the return type, the URL, and the expression `User.self` in the `JSONDecoder.decode(_:_from:)` call. But there's a better way — by replacing the concrete type, `User`, with a generic parameter, we can make our `load` method generic while leaving most of the implementation the same. At the same time, instead of parsing JSON directly, we add a parameter to `load`: a `parse` function that knows how to parse the data returned from the web service into a value of type `A` (as we'll see in a bit, this gives us a lot of flexibility):

```
extension URLSession {
    func load<A>(url: URL, parse: (Data) throws -> A)
        async throws -> A
    {
        let (data, _) = try await data(from: url)
```

```
    return try parse(data)
}
}
```

The new load function takes the URL and a parse function as parameters because these inputs now depend on the specific endpoint to load. This refactoring follows the same strategy we saw for map and other standard library methods in the [Built-In Collections](#) chapter:

0. Identifying the common pattern in a task (loading data from an HTTP URL and parsing the response).
1. Extracting the boilerplate code that performs this task into a generic method.
2. Allowing clients to inject the things that vary from call to call (the particular URL to load and how to parse the response) via generic parameters and function arguments.

We can now call load with two different endpoints and almost no code duplication:

```
let profileURL = webServiceURL.appendingPathComponent("profile")
let user = try await URLSession.shared.load(url: profileURL, parse: {
    try JSONDecoder().decode(User.self, from: $0)
})
print(user)

let postURL = webServiceURL.appendingPathComponent("blog")
let post = try await URLSession.shared.load(url: postURL, parse: {
    try JSONDecoder().decode(BlogPost.self, from: $0)
})
print(post)
```

Because a URL and the parse function to decode the data returned from that URL naturally belong together, it makes sense to group them together in a generic Resource struct:

```
struct Resource<A> {
    let url: URL
    let parse: (Data) throws -> A
}
```

Here are the same two endpoints defined as Resources:

```
let profile = Resource<User>(url: profileURL, parse: {
    try JSONDecoder().decode(User.self, from: $0)
})
let post = Resource<BlogPost>(url: postURL, parse: {
    try JSONDecoder().decode(BlogPost.self, from: $0)
})
```

Because the Resource's parse function doesn't know anything about JSON decoding, we can use Resource to represent other kinds of resources as well. For example, we can create resources for loading images or XML data. On the flipside, the added flexibility means that all JSON resources have to repeat the same JSON-decoding code in their parse functions. To avoid this duplication in each JSON resource, we create a convenience initializer that's conditional on A being Decodable. The initializer uses the generic parameter A to decode the correct type:

```
extension Resource where A: Decodable {
    init(json url: URL) {
        self.url = url
        self.parse = { data in
            try JSONDecoder().decode(A.self, from: data)
        }
    }
}
```

This allows us to define the same resources in a much shorter way:

```
let profile = Resource<User>(json: profileURL)
let blogPost = Resource<BlogPost>(json: postURL)
```

Finally, we write a version of the load method that takes a Resource value:

```
extension URLSession {
    func load<A>(_ r: Resource<A>) async throws -> A {
        let (data, _) = try await data(from: r.url)
        return try r.parse(data)
    }
}
```

We can now call load with our profile resource:

```
let user = try await URLSession.shared.load(profile)
print(user)
```

As a nice side benefit of creating the generic Resource type, we made the code easier to test because the logic we'd want to test (the parsing) is now fully synchronous and has no dependencies. The load method on URLSession is still hard to test, because it's asynchronous, and because its dependency on URLSession complicates the testing environment. But this complexity is now isolated in a single method, rather than in multiple places.

Generics Are Statically Dispatched

Swift supports function overloading, i.e. there can be multiple functions with the same name but different argument and/or return types. The compiler's decision-making process of which function to call is called **overload resolution**. For each call site, the compiler follows a set of rules that boil down to "pick the most specific overload for the given input and result types." Overload resolution always happens statically at compile time; dynamic runtime type information plays no role in it.

Here's an example of a simple formatting function with two overloads — a generic variant for all A, and a specific one for Int:

```
// Generic variant for all values.
func format<A>(_ value: A) -> String {
    String(describing: value)
}

// Overload with custom behavior for Ints.
func format(_ value: Int) -> String {
    "+\((value))+"
}
```

Unsurprisingly, when we call format with an Int, Swift calls the specific variant. For any other argument type, it must pick the generic version because that's the only one with a matching type:

```
format("Hello") // Hello
format(42) // +42+
```

Now, let's add another generic function that calls `format` as part of its implementation:

```
func process<B>(_ input: B) -> String {  
    let formatted = format(input)  
    return "-\\"(formatted)-"  
}
```

As expected, calling `process` with a string calls through to our generic `format` overload. But when we call `process` with an `Int`, it also ends up invoking the *less* specific overload of `format`:

```
process("Hello") // -Hello-  
  
// !  
process(42) // -42-
```

Many people find this surprising. Shouldn't the compiler have all the information to see that there's a more specific function for arguments of type `Int`? This may be so in this example where caller and callee are compiled together as part of the same module, but it isn't true generally: if the call to `process(42)` were in a different module than the implementation of `process`, the function would already be compiled and, because overload resolution always happens at compile time, the decision of what overload to call would have already been made.

Swift only uses information that's locally available at the call site for overload resolution, and that includes information about the generic parameters and their constraints. The compiler will never select an overload based on knowledge outside the local scope.

Another way to look at this is that, semantically, there exists only one version of each generic function or type, and this version must be able to handle all valid type arguments. This is markedly different from templates in C++, which compile into separate instantiations for each specific type. We say "semantically" because Swift does in fact compile specialized versions of generic declarations as a performance optimization, but this is an implementation detail and doesn't change the code's behavior.

So what are our options if we want to dispatch the call to `format` dynamically based on the dynamic type of the argument at runtime? One alternative is to manually recover the concrete type with a `dynamic as?` cast inside the calling function:

```
func process2<B>(_ input: B) -> String {
    let formatted: String
    if let int = input as? Int {
        formatted = format(int)
    } else {
        formatted = format(input)
    }
    return "-\\"(formatted)-"
}
```

We've essentially implemented our own dynamic dispatch mechanism here. This works for our example, but it's clumsy and requires manual maintenance as more overloads are added. The better approach is to make the format function a requirement of a protocol and constrain the generic parameters to this protocol. This solution would use dynamic dispatch because protocol requirements *are* dynamically dispatched. We'll say more about this in the next chapter, [Protocols](#).

How Generics Work

How are generics implemented in the compiler? To answer this question, let's take a closer look at the min function from the standard library (we took this example from the [Optimizing Swift Performance](#) session at Apple's 2015 Worldwide Developers Conference):

```
func min<T: Comparable>(_ x: T, _ y: T) -> T {
    return y < x ? y : x
}
```

The only constraints for the arguments and return value of min are that all three must have the same type, T, and that T must conform to Comparable. Other than that, T could be anything — Int, Float, String, or even a type the compiler knows nothing about at compile time because it's defined in another module. This means that the compiler lacks three essential pieces of information it needs to emit code for the function:

- The size of values of type T (including the arguments and return value).
- How to copy and destroy values of type T (e.g. whether they need reference counting).
- The address of the specific overload of the < function to call.

Swift solves these problems by introducing a level of indirection for generic code:

- Function arguments, return values, and variables of unknown size are passed by pointer.
- For every generic type parameter T, the compiler passes the *type metadata* for T into the function. Among other things, the type metadata record contains the type's *value witness table* (VWT). The VWT provides functions for fundamental operations on values of type T, such as copying, moving, or destroying a value. These can be simple no-ops or memcopies for trivial value types such as Int, whereas reference types include their reference counting logic here. The VWT also records the memory layout (size and alignment) of the type.
- For each constraint on T, the compiler passes a *protocol witness table* (PWT) into the function. A PWT is a vtable for a protocol's requirements. It's used to dynamically dispatch function calls to the correct implementations at runtime. In our example, the PWT for "T is Comparable" provides the correct < function.

In pseudocode, the instructions the compiler emits for the min function look something like this:

```
void func min(_ x: OpaquePointer, _ y: OpaquePointer,
    returnValue: OpaquePointer, // caller-allocated storage
    meta_T: TypeMetadata, T_is_Comparable: VTable)
{
    let result = T_is_Comparable.lessThan(y, x, meta_T, T_is_Comparable)
    if result {
        metadata_T.vwt.copyWithInit(returnValue, from: y, meta_T)
    } else {
        metadata_T.vwt.copyWithInit(returnValue, from: x, meta_T)
    }
}
```

Because T's memory layout is unknown at compile time, all values are passed by pointer, including the return value. The compiler goes through the protocol witness table to call y < x, and then through the value witness table to copy y or x into the storage for the return value.

Protocol witness tables provide a mapping between the protocols to which a generic parameter conforms (this is statically known to the compiler through the constraints) and the functions that implement the protocol for the specific type (these are only known at runtime). In fact, the only way to call the protocol's methods is through the

witness table. We couldn't declare the `min` function with an unconstrained parameter `<T>` and then expect it to work with any type that has an implementation for `<`, regardless of Comparable conformance. The compiler wouldn't allow this because there wouldn't be a witness table for it to locate the correct `<` implementation. This is why generics are so closely related to protocols — you can't do much with unconstrained generics except write container types like `Array<Element>` or `Optional<Wrapped>`. We'll revisit this topic in the next chapter in the discussion of protocol witnesses.

The use of pointers to pass generic values around in the pseudocode above might give you the impression that Swift uses *boxing* to represent values of unknown size, but that's not the case. The pointers exist because the compiler can't allocate CPU registers for variable-size values, but the values these pointers point to can still live directly on the stack without additional overhead. This can be much more efficient than boxing, especially when it comes to collections: an array of generic values `[T]` (when `T == Int`) has the exact same runtime memory layout as the concrete array `[Int]`. Generic code interacting with that array uses indirection to manipulate the elements, but the elements themselves are packed tightly and contiguously in memory. This design optimizes for the best possible cache locality in CPU caches.

If you want to learn more about how the generics system works, Swift compiler developers Slava Pestov and John McCall gave a talk on this topic at the 2017 LLVM Developers' Meeting.

Generic Specialization

Compared to non-generic code, Swift's compilation model for generics clearly has a runtime performance cost, caused by the indirection the code has to go through. This is likely negligible when you consider a single function call, but it adds up when generics are as pervasive as they are in Swift. The standard library uses generics everywhere, including for very common operations that must be as fast as possible, such as comparing values.

Luckily, the Swift compiler can make use of an optimization called **generic specialization** — or **monomorphization** — which, in many cases, completely eliminates the overhead. Specialization means that the compiler clones a generic type or function, such as `min<T>`, for a concrete type argument, such as `Int`. This specialized

function can then be optimized specifically for `Int`, removing all indirection. So the specialized version of `min<T>` for `Int` would look like this:

```
func min(_ x: Int, _ y: Int) -> Int {  
    return y < x ? y : x  
}
```

This is exactly the implementation you'd write by hand for a specific, non-generic `min` function. Generic specialization not only saves the cost of the virtual dispatch, but it also enables further optimizations, such as inlining, for which the indirection would otherwise be a barrier.

When you compile your code with optimizations enabled (`swiftc -O` on the command line, or `swift package build -c release` with SwiftPM), the optimizer will create specialized versions of your generic types and functions for each concrete type combination it can see. If your code calls `min` with `Int` and with `Float` arguments, these two specializations will end up in the binary (and the compiler will emit calls to them at the call sites). The unspecialized version, `min<T>`, will also remain available for any other calls where the input type cannot be determined at compile time.

Swift actually makes no guarantees what specializations the compiler will synthesize. In Swift 5.5, the rules are straightforward: specialize for all visible call sites when optimizations are enabled; otherwise, don't specialize anything. In the future, the optimizer may employ heuristics that take into account how often a function is called with a particular set of input types in order to make a more balanced tradeoff between runtime performance, code size, and compilation times.

The catch with specialization is that it only works when the optimizer can see the full definition of the generic type or function at the time it compiles the call site. This is always the case if both caller and callee are in the same file. If they're not, you have two options to help the optimizer:

1. Enable whole module optimization. This is a compilation mode in which all files in the current module are optimized together. Beside generic specialization, whole module optimization enables other important optimizations. For example, the optimizer will recognize when an internal class has no subclasses in the entire module. Since the `internal` modifier makes sure the class isn't visible outside the module, this means the compiler can replace dynamic dispatch with static dispatch for all methods of this class.

2. For clients in other modules, make your generic functions @inlinable. This attribute exports the body of a function as part of a module's interface, making it available to the optimizer when referenced from other modules. In this case, the module containing the generic function is already compiled when the call site is being built, but the optimizer can emit the specialized version of the function into the calling module. Here's our `min` function with the `@inlinable` attribute:

```
@inlinable
func min<T: Comparable>(_ x: T, _ y: T) -> T {
    return y < x ? y : x
}
```

The standard library — as well as other low-level packages like SwiftNIO, Swift Collections, and Swift Algorithms — use `@inlinable` heavily to make their APIs specializable (and to facilitate other optimizations). The experimental `-cross-module-optimization` compiler flag attempts to automate this process by making *all* of a module's public APIs `inlinable` by default. It's not an official feature yet as of Swift 5.5, but there's little downside to trying this out for modules that are statically linked into a single binary. Making something `inlinable` is a big commitment for ABI-stable libraries like the standard library, because it can make it impossible to change implementations or fix bugs. But these concerns don't exist for files that are always (re)compiled together.

Swift's compilation model for generics is somewhat unique in that it bridges the gap between languages like C++ and Rust on the one hand (which specialize everything) and simpler generics models like Java's on the other (where generics are used for type checking but are erased through boxing at runtime). Performance considerations aside, Swift's approach allows for *separate compilation* of generic functions or types and their clients. This is an essential feature for Apple because it allows Apple to ship Swift frameworks in binary form in its SDKs.

Recap

Over the course of this chapter, we saw how generics allow for polymorphism — more specifically, parametric polymorphism. Generics can be used in many places: we can write generic types, generic functions, and generic subscripts. We've seen and used generics throughout the book because the standard library uses them extensively. Likewise, in our own code, we can use generics to abstract away specific details and achieve a clear separation of responsibilities.

Finally, while unconstrained generics are already useful for containers like `Array` and `Optional`, combining generics with protocol constraints opens a whole new level. This brings us to our next chapter, `Protocols`.

Protocols

10

When we work with generic types, we often want to constrain their generic parameters. Protocols allow us to do exactly this. Here are some common examples:

- You can use a protocol to build an algorithm that depends on a type being a number (regardless of the concrete numeric type) or a collection. By programming against a protocol, all conforming types receive the new functionality.
- You can use a protocol to abstract over different “backends” for your code. You’d do this by programming against a protocol, which is then implemented by different types. For example, a drawing program could use a Drawable protocol and be implemented by an SVG renderer and a Core Graphics renderer. Likewise, cross-platform code could use a Platform protocol with specific instances for Linux, macOS, and iOS.
- You can use protocols to make code testable. More specifically, when you write code that uses a protocol rather than a concrete type, you can use different concrete implementations in your production code and in your tests.

A protocol in Swift declares a formal set of *requirements*. For example, the Equatable protocol requires that a conforming type implements the == operator:

```
public protocol Equatable {  
    static func == (lhs: Self, rhs: Self) -> Bool  
}
```

These requirements can consist of methods, initializers, associated types, properties, and inherited protocols. Most protocols also have additional semantic requirements that can’t be expressed in Swift’s type system. For example, the Collection protocol expects that slices use the same index as the original collection to access a particular element. Semantic requirements can include performance promises, like RandomAccessCollection’s guarantee to be able to jump between indices in constant time. A type must not conform to a protocol unless it satisfies the protocol’s semantics. This is important because algorithms written against the protocol rely on these semantic requirements.

Let’s run through a number Swift’s major protocol features. Throughout the chapter, we’ll discuss each of these features in depth.

Protocols in Swift can provide additional functionality beyond their requirements in extensions. The simplest example is Equatable: it requires the == operator to be implemented, but then it adds the != operator, which uses the definition of ==:

```
extension Equatable {
    public static func != (lhs: Self, rhs: Self) -> Bool {
        return !(lhs == rhs)
    }
}
```

Likewise, the `Sequence` protocol has few requirements (you need to provide a way to make an iterator) but adds a lot of methods through extensions.

Only protocol requirements are dynamically dispatched. That is, when you call a requirement on a variable, the decision of which concrete function is invoked is made at runtime based on the variable's dynamic type. Protocol extensions that aren't requirements, however, are always statically dispatched based on the variable's static type. We'll see why this distinction is important in the [Customizing Protocol Extensions](#) section below.

Protocols can also have conditional extensions to add APIs that require additional constraints. For example, `Sequence` has a method, `max()`, that only exists for collections with an `Element` type that conforms to `Comparable`:

```
extension Sequence where Element: Comparable {
    /// Returns the maximum element in the sequence.
    public func max() -> Element? {
        return self.max(by: <)
    }
}
```

Protocols can inherit from other protocols. For example, `Hashable` specifies that any types conforming to it must be `Equatable` as well. Likewise, `RangeReplaceableCollection` inherits from `Collection`, which in turn inherits from `Sequence`. In other words, we can form protocol hierarchies.

Additionally, protocols can be combined. For example, `Codable` is defined as a type alias that combines `Encodable` and `Decodable`. This combination is called a protocol composition.

In some cases, protocol conformances are dependent on other conformances. For example, an array conforms to `Equatable` if and only if its `Element` type conforms to `Equatable`. This is called *conditional conformance*: the conformance of `Array` to `Equatable` is conditional on its elements conforming to `Equatable`:

```
extension Array: Equatable where Element: Equatable { ... }
```

Protocols can declare one or more associated types, i.e. placeholders for related types that are then used to define the other requirements for the protocol. A conforming type must specify a concrete type for each associated type. For example, the Sequence protocol defines an associated type, Element, and every type that conforms to Sequence has defined what its Element type is. String's element type is Character; Data's is UInt8.

Protocol Witnesses

In this section, we show what Swift would look like without protocols. In turn, we hope this example will help you better intuit how protocols work. For example, let's assume we want to write a method on Array to test whether or not all elements are equal. Without the Equatable protocol, we need to explicitly pass a comparison function:

```
extension Array {
    func allEqual(_ compare: (Element, Element) -> Bool) -> Bool {
        guard let f = first else { return true }
        for el in dropFirst() {
            guard compare(f, el) else { return false }
        }
        return true
    }
}
```

To make things a little more formal, we can create a wrapper type around the function to signify its role as an equality function:

```
struct Eq<A> {
    let eq: (A, A) -> Bool
}
```

Now, we can create instances of Eq for comparing values of various concrete types, such as Int. We'll call these values **explicit witnesses** of equality:

```
let eqInt: Eq<Int> = Eq { $0 == $1 }
```

Here's allEqual again, but now using Eq rather than a plain function. Note that we use the generic Element type from Array to make sure all types match up:

```
extension Array {  
    func allEqual(_ compare: Eq<Element>) -> Bool {  
        guard let f = first else { return true }  
        for el in dropFirst() {  
            guard compare.eq(f, el) else { return false }  
        }  
        return true  
    }  
}
```

While the `Eq` type might seem a little obscure, it mirrors how protocols work under the hood: when you add an `Equatable` constraint to a generic type, a **protocol witness** is passed whenever you construct a concrete instance of that type. In the case of `Equatable`, this protocol witness contains the `==` operator to compare two values. The compiler passes this protocol witness automatically based on the concrete type you choose. Here's the version of `allEqual` that uses protocols instead of explicit witnesses:

```
extension Array where Element: Equatable {  
    func allEqual() -> Bool {  
        guard let f = first else { return true }  
        for el in dropFirst() {  
            guard f == el else { return false }  
        }  
        return true  
    }  
}
```

The term *witness* for a value or type that conforms to a protocol (or for a method that satisfies a protocol requirement) comes from logic. You can think of it like this: the type `Int: Equatable` witnesses, through its existence, that it conforms to the protocol (otherwise it wouldn't compile).

We can add an extension to `Eq` as well. Given the ability to tell if two values are equal, we could also write a method that tests two values for inequality:

```
extension Eq {  
    func notEqual(_ l: A, _ r: A) -> Bool {
```

```
    return !eq(l,r)
}
}
```

This is similar to a protocol extension: we make use of the fact that `eq` exists, and we can build more functionality on top of it. Writing the same extension on the `Equatable` protocol looks almost identical. Rather than using `A`, we use `Self`, which, in protocols, is a placeholder for the conforming type:

```
extension Equatable {
    static func notEqual(_ l: Self, _ r: Self) -> Bool {
        return !(l == r)
    }
}
```

This is exactly how the standard library implements the `!=` operator for `Equatable` types.

Conditional Conformance

To write a version of `Eq` for arrays, we need a way to compare two elements. We can write our `eqArray` as a function and pass in the explicit witness as a value:

```
func eqArray<El>(_ eqElement: Eq<El>) -> Eq<[El]> {
    return Eq { arr1, arr2 in
        guard arr1.count == arr2.count else { return false }
        for (l, r) in zip(arr1, arr2) {
            guard eqElement.eq(l, r) else { return false }
        }
        return true
    }
}
```

Again, this corresponds directly with conditional conformance in Swift. For example, this is how the standard library conforms `Array` to `Equatable`:

```
extension Array: Equatable where Element: Equatable {
    static func ==(lhs: [Element], rhs: [Element]) -> Bool {
        ...
    }
}
```

Adding the constraint that Element must conform to Equatable is equivalent to adding the parameter `eqElement` to the `eqArray` function above. Within the extension, we can now make use of the `==` operator to compare two elements. The big difference is that with protocols, the protocol witness is passed automatically.

Protocol Inheritance

Swift also supports protocol inheritance. For example, the `Comparable` protocol requires any conforming types to conform to `Equatable` as well. This is also called **refining**. `Comparable` refines `Equatable`:

```
public protocol Comparable: Equatable {
    static func <(lhs: Self, rhs: Self) -> Bool
    // ...
}
```

In our imaginary version of Swift without protocols, we can also express this concept by creating an explicit witness type for `Comparable` and including `Equatable` inside it, along with the `lessThan` function:

```
struct Comp<A> {
    let equatable: Eq<A>
    let lessThan: (A, A) -> Bool
}
```

Again, this is similar to how protocol witnesses work when a protocol inherits from other protocols. In an extension of `Comp`, we can now use both `Eq` and `lessThan`:

```
extension Comp {
    func greaterThanOrEqual(_ l: A, _ r: A) -> Bool {
        return lessThan(r, l) || equatable.eq(l, r)
    }
}
```

The concept of passing explicit witnesses is a useful mental model for understanding what the compiler does internally. We find it particularly helpful when we get stuck trying to solve a problem with protocols.

However, keep in mind the two approaches aren't completely equivalent. While we can create an unlimited number of explicit witness values with the same type, a type can conform to a protocol at most once. And unlike an explicit witness, these conformances are passed around automatically.

If multiple conformances per type were allowed, the compiler would have to come up with a way to pick the most appropriate conformance. Combined with features like conditional conformance, this gets complicated quickly. So to avoid this complexity, Swift doesn't allow multiple conformances.

Designing with Protocols

In this section, we'll look at an example of a drawing protocol that's implemented by two concrete types: we can render a drawing either as an SVG or into a graphics context of Apple's Core Graphics framework. Let's start by defining the protocol with requirements for drawing an ellipse and a rectangle:

```
protocol DrawingContext {  
    mutating func addEllipse(rect: CGRect, fill: UIColor)  
    mutating func addRectangle(rect: CGRect, fill: UIColor)  
}
```

Conforming CGContext is straightforward; we set the fill color and then fill the specified shape:

```
extension CGContext: DrawingContext {  
    func addEllipse(rect: CGRect, fill fillColor: UIColor) {  
        setFillColor(fillColor.cgColor)  
        fillEllipse(in: rect)  
    }  
  
    func addRectangle(rect: CGRect, fill fillColor: UIColor) {  
        setFillColor(fillColor.cgColor)  
        fill(rect)  
    }  
}
```

Likewise, conforming our SVG type isn't very complicated. We convert the rectangle into a set of XML attributes and convert the UIColor into a hex string (e.g. a white color becomes #ffffff):

```
extension SVG: DrawingContext {  
    mutating func addEllipse(rect: CGRect, fill: UIColor) {  
        var attributes: [String:String] = rect.svgEllipseAttributes  
        attributes["fill"] = String(hexColor: fill)  
        append(Node(tag: "ellipse", attributes: attributes))  
    }  
  
    mutating func addRectangle(rect: CGRect, fill: UIColor) {  
        var attributes: [String:String] = rect.svgAttributes  
        attributes["fill"] = String(hexColor: fill)  
        append(Node(tag: "rect", attributes: attributes))  
    }  
}
```

(The definitions of SVG, CGRect.svgAttributes, CGRect.svgEllipseAttributes, and String.init(hexColor:) aren't shown here; they're not important for the example.)

Protocol Extensions

One of the key features of Swift protocols is **protocol extensions**. Once we know how to draw ellipses, we can also add an extension to draw circles around a center point. We do this as an extension of DrawingContext:

```
extension DrawingContext {  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {  
        let diameter = radius * 2  
        let origin = CGPoint(x: center.x - radius, y: center.y - radius)  
        let size = CGSize(width: diameter, height: diameter)  
        let rect = CGRect(origin: origin, size: size)  
        addEllipse(rect: rect.integral, fill: fill)  
    }  
}
```

To use the above, we can write another extension on DrawingContext that renders a blue circle on top of a larger yellow square:

```
extension DrawingContext {
    mutating func drawSomething() {
        let rect = CGRect(x: 0, y: 0, width: 100, height: 100)
        addRectangle(rect: rect, fill: .yellow)
        let center = CGPoint(x: rect.midX, y: rect.midY)
        addCircle(center: center, radius: 25, fill: .blue)
    }
}
```

By writing the method as an extension of `DrawingContext`, we can now call it both on `SVG` and on `CGContext` instances. This is a technique used throughout the Swift standard library: while conforming to a protocol requires implementing a few methods, you then receive a lot more functionality “for free” through extensions on that protocol.

Customizing Protocol Extensions

Adding a method as a protocol extension does *not* make it part of the protocol’s requirements. This can lead to tricky behavior because only requirements are dispatched dynamically. Let’s see what we mean by that. Going back to our example, we’d like to make use of SVG’s `circle` element, i.e. a circle should be rendered as a `<circle>` and not as an `<ellipse>`. Let’s add the specialized `addCircle` method to our `SVG` implementation:

```
extension SVG {
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {
        let attributes = [
            "cx": "\(center.x)",
            "cy": "\(center.y)",
            "r": "\(radius)",
            "fill": String(hexColor: fill),
        ]
        append(Node(tag: "circle", attributes: attributes))
    }
}
```

When we create a variable of type `SVG` and call `addCircle` on it, the above method will get called:

```
var circle = SVG()
circle.addCircle(center: .zero, radius: 20, fill: .red)
```

```
circle
/*
<svg>
<circle cx="0.0" cy="0.0" fill="#ff0000" r="20.0"/>
</svg>
*/
```

No surprise there. But when we call `drawSomething()` on `Drawing` (which contains a call to `addCircle`), the above method won't get called. Notice that the resulting SVG syntax contains an `<ellipse>` tag instead of the `<circle>` tag we intended to have:

```
var drawing = SVG()
drawing.drawSomething()
drawing
/*
<svg>
<rect fill="#ffff00" height="100.0" width="100.0" x="0.0" y="0.0"/>
<ellipse cx="50.0" cy="50.0" fill="#0000ff" rx="25.0" ry="25.0"/>
</svg>
*/
```

This behavior can be surprising. To understand what's going on, we'll first rewrite our `drawSomething` extension as a free function with explicit generics. This has exactly the same semantics as the protocol extension we wrote before:

```
func drawSomething<D: DrawingContext>(context: inout D) {
    let rect = CGRect(x: 0, y: 0, width: 100, height: 100)
    context.addRectangle(rect: rect, fill: .yellow)
    let center = CGPoint(x: rect.midX, y: rect.midY)
    context.addCircle(center: center, radius: 25, fill: .blue)
}
```

Here, the generic parameter `D` is constrained to `DrawingContext`. This means that the compiler will pass a protocol witness for the `DrawingContext` protocol when we call it. The witness contains only the protocol's requirements, i.e. the `addRectangle` and `addEllipse` methods. Since the `addCircle` method is only defined in an extension and not in the protocol, it's not included in the witness.

The crux is that only methods that are part of the witness can be dynamically dispatched to a specialized concrete implementation in the conforming type, because the witness is the only dynamic information that's available at runtime. Calls to non-requirements *in*

generic contexts are always dispatched statically to the implementation defined in the protocol extension.

As a result, when we call `addCircle` from within the `drawSomething` function, it's statically dispatched to the extension on our protocol. The compiler simply can't generate the code that would be necessary to dynamically dispatch the call to our extension on SVG.

To get dynamic dispatch behavior, we should make `addCircle` a requirement of the protocol:

```
protocol DrawingContext {  
    mutating func addEllipse(rect: CGRect, fill: UIColor)  
    mutating func addRectangle(rect: CGRect, fill: UIColor)  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor)  
}
```

The existing `addCircle` implementation in the protocol extension now becomes a *default implementation* of the requirement. Because the default implementation exists, we don't have to do anything else to make our code compile. Now that `addCircle` is part of the protocol, it's also part of the protocol witnesses, and when we call `drawSomething` with an SVG value, the correct custom implementation will be called:

```
var drawing2 = SVG()  
drawing2.drawSomething()  
drawing2  
/*  
<svg>  
<rect fill="#ffff00" height="100.0" width="100.0" x="0.0" y="0.0"/>  
<circle cx="50.0" cy="50.0" fill="#0000ff" r="25.0"/>  
</svg>  
*/
```

A protocol method with a default implementation is sometimes called a **customization point** in the Swift community. Conforming types receive a default implementation, but they can override it if they choose to. The standard library uses customization points extensively to provide shared defaults that can be overridden, often for performance reasons. An example is the `distance(from:to:)` method for computing the distance between two collection indices. The default implementation is an $O(n)$ operation since it iterates over all of the collection's indices. Because `distance(from:to:)` is a

customization point, collection types that can provide a more efficient implementation, such as `Array`, are able to override the default implementation.

Protocol Composition

Protocols can be combined together into a protocol composition. An example from the standard library is `Codable`; it's a type alias for `Encodable` & `Decodable`:

```
typealias Codable = Decodable & Encodable
```

This means we can write the following function, and within its body, we can make use of both protocols when working with value:

```
func use Codable<C: Codable>(value: C) {  
    // ...  
}
```

In our drawing example, we might want to render some attributed strings (these are strings in which subranges are attributed with formatting, such as emphasis, fonts, and colors). However, the SVG file format doesn't support attributed strings natively (but Core Graphics does). Rather than adding a method to our `DrawingContext` protocol, we'll create a separate protocol:

```
protocol AttributedDrawingContext {  
    mutating func draw(_ str: AttributedString, at point: CGPoint)  
}
```

This way, we can conform `CGContext` to the new protocol, but we don't have to add SVG support. We can then combine the two protocols — for example, with a `DrawingContext` extension that's constrained to types that also conform to `AttributedDrawingContext`:

```
extension DrawingContext where Self: AttributedDrawingContext {  
    mutating func drawSomething2() {  
        let size = CGSize(width: 200, height: 100)  
        addRectangle(rect: .init(origin: .zero, size: size), fill: .red)  
        var text = AttributedString("Hello")  
        text.font = UIFont.systemFont(ofSize: 48)  
        draw(text, at: CGPoint(x: 20, y: 20))  
    }  
}
```

Alternatively, we could've written a function with explicit generics. Like before, this is semantically equivalent to the method above:

```
func drawSomething2<C: DrawingContext & AttributedDrawingContext>(  
    _ c: inout C)  
{  
    // ...  
}
```

Protocol composition can be very powerful as a way to opt in to features that might not be implemented by every conforming type.

Protocol Inheritance

Rather than composing our protocol when we use it, we can also let protocols inherit from each other. For example, we could've written our definition of AttributedDrawingContext like this:

```
protocol AttributedDrawingContext: DrawingContext {  
    mutating func draw(_ str: AttributedString, at point: CGPoint)  
}
```

This definition requires that any type that conforms to AttributedDrawingContext should also conform to DrawingContext.

Both protocol inheritance and protocol composition have their use cases. For example, the Comparable protocol inherits from Equatable. This means it can add definitions such as `>=` and `<=` while only requiring conforming types to implement `<`. In the case of Codable, it wouldn't have made sense to let Encodable inherit from Decodable, or the other way around. However, it would've made sense to define a new protocol named Codable that inherits from both Encodable and Decodable. In fact, writing `typealias Codable = Encodable & Decodable` is semantically equivalent to writing `protocol Codable: Encodable, Decodable {}`. The type alias is a little more lightweight and makes it clear that Codable is *only* the composition of the two protocols and doesn't add any functionality of its own.

Associated Types

Some protocols require more than just methods, properties, and initializers: they need one or more related types to be useful. This can be achieved using associated types.

While we don't find ourselves writing protocols with associated types in our own code very often, the standard library uses associated types a lot. One of the smallest examples from the standard library is the `IteratorProtocol` protocol. It has an associated type for the elements that are iterated over, along with a single method to get the next element:

```
protocol IteratorProtocol {  
    associatedtype Element  
    mutating func next() -> Element?  
}
```

Note that the `next` requirement uses the associated type in its return type. When you write a conformance to the protocol, you specify the specific type to be used for `Element`. The following example defines an iterator of `Int` elements:

```
struct Counter: IteratorProtocol {  
    typealias Element = Int  
    func next() -> Int? { ... }  
}
```

The type alias sets the associated type explicitly. We could leave it out and the compiler will infer the associated type from the return type of the `next` method. This small example shows that associated types allow the author of a protocol to define the protocol's requirements in terms of one or more related types. The concrete types to be used don't need to be known until the protocol is implemented.

Associated types can have default values, written with an equals sign after the associated type declaration. The `Collection` protocol has five associated types, and many of these have default values. For example, the `SubSequence` associated type has a default value of `Slice<Self>`:

```
public protocol Collection: Sequence {  
    associatedtype Iterator = IndexingIterator<Self>  
    associatedtype SubSequence: Collection = Slice<Self>  
    ...  
}
```

An associated type with a default value becomes another customization point, just like a method with a default implementation: when conforming a custom type to `Collection`, you can stick with the default and save some implementation work as a result. However, some collections override their slice type, often for performance or convenience (for example, `String` uses `Substring` as the `SubSequence`). We discuss all associated types of `Collection` in depth in the [Collection Protocols](#) chapter.

Associated types are similar to generic parameters in that they're placeholders that get filled in at a later time. What differentiates them from generic types is who gets to fill these placeholders in, and where. Generic parameters are provided by *users* of a type when they want to use that type in their code. Associated types are determined by the *author* of a type (or someone extending an existing type) when they implement the protocol. For instance, users of an array can choose the array's element type (e.g. `Array<Int>` vs. `Array<String>`) because the standard library authors exposed that as a generic parameter. But users *can't* customize an array's iterator or subsequence type; those have been fixed by its `Collection` conformance in the standard library.

Self

Inside a protocol definition or extension, `Self` refers to the conforming type, i.e. the type that's implementing the protocol. You can think of `Self` as an implicit associated type that's always available. We saw earlier how `Equatable` uses `Self` to define the `==` operator. Here's another example, from the `BinaryInteger` protocol:

```
public protocol BinaryInteger: ... {
    func quotientAndRemainder(dividingBy rhs: Self)
        -> (quotient: Self, remainder: Self)
}
```

This declaration establishes a same-type relationship between both the type that's implementing the protocol and the argument, and the type and the two return values.

Example: State Restoration

For the main example in this section, we reimplement a small version of UIKit's state restoration mechanism by using a protocol with an associated type. In UIKit, state restoration takes a hierarchy of view controllers and views and serializes their state when an app is suspended. During the next launch of the app, UIKit attempts to restore the application's state.

Instead of a class hierarchy, we'll use protocols to indicate view controllers. In a real implementation, the `ViewController` protocol would have many methods, but for the sake of simplicity, we'll make it an empty protocol:

```
protocol ViewController {}
```

To restore a specific view controller, we need to be able to read and write its state. We want this state to conform to `Codable` so that we can encode and decode it. Because the type of the state depends on the specific view controller, we model it as an associated type:

```
protocol Restorable {
    associatedtype State: Codable
    var state: State { get set }
}
```

As an example, we could create a view controller to display messages. The view controller's state consists of an array of messages and the current scroll position. We model the state as a nested struct and make it conform to `Codable`:

```
class MessagesVC: ViewController, Restorable {
    typealias State = MessagesState

    struct MessagesState: Codable {
        var messages: [String] = []
        var scrollPosition: CGFloat = 0
    }
    var state: MessagesState = MessagesState()
}
```

Note that we don't have to declare the `typealias State` in the conformance; the compiler is smart enough infer it by looking at the type of the `state` property. We could also rename our `MessagesState` to `State`, and everything will keep working.

Conditional Conformance with Associated Types

Some types only conform to a protocol when certain conditions hold. As we saw in the section on conditional conformance, `Array` conforms to `Equatable` if and only if its elements conform to `Equatable`. Conditions can use information about associated types as well. For example, `Range` has a generic parameter, `Bound`. `Range` conforms to

Sequence if and only if Bound is Strideable and the Bound's Stride (an associated type of Strideable) conforms to SignedInteger:

```
extension Range: Sequence
  where Bound: Strideable, Bound.Stride: SignedInteger
```

Note that constraints this complex are the exception, even in the standard library.

In our hypothetical UI framework, we also have split view controllers, which are generic over their two child view controllers:

```
class SplitViewController<Master: ViewController, Detail: ViewController> {
  var master: Master
  var detail: Detail
  init(master: Master, detail: Detail) {
    self.master = master
    self.detail = detail
  }
}
```

Assuming the split view controller doesn't have state of its own, we can combine the state of both child view controllers. Ideally, we'd like to write

var state: (Master.State, Detail.State), but alas, tuples don't conform to Codable — not even conditionally. (In fact, tuples can't conform to any protocol; [a proposal](#) to make all tuples automatically Equatable, Hashable, and Comparable was accepted but isn't yet implemented.) Instead, we have to write our own generic Pair struct. We can then conditionally make it conform to Codable:

```
struct Pair<A, B>: Codable where A: Codable, B: Codable {
  var left: A
  var right: B
  init(_ left: A, _ right: B) {
    self.left = left
    self.right = right
  }
}
```

Finally, to make SplitViewController conform to Restorable, we have to require that both Master and Detail are Restorable too. Rather than keeping the state locally in our SplitViewController, we compute it from the two view controllers, and rather than setting a local variable, we immediately forward mutations to the two children:

```
extension SplitViewController: Restorable
    where Master: Restorable, Detail: Restorable
{
    var state: Pair<Master.State, Detail.State> {
        get {
            return Pair(master.state, detail.state)
        }
        set {
            master.state = newValue.left
            detail.state = newValue.right
        }
    }
}
```

As we noted in the section on conditional conformance, any type can conform to a protocol at most once. This means we can't have any additional conformances for when Master is Restorable but Detail isn't, or vice versa.

Retroactive Conformance

One of the major features of Swift's protocols is that it's possible to conform types to a protocol in retrospect. For example, above we conformed CGContext to our Drawable protocol. This allows programmers to extend the functionality of types that are defined in other modules and make those types available to algorithms that use the protocol as a constraint.

However, it's possible to take this freedom too far. When conforming a type to a protocol, we should always make sure that we're either the owner of the type or the owner of the protocol (or both). Conforming a type you don't own to a protocol you don't own isn't recommended.

For example, at the moment of writing, CLLocationCoordinate2D from the Core Location framework doesn't conform to the Codable protocol. Even though it's easy to add the conformance ourselves, our implementation might break if and when Apple decides to conform CLLocationCoordinate2D to Codable. In such a case, Apple might choose a different implementation, and as a result, we'd no longer be able to deserialize existing file formats.

Conformance collisions can also happen when two separate packages conform a type to the same protocol. This problem occurred when both SourceKit-LSP and SwiftPM conformed Range to Codable, both with different constraints. (In Swift 5, the standard library added Codable conformance for Range.)

As a solution to these potential problems, we can often use wrapper types and add the conditional conformance there. For example, we could make a wrapper struct around CLLocationCoordinate2D and make the wrapper conform to Codable. We'll see an example of this in the Encoding and Decoding chapter.

Existentials

Strictly speaking, protocols cannot be used as concrete types in Swift; they can only be used to constrain generic parameters. Yet the following code compiles just fine (we're using the DrawingContext protocol from above as an example):

```
let context: DrawingContext = SVG()
```

When we're using a protocol like a concrete type, the compiler creates a wrapper type for the protocol, called an *existential*, behind the scenes. Like *witness* and much of type theory in general, the term comes from logic: the DrawingContext asserts that there exists some type that satisfies its requirements. (In contrast, generics create "for all" relationships: the generic type `Array<Element>` asserts that *for all* types Element, there's a corresponding type, `Array<Element>`.)

The fact that Swift has been using the same syntax for the protocol (constraint) and the existential type has contributed to the general confusion among Swift programmers about the distinction between the two, especially because existentials have some unintuitive limitations, as we'll see in the following sections. This violates a fundamental rule of good API and language design — namely that the same or similar concepts should be spelled the same, and different things should be spelled differently. It's also unfortunate that the lightweight syntax might wrongly attract developers to a feature (existentials) whose use is generally discouraged in favor of its syntax-heavier alternatives (generics and opaque types) unless you specifically need the flexibility.

Accordingly, Swift 5.6 introduced a new any P syntax for existentials. The old spelling will remain valid for now, but it's slated for deprecation or removal in a future language version. We'll use the any P syntax in the remainder of this chapter:

```
let context: any DrawingContext = SVG()
```

We can think of any DrawingContext as an alternative spelling for something like Any<DrawingContext> (if Any were a generic type), i.e. an Any value with an additional constraint. When the compiler sees any DrawingContext, it creates an Any box (which is four words long, or 32 bytes on 64-bit platforms), and it adds a one-word protocol witness for each protocol. We can verify it for the example above like this:

```
MemoryLayout<Any>.size // 32  
MemoryLayout<any DrawingContext>.size // 40
```

This box for values of a protocol type is also called an *existential container*. The compiler has to create these containers because it needs to know the size of the type at compile time. Since different types with different sizes can conform to a protocol, wrapping the protocol(s) up in an existential container creates a type with constant size for the compiler to lay values out in memory. Three of the four words used by the Any container are used to store small values directly inline. If the boxed value is larger than three words, the compiler will store it on the heap and put a pointer into the box. The fourth word stores a pointer to the boxed type's type metadata record. The type metadata contains the value witness table, which provides functions for basic operations, such as creating, destroying, or copying a value.

We can see that the size of the existential container increases with the number of protocols we use. For example, Codable is shorthand for both Encodable and Decodable, so we'd expect the size of the any Codable existential to be 32 bytes for the Any container, plus two times 8 bytes for the protocol witnesses:

```
MemoryLayout<any Codable>.size // 48
```

When we create an array of any Codables, the compiler knows that each element is 48 bytes in size, no matter which concrete type we use. For example, for an array with three elements, 144 bytes have to be allocated:

```
let codables: [any Codable] = [Int(42), Double(42), "fourtytwo"]
```

The only thing we can do with the elements of the codables array (short of runtime casting using as, as?, or is) is to use the Encodable and Decodable APIs, since the concrete types of the elements are hidden by the existential container. The container is mostly invisible to the programmer. For example, calling type(of:) will return the type of the boxed value, and not the type of the box itself:

```
type(of: codables[0]) // Int
```

Existentials vs. Generics

Sometimes existentials can be used interchangeably with constrained generic parameters. Consider the following two functions:

```
func encode1(x: any Encodable) { /* ... */}  
func encode2<E: Encodable>(x: E) { /* ... */}
```

While we can call both functions with any type that conforms to `Encodable`, they aren't equivalent. In the case of `encode1`, the compiler will wrap the parameter in an `any Encodable` existential container. This wrapping costs some performance and possibly requires an extra memory allocation call if the wrapped value is too big to fit into the existential directly. Perhaps most importantly, the existential prevents additional optimizations because all method calls on the wrapped value have to go through the existential's witness table.

For the generic function, on the other hand, the compiler can generate specialized versions for some or all of the types `encode2` is called with. These specializations achieve the same performance as manually writing a specific function for every concrete type. The downsides compared to the version taking an existential are longer compile times and a larger binary size. Refer to the [Generics](#) chapter for more on generic specialization.

In most code, the overhead of existential containers isn't a problem, but it can be useful to keep in mind when optimizing performance-critical code. If you called one of the `encode` functions in a loop thousands of times, you'd probably see that `encode2` would be significantly faster. (In simple cases, the optimizer can actually rewrite an existential-taking function into the equivalent generic function, but the general point still stands.)

Existentials and Associated Types

In Swift 5.6, existentials are limited to protocols that have neither associated types nor requirements that reference `Self` (except when `Self` is used as a return type). This is a longstanding limitation that many Swift developers recognize by its compiler diagnostic: "Protocol 'P' can only be used as a generic constraint because it has `Self` or associated

type requirements.” The original reason for the error was a missing feature in the compiler implementation that has since been fixed. Consequently, Swift 5.7 will lift the restriction and allow any protocol to be used as an existential.

But there’s a more fundamental limitation here that the best compiler can’t lift: certain protocol requirements are inherently incompatible with existentials. Consider this example:

```
let a: any Equatable = "Alice" // Error in Swift 5.5, legal in Swift 5.7
let b: any Equatable = "Bob"
a == b
// Swift 5.7 error: binary operator '==' cannot be applied
// to two 'Equatable' operands.
```

The code above still won’t compile in Swift 5.7 because the `==` operator references `Self` in its parameters:

```
public protocol Equatable {
    static func == (lhs: Self, rhs: Self) -> Bool
}
```

The `==` function expects two arguments that are exactly the same type. Existentials cannot satisfy this because they erase type information, so the compiler no longer knows that `a` and `b` are in fact the same type. This example illustrates that the “`Self` or associated type” restriction does not (and cannot) disappear fully — rather, it gets shifted down from the protocol to the level of the individual protocol requirements. Therefore, certain requirements won’t be available on values of existential type.

Interestingly, this does *not* apply to requirements that use `Self` or an associated type in a covariant position (for example, in the return type). This code using a `FloatingPoint` existential will be legal in Swift 5.7:

```
let number: any FloatingPoint = 1.0
print(number.nextUp) // 1.0000000000000002
```

Even though `FloatingPoint.nextUp` returns `Self`, it’s callable on an existential. This is because when `Self` is used as a return type, the compiler knows that it can box the result up again in another existential container. The static type of the expression `number.nextUp` is any `FloatingPoint`.

The same rules we saw for `Self` apply to requirements that reference associated types. Functions that use associated types in parameters will continue to be unavailable on existentials:

```
let anyCollection: any Collection = [1, 2, 3] // Legal in Swift 5.7
anyCollection.index(after: 0)
// Swift 5.7 error: member 'index' cannot be used on value of protocol
// type 'Collection'; use a generic constraint instead
```

Meanwhile, requirements that *return* an associated type will become available:

```
anyCollection.first // 1 (type is Optional<Any>)
```

The return type of `Collection.first` is `Optional<Element>`. Since the existential erases the concrete element type, the compiler replaces it with `Any`. The static type of `anyCollection.first` becomes `Optional<Any>`.

This works because the compiler considers `Optional<T>` to be a subtype of `Optional<Any>`. The compiler has special knowledge of how to unwrap an `Optional<T>` and rebox the wrapped value in an `Optional<Any>`. In other words, `Optional` is covariant in its generic type. Covariance is hardcoded into the compiler, and `Optional` is one of the few types that supports it. The others are `Array` and `Dictionary`, the latter for its `Value` type. Tuples are also considered covariant in either of their element types. Swift doesn't support general covariance relationships for arbitrary generic types, i.e. `SomeStruct<T>` is *not* a subtype of `SomeStruct<Any>`. Allowing this in the general case would make the type system unsound.

What if we wanted to preserve the concrete element type of an existential collection? It'd be convenient if Swift supported where constraints on existentials, like so:

```
let anyIntCollection: any Collection where Element == Int // Not legal syntax
```

This isn't supported yet, but there's a good chance something like it will be in the future.

In Swift 5.7, we'll be able to achieve a similar effect with a workaround. We can introduce a new protocol that inherits from the base protocol and adds a constraint for the associated type we want to fix:

```
protocol IntCollection: Collection where Element == Int {}
extension Array: IntCollection where Element == Int {}
```

```
let anyIntCollection: any IntCollection = [1, 2, 3]
anyIntCollection.first // 1 (type is Optional<Int>)
```

We effectively moved the `where` constraint on the existential to a new protocol that encodes the same constraint. This gives the compiler enough information to know that the static type of any protocol API that uses `Element` must be `Int`.

Existentials Don't Conform to Protocols

“Protocols don’t conform to themselves” is a well-known dictum in the Swift community. Now that we’ve seen that protocols and the implicit compiler-generated protocol types (existentials) are different things, we can phrase this more precisely: an existential type doesn’t conform to “its” protocol of the same name. In other words, you can’t pass an existential `any P` to a generic function `func f<T: P>(_ x: P)`.

The fundamental reason for this restriction is the same as in the previous section: not all requirements are available on existentials. In addition to requirements that reference `Self` or associated types, this also applies to initializers and static methods, which can both be viewed as special cases of functions that take `Self` as their first argument.

Take the `Decodable` protocol as an example. The `JSONDecoder` type provides this method for decoding a JSON value:

```
extension JSONDecoder {
    func decode<T: Decodable>(_ type: T.Type, from data: Data) throws -> T
}
```

The `decode` method will ultimately call `T.init(from:)`, which is the initializer requirement that’s part of the `Decodable` protocol. If it were permitted to pass the existential type `(any Decodable).self` to `decode`, the compiler wouldn’t know what type to instantiate. It makes sense that this is illegal:

```
let json = #"{ "email": "alice@example.com" }#
let jsonData = Data(json.utf8) // 32 bytes
let decoded = try JSONDecoder().decode((any Decodable).self, from: jsonData)
// Error: protocol 'Decodable' as a type cannot conform to the protocol itself.
```

Unlike the limitation on variables of protocol types, which we've seen move down from the entire protocol to individual requirements, the restriction on "self-conformance" of existentials still applies universally to all protocols — with the exception of the `Error` protocol.

The existential type `Error` *does* conform to the `Error` protocol. This special case is hardcoded into the compiler to allow any `Error` (the existential) to be used for generic parameters that are constrained to `Error` (the protocol). Without it, very common types, such as `Result<Int, any Error>` (written as `Result<Int, Error>` before Swift 5.6), would be illegal — you'd have to specify a concrete error type, such as `Result<Int, URLError>`. This compiler magic is possible because the `Error` protocol is a pure marker protocol without any requirements, so there's no problem with requirements being unavailable.

Don't Use Existentials Prematurely

As a rule, you should prefer generics over working with protocol types unless you need the additional flexibility the boxing provides, e.g. to store heterogeneous values in a collection. Existentials erase type information while generics preserve it. Constant boxing in and unboxing from existential containers is bad for performance in and of itself; additionally, the premature type erasure will inevitably block some compiler optimizations.

Furthermore, we've seen that even as future Swift releases will lift more restrictions that currently still exist for protocol types, there are fundamental limitations that make some APIs permanently unavailable on values of existential type. Unlocking existentials for all protocols will eliminate a source of confusion nearly all Swift developers stumble over. We think this is a good thing, even though it unavoidably introduces new sources of perplexing diagnostics (e.g. why can I call this protocol method but not that one?). There's a real risk that inexperienced developers start out with existentials everywhere — not least because of the convenient syntax — and then find out hours or days later that they should have used generics with protocol constraints instead. The fear of inviting footguns like this was a big factor in the Swift team's hesitation to lift the restrictions on existentials.

One of the main reasons to use a statically typed language with a type system that's as strong as Swift's is to give the compiler as much information as possible to perform its tasks. Unnecessary type erasure works against this.

Opaque Types

Opaque types are a way for the author of an API to *hide* the concrete return type of that API without having to resort to existentials (which would *erase* the type information). This is achieved through the `some MyProtocol` syntax, which stands for “some concrete type that satisfies the listed constraint.” The underlying concrete type is hidden from clients; they can only manipulate it through the declared capabilities (here, `MyProtocol`). So far, this sounds like an existential, but unlike existentials, the type system preserves an opaque type’s (hidden) type identity. This allows clients to do some things with opaque types that would be unavailable on existentials, such as calling APIs with `Self` or associated type constraints. Moreover, the opaque type `some MyProtocol` *does* conform to the protocol, whereas we’ve seen that an existential does not. Opaque types are also generally more efficient and easier for the compiler to optimize than existentials.

Information Hiding

Apple uses opaque types extensively in SwiftUI. Without them, writing and reading SwiftUI code would be so inconvenient that SwiftUI’s API design would likely look very different. The information-hiding aspect of opaque types has two sides to it: (a) making deeply nested generics easier to use, and (b) hiding implementation details. We’ll take a closer look at both by way of a larger example.

Let’s build a small Markdown-inspired markup syntax for text formatting. We want to define an API for applying formatting attributes to text and generating the resulting markup. We start by defining a protocol to represent rich text that can be rendered as markup:

```
protocol RichText {  
    func render() -> String  
}
```

We’ll only add support for **bold** and *italic* text here, but starting with a protocol allows users to extend the system with their own markup syntax.

For plain, unformatted text, we can make `String` conform to `RichText`:

```
extension String: RichText {  
    func render() -> String { self }  
}
```

Next, let's create a type to represent bold text. This struct stores a RichText value and applies the correct markup for rendering:

```
struct Bold<Text: RichText>: RichText {  
    var text: Text  
  
    func render() -> String {  
        "***\\(text.render())***"  
    }  
}
```

Note that we chose to make the struct generic over its input text. This isn't the only option (we could've used an existential instead), but it seems appropriate given the general rule of avoiding premature type erasure.

The type for representing italic text looks almost identical:

```
struct Italic<Text: RichText>: RichText {  
    var text: Text  
  
    func render() -> String {  
        "_\\(text.render())_"  
    }  
}
```

And here's a usage example of what we have so far:

```
Bold(text: "bold").render() // **bold**  
Bold(text: Italic(text: "bold and italic")).render() // **_bold and italic_**
```

To model text with varying formatting, we'll need a way to combine multiple text fragments. We can define a struct, Concat, that concatenates two RichText values:

```
struct Concat<Text1: RichText, Text2: RichText>: RichText {  
    var a: Text1  
    var b: Text2  
  
    func render() -> String {  
        "\\(a.render())\\(b.render())"  
    }  
}
```

This works, but the syntax is unwieldy: concatenating more than two fragments requires nesting. Adding some SwiftUI-style “modifiers” to the protocol gives us a more fluent API:

```
extension RichText {
    func bold() -> Bold<Self> {
        Bold(text: self)
    }

    func italic() -> Italic<Self> {
        Italic(text: self)
    }

    func appending<Other: RichText>(_ other: Other) -> Concat<Self, Other> {
        Concat(a: self, b: other)
    }
}
```

Here's the new API in use:

```
let text = "Hello,"
    .bold()
    .appending(" ")
    .appending(
        "world!".italic()
    )
text.render() // **Hello,** _world!
```

Nice! If you've seen SwiftUI code, this should look very familiar. What's not so nice, however, is the type of the `text` value, `Concat<Concat<Bold<String>, String>, Italic<String>>`. This illustrates two common problems with deeply nested generic types:

0. **Nested generics quickly become unmaintainable.** Type inference can help with this somewhat, but function declarations require explicit return types. Imagine a function that constructs some formatted text and returns it. Not only would you have to write out the full return type in the function's signature, making it hard to parse for users, but you'd also have to update this type every time you changed the structure of the returned value (e.g. by appending another text fragment).

1. **The type exposes implementation details.** Once you publish a type signature as a public API, it may be impossible to change it without breaking clients. A library that forced its users to update their code every time it updated some text formatting wouldn't be very user-friendly. Moreover, a text markup library may not want to leak out types like `Bold` or `Concat` at all if callers are only supposed to interact with the `RichText` interface.

Opaque types solve these problems by letting us replace a concrete return type, such as `Concat<Self, Other>`, with `some RichText`. Let's do this for our text modifiers (the implementations are unchanged):

```
extension RichText {  
    func bold() -> some RichText {  
        Bold(text: self)  
    }  
  
    func italic() -> some RichText {  
        Italic(text: self)  
    }  
  
    func appending<Other: RichText>(_ other: Other) -> some RichText {  
        Concat(a: self, b: other)  
    }  
}
```

The call site remains unchanged:

```
let text = "Hello,"  
    .bold()  
    .appending(" ")  
    .appending("world!".italic())  
text.render() // **Hello,** _world!
```

The difference is that the concrete type of `text` is now hidden, as it would be if the API had returned an existential. The new function signatures better reflect the contract we as authors of the `RichText` library want to provide to clients. Since callers can only access the stated capabilities of the opaque type (here, `RichText`), clients don't need to know about the internal types, and no one needs to update their code if the underlying concrete type ever changes.

Rules for Opaque Types

The formal rules for opaque types are outlined below.

- 1. Opaque types can appear as the return type of functions, properties/variables, or subscripts.** They're also called opaque *result* types because they're exclusively about output types.

Swift 5.7 will extend [the `some P` syntax to function parameters](#) as a lightweight alternative syntax for generic parameters. However, note that despite the identical spelling, there's an important semantic difference between opaque function parameters and opaque result types. When `some P` appears in a function parameter, the *caller* determines the type that's passed into the function, whereas the concrete type hidden behind an opaque result type is chosen by the function's *author*.

- 2. The constraint is usually a protocol, but it can also be a class constraint** (some `UIView` means any `UIView` subclass) or a composition of several constraints (some `AnyObject & Encodable`).

- 3. A function with an opaque type must return the same type in all code paths.** For example, this alternative implementation of appending that just returns `self` when the appended text is empty is invalid:

```
extension RichText {  
    // Error: function declares an opaque return type, but the return  
    // statements in its body do not have matching underlying types.  
    func appending2<Other: RichText>(_ other: Other) -> some RichText {  
        if other.render().isEmpty {  
            return self  
        } else {  
            return Concat(a: self, b: other)  
        }  
    }  
}
```

The compiler guarantees that both code paths return the same type. Although the concrete type isn't known outside the function's body, the type system treats the type

“return type of appending2” as a single identity that never changes. This is a major difference compared to how existentials are handled.

You may then wonder how SwiftUI allows conditionals with a different view type in each branch. For instance, this is perfectly legal:

```
struct ContentView: View {  
    var rounded: Bool  
    var body: some View {  
        if rounded {  
            Circle()  
        } else {  
            Rectangle()  
        }  
    }  
}
```

This works because the getter for the `body` property can implicitly become a result builder function. Note that we didn’t (and couldn’t) write `return` statements in the `if/else` branches. The concrete type of the opaque `some View` is neither `Circle` nor `Rectangle` here, but `_ConditionalContent<Circle, Rectangle>`. See the [Functions](#) chapter for more on result builders.

4. A function with an opaque type must return the same concrete type on every invocation. The compiler “knows” this and can make use of it. For instance, we can call a function that returns a `some BinaryInteger` multiple times and add the returned values:

```
func randomNumber() -> some BinaryInteger {  
    Int16.random(in: 1...20)  
}  
  
let a = randomNumber() // 20  
let b = randomNumber() // 3  
a + b // 23
```

If `randomNumber` returned an any `BinaryInteger` existential instead, this would be an error, because the `+` operator is only defined for operands of the same type, and existentials can box any conforming type.

5. You can recover the concrete type through dynamic casting. Opaque types are only known to the static type system; they have no representation at runtime. Calling

`type(of:)` on an opaque value returns the actual concrete type. To test if an opaque value has a particular concrete type, you can use the `is` or `as?` operators:

```
if let d = randomNumber() as? Int16 {
    print("\(d) is an Int16")
} else {
    print("It's some other type")
} /*end*/
// 4 is an Int16
```

Limitations of Opaque Types

SwiftUI has demonstrated the usefulness of opaque types for hiding deeply nested generic type signatures. Our own `RichText` example is fundamentally similar (on a much smaller scale). Opaque types in Swift 5.6 have some significant restrictions that limit their utility for many other use cases. We expect these restrictions to be lifted in future releases, making opaque types more broadly applicable.

1. Wrapping opaque types in other types isn't supported. For example, a function currently can't return an optional opaque type, `(some P)?`, or a tuple of opaque types, `(some P, some Q)`. This limitation will be [lifted in Swift 5.7](#).

2. An opaque type's identity is bound to the declaration site. That is, it's impossible to express that two functions return the same opaque type, which makes the following code illegal:

```
func positiveNumber() -> some BinaryInteger {
    Int16.random(in: 1...20)
}

func negativeNumber() -> some BinaryInteger {
    -Int16.random(in: 1...20)
}

let c = positiveNumber() // 6
let d = negativeNumber() // -2
c + d
// Error: operator '+' cannot be applied to operands of type
// 'some BinaryInteger' (result of 'positiveNumber()') and
// 'some BinaryInteger' (result of 'negativeNumber()').
```

The way the compiler sometimes refers to opaque types in diagnostics (“‘some BinaryInteger’ (result of ‘positiveNumber()’)” takes some getting used to, but it illustrates that the unique type identity is attached to the positiveNumber function, and not the some BinaryInteger constraint.

This restriction could be lifted by introducing a syntax for “opaque type aliases,” which would let programmers define a type alias for an opaque type constraint that they could then use in multiple declarations.

3. Opaque types can't have where constraints. This is the big piece of missing functionality that makes opaque types a lot less useful than they could be.

At the same time that Apple introduced SwiftUI, it also released the reactive programming framework Combine. It too uses deeply nested generic types with unwieldy type signatures. For example, here's a typical publisher (Combine's name for an event stream) with a few transformations applied to it:

```
let url: URL = ...
let downloadedUser = URLSession.shared.dataTaskPublisher(for: url)
    .map(\.data)
    .decode(type: User.self, decoder: JSONDecoder())
```

The type of this variable is:

```
Publishers.Decode<
    Publishers.MapKeyPath<URLSessionDataTaskPublisher, Data>,
    User,
    JSONDecoder
>
```

It's hard to read and doesn't give the reader the essential information: what are the output and failure types of this publisher?

Many Combine users resort to appending an .eraseToAnyPublisher() call to their publisher chains. This performs manual type erasure, which we'll discuss in detail later in the chapter. It solves the immediate problem — the type becomes AnyPublisher<User, Error> — but it comes with the usual downsides of premature type erasure.

On the surface, opaque types should be an ideal match for this: we want to hide the irrelevant concrete types from users while exposing only the relevant protocol. Indeed, we *can* write `let downloadedUser: some Publisher where .Output == User, .Failure == Error`

```
let downloadedUser: some Publisher where .Output == User, .Failure == Error  
= ...
```

This syntax is currently not supported, but we're hopeful we'll see it in a future Swift version. This would make opaque types much more capable and universally applicable.

Imagining the Standard Library with Opaque Types

If Swift had had opaque result types (with where constraints) from the beginning, the standard library would likely look different. Many collection APIs that currently return a concrete type would be clearer by using opaque types. Here are some fictional examples:

```
extension Sequence {  
    func enumerated() -> some Sequence  
        where .Element == (offset: Int, element: Self.Element)  
}  
  
extension BidirectionalCollection {  
    func reversed() -> some BidirectionalCollection  
        where .Element == Self.Element  
}  
  
func zip<S1: Sequence, S2: Sequence>(_ seq1: S1, _ seq2: S2)  
    -> some Sequence where .Element == (S1.Element, S2.Element)
```

The current concrete types that represent these sequences — `EnumeratedSequence`, `ReversedSequence`, `Zip2Sequence`, and many more — would still exist, but they wouldn't have to be public. By only exposing what's important, the APIs would better describe the contract between caller and callee.

Opaque Types Support Library Evolution

Opaque types provide a stable binary interface across module boundaries. A library can freely change the concrete type hidden behind an opaque result type without breaking its clients, even if clients link the library dynamically. For example, if Apple updates the implementation of some SwiftUI API to return a different concrete view type, existing apps that call this API won't break.

This works because code that accesses an instance of an opaque type does so indirectly through the type's value witness table. Because the value's memory layout is unknown at compile time, the compiler inserts code that goes through the value witness table every time an opaque value is accessed. This is the same mechanism we discussed in the [How Generics Work](#) section in the previous chapter.

Like generic values (and unlike existentials), the opaque value itself isn't boxed — an `Int` hidden behind a `some BinaryInteger` veil has the exact same memory layout as a plain `Int`. This is good for performance because values can be tightly packed in memory and take full advantage of CPU caches.

Naturally, the indirection on value accesses doesn't come for free. The cost is unavoidable when maintaining binary compatibility is a concern (as we've seen with SwiftUI), but this isn't always the case. When the body of a function with an opaque return type is visible to the caller (due to inlining or because the caller and callee are in the same module), the optimizer can actually see the specific concrete type the function returns and eliminate any indirection. This doesn't change the semantics of opaque types from the programmer's perspective, but it makes a lot of code that uses opaque types as fast as it'd be if it used concrete types.

Opaque Types vs. Existentials

Both opaque types and existentials hide the concrete type from the static type system and only allow access via the listed protocol constraints. As we've seen, the fundamental difference is that opaque types preserve type identity and existentials erase it. Unless you need the flexibility existentials provide, it's generally recommended to stick with opaque types because they better express most API contracts ("this function always returns the *same* type, it's just that the concrete type isn't important"). It's also a performance win, as the compiler can generally optimize opaque types better.

Type Erasers

We've seen that existentials are a form of *type erasure*. Due to the current restriction that limits existentials to protocols without any Self or associated type requirements, it's sometimes desirable to write types that perform type erasure but don't have the same usage restrictions that existentials have. We call these types (**manual**) **type erasers**. Even the more flexible existentials in Swift 5.7 won't cover all use cases (namely, type erasers that preserve one or more associated types), so writing manual type erasers remains relevant, at least for a while.

For example, consider the following expression:

```
let seq = [1, 2, 3].lazy.filter { $0 > 1 }.map { $0 * 2 }
```

Its type is `LazyMapSequence<LazyFilterSequence<[Int>, Int>`. As we chain more operations, the type becomes even more complex. Sometimes, we might want to **erase** that type and "just" return a Sequence with Int elements. Using an opaque type (some Sequence) or an existential (as of Swift 5.7) can express the former constraint (a Sequence) but not the latter (Int elements). Conveniently, the `AnySequence` struct allows us to hide the underlying type:

```
let anySeq = AnySequence(seq)
```

The type of `anySeq` is `AnySequence<Int>`. While it's much simpler to read and has the same interface, there's a price to pay: using `AnySequence` adds another level of indirection and is much slower than using the underlying sequence directly.

The standard library provides type erasers for a number of its protocols: for example, there's also `AnyCollection` and `AnyHashable`. In the remainder of this section, we'll look at a simple implementation of a type eraser for our `Restorable` protocol from earlier in this chapter. Here it is again:

```
protocol Restorable {
    associatedtype State: Codable
    var state: State { get set }
}
```

As a first attempt, we might write our `AnyRestorable` like what's shown below. However, that won't work, because the underlying `R` will still be visible in the generic parameter — we haven't won anything:

```
struct AnyRestorable<R: Restorable> {
    var restorable: R
}
```

Instead, we want AnyRestorable to be generic only over the State. This is a common pattern for type erasers: they erase the concrete conforming type while preserving one associated type that's material for users of the type eraser. For instance, AnyCollection<Element> preserves the erased collection's element type because most clients need the element type to work with the collection. All other specifics of the Collection conformance, including the other associated types, are erased.

To make AnyRestorable conform to Restorable, we'll also need to provide a state property. We'll use the same implementation technique as the standard library: it uses three classes to implement a single type eraser. First of all, we'll create an abstract class, AnyRestorableBoxBase, which is only generic over State, and not over Restorable. We'll conform to Restorable by implementing state with fatalError. This class is private to the implementation and will never be instantiated:

```
class AnyRestorableBoxBase<State: Codable>: Restorable {
    internal init() {}
    public var state: State {
        get { fatalError() }
        set { fatalError() }
    }
}
```

Next, we create a subclass of AnyRestorableBoxBase, which is generic over R. The special trick that makes the type erasure work is to constrain AnyRestorableBoxBase's generic parameter to be the same as the base class's R.State:

```
class AnyRestorableBox<R: Restorable>: AnyRestorableBoxBase<R.State> {
    var r: R
    init(_ r: R) {
        self.r = r
    }

    override var state: R.State {
        get { r.state }
        set { r.state = newValue }
    }
}
```

The subclass relationship means we can create an instance of AnyRestorableBox, but we use it as if it were an AnyRestorableBoxBase, which conveniently only exposes the generic State parameter. Because the latter class conforms to Restorable, we can use it as a Restorable straight away. As the final step, we create a wrapper struct, AnyRestorable, that hides AnyRestorableBox:

```
struct AnyRestorable<State: Codable>: Restorable {
    private let box: AnyRestorableBoxBase<State>
    init<R>(_ r: R) where R: Restorable, R.State == State {
        self.box = AnyRestorableBox(r)
    }

    var state: State {
        get { box.state }
        set { box.state = newValue }
    }
}
```

This struct is the only publicly visible part of the type eraser. It corresponds directly to AnySequence, AnyIterator, etc. in the standard library. Notice how the struct is only generic over State and the initializer is generic over R: Restorable. It's this initializer that ensures the relationship between the generic State parameter and the incoming (and to-be-erased) Restorable's State type.

In general, when we write a type eraser, we should take care to include all methods that are in the protocol. While the compiler will be helpful, it won't point out when we forget to include a method that's part of the protocol but has a default implementation. In a type eraser, we shouldn't rely on default implementations, but instead we should always call the underlying type's implementation, because it might have been customized.

Manual Type Erasure with Unlocked Existentials

While more powerful existentials in Swift 5.7 won't immediately make manual type-erasing wrappers obsolete, they enable a new way to write these wrappers that's more concise and easier for the compiler to optimize. We start by writing a straightforward AnyRestorable wrapper type, which can now store the existential directly:

```
struct AnyRestorable<State: Codable> {
    private var _value: any Restorable
```

```
init<R: Restorable>(_value: R) where R.State == State {
    self._value = value
}
}
```

We then run into a problem when trying to implement the protocol conformance for `Restorable`. Since we've erased the relationship between our `State` type and the `Restorable` existential, there's no simple way to implement the protocol requirement that references `State`. The trick is to write a protocol extension that acts as a bridge between the typed and the type-erased worlds:

```
private extension Restorable {
    // Forward to the requirement in an existential-accessible way.
    func _getStateThunk<_State>() -> _State {
        assert(_State.self == State.self)
        return unsafeBitCast(state, to: _State.self)
    }

    mutating func _setStateThunk<_State>(newValue: _State) {
        assert(_State.self == State.self)
        state = unsafeBitCast(newValue, to: State.self)
    }
}
```

In general, this extension will include one method for each of your protocol's requirements that can't be accessed on the existential directly. We had to split our mutable property requirement into separate getter and setter methods because the methods must be generic and properties can't have generic parameters. The protocol conformance can now forward to the new helper methods:

```
extension AnyRestorable: Restorable {
    var state: State {
        get { _value._getStateThunk() }
        set { _value._setStateThunk(newValue: newValue) }
    }
}
```

This clever piece of code takes advantage of the fact that inside a protocol extension, the protocol's associated types (and `Self`) are available and all APIs that reference them are callable. Bitcasting between `State` (the associated type) and `_State` (the generic

parameter) is safe because the initializer of our type-erasing wrapper ensures that only compatible types are involved.

Recap

Swift's protocols can be used together with generics to write reusable, extensible code. Advanced features like protocol extensions, conditional conformance, and associated types allow us to model complex interfaces.

If you think about it, the standard library and some other libraries use protocols for a different purpose than most app developers. The Collection protocol hierarchy has been meticulously designed to provide abstractions against which programmers can write meaningful generic algorithms. Each protocol has to justify its existence by (a) having the requirements and semantics that enable a new class of algorithms, (b) composing well with other protocols to enable even more algorithms, and (c) not being overly demanding to allow as many types as possible to implement it. But not every protocol has to be this way. There's nothing wrong with a simple protocol to abstract over a dependency for making your code more testable.

And keep in mind that like any abstraction, protocols can simplify code, but they can also have the opposite effect: we've definitely seen (and written) code that became very difficult to understand due to its overuse of protocols. In many cases, the code can be written in a simpler way by using values or functions. Finding the right balance takes some experience. When rewriting protocol-based code to regular functions, explicit protocol witnesses can be a first step.

Collection Protocols

11

We mentioned in the [Built-In Collections](#) chapter that Swift's collection types — like `Array`, `Dictionary`, and `Set` — are implemented on top of a rich set of abstractions for processing sequences of elements. This chapter is all about the `Sequence` and `Collection` protocols, which form the cornerstones of this model. We'll cover how these protocols work, why they work the way they do, and how you can write your own sequences and collections.

To better understand collection protocols, it can help to see them in an inheritance diagram:

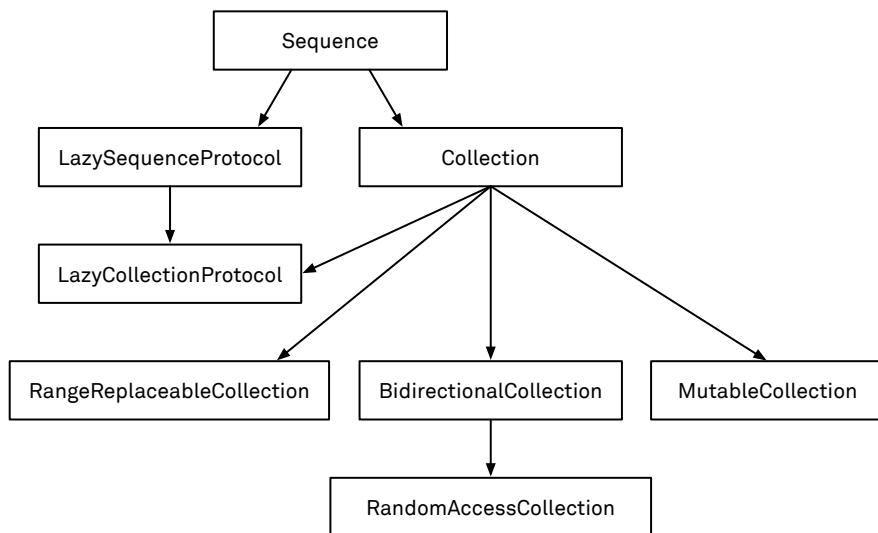


Figure 11.1: The collection protocol hierarchy in the standard library.

- **Sequence** provides iteration. It allows you to create an iterator, but there are no guarantees about whether the sequence is single-pass (e.g. reading from standard input) or multi-pass (iterating over an array).
- **Collection** extends `Sequence`. It guarantees that the sequence is multi-pass, and it allows you to look up elements by their indices. It also adds slicing capabilities via its `SubSequence` type, which is a collection itself.
- **MutableCollection** adds the ability to mutate an element through a subscript in constant time. It does *not* allow you to add or remove elements. `Array` is a

`MutableCollection`, but notably, `String` is not because it cannot guarantee constant-time mutation, as characters don't have a fixed width.

- **RangeReplaceableCollection** adds the ability to replace a contiguous range of elements in a collection. By extension, this also adds methods like `append`, `remove`, and so on. Many mutable collections are range-replaceable as well, but there are exceptions. Most notably, `Set` and `Dictionary` don't conform, but types like `String` and `Array` do.
- **BidirectionalCollection** adds the ability to iterate backward through a collection. For example, a `Dictionary` doesn't allow reverse iteration and doesn't conform, but a `String` does. Backward iteration is critical for some algorithms.
- **RandomAccessCollection** extends `BidirectionalCollection` and adds the ability to compute with indices more efficiently: it requires that measuring the distance between indices and moving indices by a certain distance takes constant time. For example, an `Array` is a random-access collection, but a `String` is not, because computing the distance between two string indices takes linear time.
- **LazySequenceProtocol** models a sequence that computes its elements lazily while it's being iterated. This is mostly useful for writing algorithms in a functional style: you can take an infinite sequence and filter it, and then take the first element, all without incurring the (infinite) cost of computing elements the subsequent code doesn't need.
- **LazyCollectionProtocol** is the same as `LazySequenceProtocol`, but for collections.

In this chapter, we'll look at each of these protocols in detail. Keep the protocol hierarchy in mind when you write your own collection algorithms: if you can write an algorithm on one of the protocols toward the root of the hierarchy, more types can take advantage of the algorithm.

Sequences

The `Sequence` protocol stands at the base of the hierarchy. A sequence is a series of values of the same type that lets you iterate over the values. The most common way to traverse a sequence is a `for` loop:

```
for element in someSequence {  
    doSomething(with: element)  
}
```

This seemingly simple capability of enumerating elements forms the foundation for a large number of useful operations Sequence provides to adopters of the protocol. We already saw many of them in the previous chapters. Whenever you come up with a common operation that depends on sequential access to a series of values, you should consider implementing it on top of Sequence too.

The requirements of the Sequence protocol are fairly small. All a conforming type must do is provide a `makeIterator()` method that returns an *iterator*:

```
protocol Sequence {  
    associatedtype Element  
    associatedtype Iterator: IteratorProtocol  
  
    func makeIterator() -> Iterator  
    // ...  
}
```

We can learn two things from this (simplified) definition of Sequence: a Sequence has an associated Element type, and it knows how to make an iterator. So let's first take a closer look at iterators.

Iterators

Sequences provide access to their elements by creating an iterator. The iterator produces the values of the sequence one at a time and keeps track of its own iteration state as it traverses the sequence. The only method defined in `IteratorProtocol` is `next()`, which must return the next element in the sequence on each subsequent call, or `nil` when the sequence is exhausted:

```
protocol IteratorProtocol {  
    associatedtype Element  
    mutating func next() -> Element?  
}
```

Most protocols don't end in Protocol, but there are a few exceptions in the standard library: `[Async]IteratorProtocol`, `StringProtocol`, `Keyed[En|De]CodingContainerProtocol`, and `Lazy[Collection|Sequence]Protocol`. This is done to avoid name clashes with associated or concrete types that

use the suffixless names. The [API Design Guidelines](#) suggest that protocols should either be nouns or have a suffix of -able, -ible, or -ing, depending on the protocol's role.

The associated `Element` type specifies the type of the values the iterator produces. For example, the element type of the iterator for `String` is `Character`. By extension, the iterator also defines its sequence's element type. This is done through a constraint on `Sequence`'s associated Iterator type — `Iterator.Element == Element` ensures that both element types are the same:

```
protocol Sequence {
    associatedtype Element
    associatedtype Iterator: IteratorProtocol
    where Iterator.Element == Element
    // ...
}
```

You normally only have to care about iterators when you implement one for a custom sequence type. Other than that, you rarely need to use iterators directly, because a `for` loop is the idiomatic way to traverse a sequence. In fact, this is how a `for` loop works under the hood: the compiler creates a fresh iterator for the sequence and repeatedly calls `next` on that iterator until `nil` is returned. The `for` loop example we showed above is essentially shorthand for the following:

```
var iterator = someSequence.makeIterator()
while let element = iterator.next() {
    doSomething(with: element)
}
```

Iterators are single-pass constructs; they can only be advanced, and they can never be reversed or reset. To restart iteration, you create a new iterator (in fact, that's exactly what `Sequence` allows through `makeIterator()`). While most iterators will produce a finite number of elements and eventually return `nil` from `next()`, nothing stops you from vending an infinite series that never ends. As a matter of fact, the simplest iterator imaginable — short of one that immediately returns `nil` — is one that just returns the same value over and over again:

```
struct ConstantIterator: IteratorProtocol {
    typealias Element = Int
    mutating func next() -> Int? {
```

```
    1
}
}
```

The explicit type alias for `Element` is optional (but it's often useful for documentation purposes, especially in larger protocols). If we omit it, the compiler infers the concrete type of `Element` from the return type of `next()`:

```
struct ConstantIterator: IteratorProtocol {
    mutating func next() -> Int? {
        1
    }
}
```

Notice that the `next()` method is declared as `mutating`. This isn't strictly necessary in this simplistic example because our iterator has no mutable state. In practice, though, iterators are inherently stateful. Almost any useful iterator requires mutable state to keep track of its position in the sequence.

We can create a new instance of `ConstantIterator` and loop over the sequence it produces in a while loop, printing an endless stream of ones:

```
var iterator = ConstantIterator()
while let x = iterator.next() {
    print(x)
}
```

Let's look at a more elaborate example. `FibsIterator` produces the [Fibonacci sequence](#). It keeps track of the current position in the sequence by storing the upcoming two numbers. The `next` method then returns the first number and updates the state for the following call. Like the previous example, this iterator also produces an “infinite” stream; it keeps generating numbers until it reaches integer overflow, and then the program crashes:

```
struct FibsIterator: IteratorProtocol {
    var state = (0, 1)
    mutating func next() -> Int? {
        let upcomingNumber = state.0
        state = (state.1, state.0 + state.1)
    }
}
```

```
    return upcomingNumber
}
}
```

Conforming to Sequence

A more useful example of a (finite) sequence would be the sequence of all nodes in an HTML document. In the [Enums](#) chapter, we defined an enum to represent HTML nodes:

```
enum Node: Hashable {
    case text(String)
    indirect case element(
        name: String,
        attributes: [String: String] = [:],
        children: Node = .fragment([]))
    case fragment([Node])
}
```

Here's an example of a `Node` (in the [Enums](#) chapter, we also defined a number of helper methods to create nodes, but we'll omit them here for the sake of brevity):

```
let header: Node = .element(name: "h1", children: .fragment([
    .text("Hello "),
    .element(name: "em", children: .text("World"))
]))
```

We can conform our `Node` type to `Sequence` by creating a custom iterator. Inside the iterator, we keep track of all the “remaining” nodes. We'll start by adding the root node of our HTML tree. The iterator always removes and returns the first element in the array of remaining nodes. If that element has any children, it'll add them to the remaining nodes:

```
struct NodeIterator: IteratorProtocol {
    var remaining: [Node]

    mutating func next() -> Node? {
        guard !remaining.isEmpty else { return nil }
        let result = remaining.removeFirst()
        switch result {
```

```
    case .text(_):
        break
    case .element(name: _, attributes: _, children: let children):
        remaining.append(children)
    case .fragment(let elements):
        remaining.append(contentsOf: elements)
    }
    return result
}
}
```

This isn't the only possible way to iterate over a tree of nodes. This iterator is a breadth-first traversal of the tree, but you could just as well use any other tree traversal algorithm. However, you can only conform `Node` to `Sequence` once, so you have to pick the algorithm that makes the most sense.

Conforming `Node` to `Sequence` is now as simple as creating the `NodeIterator`:

```
extension Node: Sequence {
    func makeliterator() -> NodeIterator {
        NodeIterator(remaining: [self])
    }
}
```

Just conforming to `Sequence` makes a lot of useful methods available on `Node`. For example, we can use `contains(where:)` to check if the document contains emphasized nodes:

```
header.contains(where: { node in
    guard case .element(name: "em", _, _) = node else { return false }
    return true
})
// true
```

Or we can use `compactMap` to extract all the text from a document:

```
header.compactMap { node -> String? in
    guard case let .text(t) = node else { return nil }
```

```
    return t
}
// ["Hello ", "World"]
```

There are many other useful operations: we can use Array's initializer to create an array of all the nodes, `allSatisfy` to check whether a condition holds for every element, or just a `for` loop to iterate over all the nodes of a document.

We can create sequences for `ConstantIterator` and `FibsIterator` in a similar way. We're not showing them here, but you may want to try this yourself. Just keep in mind that these iterators create infinite sequences. Use a construct like `for i in fibsSequence.prefix(10)` to slice off a finite piece.

Iterators and Value Semantics

The iterators we've seen thus far all have value semantics. If you make a copy of one, the iterator's entire state will be copied, and the two instances will behave independently of one other, as you'd expect. That said, most iterators in the standard library also have value semantics, but there are exceptions.

To illustrate the difference between value and reference semantics, we first take a look at `StrideTolterator`. It's the underlying iterator for the sequence that's returned from the `stride(from:to:by:)` function. Let's create a `StrideTolterator` and call `next` a couple times:

```
// A sequence from 0 to 9.
let seq = stride(from: 0, to: 10, by: 1)
var i1 = seq.makeIterator()
i1.next() // Optional(0)
i1.next() // Optional(1)
```

`i1` is now ready to return 2. Now, say you make a copy of it:

```
var i2 = i1
```

Both the original and the copy are now separate and independent, and both return 2 when you call `next`:

```
i1.next() // Optional(2)
i1.next() // Optional(3)
```

```
i2.next() // Optional(2)  
i2.next() // Optional(3)
```

This is because `StrideTolterator`, a pretty simple struct whose implementation isn't too different than that of our Fibonacci iterator above, has value semantics.

Now let's look at an iterator that doesn't have value semantics. `Anyliterator` is an iterator that wraps another iterator, thus "erasing" the base iterator's concrete type. An example where this might be useful is if you want to hide the concrete type of a complex iterator that would expose implementation details in your public API. The way `Anyliterator` does this is by wrapping the base iterator in an internal box object, which is a reference type. (If you want to learn exactly how this works, check out the section on type erasure in the Protocols chapter.)

To see why this is relevant, we create an `Anyliterator` that wraps `i1`, and then we make a copy:

```
var i3 = Anyliterator(i1)  
var i4 = i3
```

In this situation, the original and the copy aren't independent because, despite being a struct, `Anyliterator` doesn't have value semantics. The box object `Anyliterator` uses to store its base iterator is a class instance, and when we assigned `i3` to `i4`, only the reference to the box was copied. The storage of the box is shared between the two iterators. Any calls to `next` on either `i3` or `i4` now increment the same underlying iterator:

```
i3.next() // Optional(4)  
i4.next() // Optional(5)  
i3.next() // Optional(6)
```

Obviously, this could lead to bugs, although in all likelihood, you'll rarely encounter this particular problem in practice, as iterators are usually not something you pass around in your code. You're much more likely to create one locally — sometimes explicitly, but mostly implicitly through a `for` loop — use it once to loop over the elements, and then throw it away. If you find yourself sharing iterators with other objects, consider wrapping the iterator in a sequence instead.

Function-Based Iterators and Sequences

AnyIterator has a second initializer that takes the next function directly as its argument. Together with the corresponding AnySequence type, this allows us to create iterators and sequences without defining any new types. For example, we could've defined the Fibonacci iterator as a function that returns an AnyIterator:

```
func fibsIterator() -> AnyIterator<Int> {
    var state = (0, 1)
    return AnyIterator {
        let upcomingNumber = state.0
        state = (state.1, state.0 + state.1)
        return upcomingNumber
    }
}
```

By keeping the `state` variable outside of the iterator's `next` closure and capturing it inside the closure, the closure can mutate the state every time it's invoked. There's only one functional difference between the two Fibonacci iterators: the definition using a custom struct has value semantics, and the definition using AnyIterator doesn't.

Creating a sequence out of this is even easier now because AnySequence provides an initializer that takes a function that produces an iterator:

```
let fibsSequence = AnySequence(fibsIterator)
Array(fibsSequence.prefix(10)) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Another alternative is to use the `sequence` function, which has two variants. The first, `sequence(first:next:)`, returns a sequence whose first element is the first argument you passed in; subsequent elements are produced by the function passed in the `next` argument. The other variant, `sequence(state:next:)`, is even more powerful because it can keep some arbitrary mutable state around between invocations of the `next` function. We can use this to build the Fibonacci sequence with a single function call:

```
let fibsSequence2 = sequence(state: (0, 1)) { state -> Int? in
    let upcomingNumber = state.0
    state = (state.1, state.0 + state.1)
    return upcomingNumber
}
```

```
Array(fibsSequence2.prefix(10)) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

The return type of `sequence(first:next:)` and `sequence(state:next:)` is `UnfoldSequence`. This term comes from functional programming, where the same operation is often called *unfold*. The `sequence` function is the natural counterpart to `reduce` (which is often called *fold* in functional languages). Where `reduce` reduces (or *folds*) a sequence into a single return value, `sequence` expands (or *unfolds*) a single value to generate a sequence.

While `AnyIterator` often seems friendlier than a complicated iterator type with a long name, the standard library prefers custom iterator types for performance reasons. An `AnyIterator` makes it much more difficult for the compiler to optimize the code, which is something that can cause a loss in performance of up to 100 times. Ben wrote about this in detail [in a post on the Swift Forums](#).

Like all iterators we've seen so far, the `sequence` functions apply their `next` functions lazily, i.e. the next value isn't computed until it's requested by the caller. This makes constructs like `fibsSequence2.prefix(10)` work. `prefix(10)` only asks the sequence for its first (up to) 10 elements and then stops. If the sequence had tried to compute all its values eagerly, the program would've crashed with an integer overflow before the next step had a chance to run.

The possibility of creating infinite sequences is one thing that sets sequences apart from collections, which can't be infinite.

Single-Pass Sequences

Sequences aren't limited to classic collection data structures, such as arrays or lists. Network streams, files on disk, streams of UI events, and many other kinds of data can all be modeled as sequences. But not all of these behave like an array when you iterate over the elements more than once.

While the Fibonacci sequence isn't affected by a traversal of its elements (a subsequent traversal starts again at zero), a sequence that represents a stream of network packets is single-pass; it won't produce the same values again if you start a new iteration. However, both are valid sequences, so [the documentation is very clear that Sequence makes no guarantee about multiple traversals](#):

The Sequence protocol makes no requirement on conforming types regarding whether they'll be destructively consumed by iteration. As a consequence, don't assume that multiple for-in loops on a sequence will either resume iteration or restart from the beginning:

```
for element in sequence {  
    if ... some condition { break }  
}  
  
for element in sequence {  
    // No defined behavior  
}
```

A conforming sequence that isn't a collection is allowed to produce an arbitrary sequence of elements in the second for-in loop.

This also explains why the seemingly trivial first property is only available on collections and not on sequences. Invoking a property getter ought to be free of side effects, and only the Collection protocol guarantees safe multi-pass iteration.

As an example of a single-pass sequence, consider this wrapper around the `readLine` function, which reads lines from the standard input:

```
let standardIn = AnySequence {  
    AnyIterator {  
        readLine()  
    }  
}
```

Now you can use this sequence with the various extensions of `Sequence`. For example, you could write a line-numbering version of the Unix `cat` utility:

```
let numberedStdIn = standardIn.enumerated()  
for (i, line) in numberedStdIn {  
    print("\(i+1): \(line)")  
}
```

The `enumerated` method wraps a sequence in a new sequence that produces pairs of the original sequence's elements and incrementing numbers starting at zero. Just like our

wrapper of `readLine`, elements are lazily generated. The consumption of the base sequence only happens when you move through the enumerated sequence using its iterator, and not when it's created. So if you run the above code from the command line, you'll see it waiting inside the `for` loop. It prints the lines you type as you hit return; it does *not* wait until the input is terminated with Control-D. Nonetheless, each time `enumerated` serves up a line from `standardIn`, it's consuming the standard input. You can't iterate over it twice to get the same results.

As an author of a `Sequence` extension, you aren't required to consider whether or not the sequence is single-pass. However, if possible, try to write your algorithm using a single pass. As a *caller* of a method on a sequence type, you should definitely keep in mind whether or not the sequence is single-pass.

A certain sign that a sequence is multi-pass is if it also conforms to `Collection`, because this protocol makes that guarantee. The reverse isn't true. Even the standard library has some sequences that can be traversed safely multiple times although they aren't collections. Examples include the `StrideTo` and `StrideThrough` types, as returned by `stride(from:to:by:)` and `stride(from:through:by:)`.

The Relationship between Sequences and Iterators

Sequences and iterators are so similar that you may wonder why these need to be separate types at all. Can't we just fold the `IteratorProtocol` functionality into `Sequence`? This would indeed work fine for single-pass sequences, like our standard input example. Sequences of this kind carry their own iteration state and are mutated as they're traversed.

Multi-pass sequences, like arrays or our Fibonacci sequence, must not be mutated by a `for` loop; they require separate traversal state, and that's what the iterator provides (along with the traversal logic, but that might as well live in the sequence). The purpose of the `makeIterator` method is to create this traversal state.

Every iterator can also be seen as a single-pass sequence over the elements it has yet to return. As a matter of fact, you can turn every iterator into a sequence simply by declaring conformance; `Sequence` comes with a default implementation for `makeIterator` that returns `self` if the conforming type is an iterator:

```
extension Sequence where Iterator == Self {  
    func makeIterator() -> Self {
```

```
    return self
}
}
```

Most iterators in the standard library conform to Sequence.

Collections

A collection is a multi-pass sequence that can be traversed nondestructively multiple times. A collection's elements can not only be traversed linearly, but they can also be accessed via subscript with an index. Collection indices are often integers, as they're in arrays. But as we'll see, indices can also be opaque values (as in dictionaries or strings), which sometimes makes working with them non-intuitive. A collection's indices invariably form a finite range with a defined start and end. This means that, unlike sequences, collections can't be infinite. Every collection also has an associated SubSequence that represents a contiguous slice of the collection.

The Collection protocol builds on top of Sequence. In addition to all the methods inherited from Sequence, collections have additional capabilities that either depend on accessing elements at specific positions or rely on the guarantee of multi-pass iteration, like the count property (if counting the elements of a single-pass sequence consumed the sequence, it would kind of defeat the purpose).

Even if your custom sequence type doesn't need the special features of a collection, you can use Collection conformance to signal to users that your type is finite and supports multi-pass iteration. It's somewhat strange that you have to come up with an index if all you want is to document that your sequence is multi-pass, especially when you consider that picking a suitable index type is often the hardest part of implementing the Collection protocol. One reason for this design is that the Swift team wanted to avoid the potential confusion of having a distinct protocol for multi-pass sequences that had requirements identical to Sequence but different semantics.

Collections are used extensively throughout the standard library. In addition to Array, Dictionary, and Set, String and its views are all collections, as are [Closed]Range (conditionally) and UnsafeBufferPointer. Outside the standard library, Foundation's Data type is one of the most frequently used collection types. And the Swift Collections package provides additional data structures, such as ordered sets and dictionaries.

A Custom Collection

To demonstrate how collections in Swift work, we'll implement one of our own. Probably the most useful container type not present in the Swift standard library is a queue (although there's a [double-ended queue](#) available in the [Swift Collections package](#)). Swift arrays can easily be used as stacks, with `append` to push and `popLast` to pop. However, they're not ideal to use as queues. You could use `push` combined with `remove(at: 0)`, but removing the first element of an array is an $O(n)$ operation — because arrays are held in contiguous memory, every element has to shuffle down to fill the gap (unlike popping the last element, which can be done in constant time).

Below is a simple FIFO (first in, first out) queue, with just `enqueue` and `dequeue` methods implemented on top of two arrays:

```
/// An efficient variable-size FIFO queue of elements of type `Element`.
struct FIFOQueue<Element> {
    private var left: [Element] = []
    private var right: [Element] = []

    /// Add an element to the back of the queue.
    /// - Complexity:  $O(1)$ .
    mutating func enqueue(_ newElement: Element) {
        right.append(newElement)
    }

    /// Removes front of the queue.
    /// Returns `nil` if the queue is empty.
    /// - Complexity: Amortized  $O(1)$ .
    mutating func dequeue() -> Element? {
        if left.isEmpty {
            left = right.reversed()
            right.removeAll()
        }
        return left.popLast()
    }
}
```

This implementation uses a technique of simulating a queue through the use of two stacks (two regular arrays). As elements are enqueued, they're pushed onto the “right” stack. Then, when elements are dequeued, they're popped off the “left” stack, where

they're held in reverse order. When the left stack is empty, the right stack is reversed onto the left stack.

You might find the claim that the `dequeue` operation is $O(1)$ slightly surprising. Surely it contains a reversed call that's $O(n)$? But while this is true, the overall amortized time required to pop an item is constant — over a large number of pushes and pops, the time taken for them all is constant, even though the time for individual pushes or pops might not be.

The key to why this is lies in understanding how often the reversing happens and on how many elements. One technique to analyze this is the “banker’s methodology.” Imagine that each time you put an element on the queue, you deposit a token into the bank. Single enqueue, single token, so constant cost. Then when it comes time to reverse the right-hand stack onto the left-hand one, you have a token in the bank for every element enqueued, and you use those tokens to pay for the reversal. The account never goes into debit, so you never spend more than you paid.

This kind of reasoning is good for explaining why the “amortized” cost of an operation over time is constant, even though individual calls might not be. The same kind of justification can be used to explain why appending an element to an array in Swift is an (amortized) constant-time operation. When the array runs out of storage, it needs to allocate bigger storage and copy all its existing elements into the new storage. But since the storage size increases exponentially, you can use the same “append an element, pay a token, double the array size, spend all the tokens but no more” argument.

We now have a container that can enqueue and dequeue elements. The next step is to add Collection conformance to `FIFOQueue`. Unfortunately, figuring out the minimum set of implementations you must provide to conform to a protocol can sometimes be a frustrating exercise in Swift.

At the time of writing, the Collection protocol has a whopping five associated types, five properties, six instance methods, and two subscripts:

```
protocol Collection: Sequence {  
    associatedtype Element // Inherited from Sequence.  
    associatedtype Index: Comparable  
    // where ... clause omitted.
```

```

var startIndex: Index { get }
var endIndex: Index { get }

associatedtype Iterator = IndexingIterator<Self>
associatedtype SubSequence: Collection = Slice<Self>
  where Element == SubSequence.Element,
        SubSequence == SubSequence.SubSequence

subscript(position: Index) -> Element { get }
subscript(bounds: Range<Index>) -> SubSequence { get }

associatedtype Indices: Collection =
  DefaultIndices<Self> where Indices == Indices.SubSequence

var indices: Indices { get }
var isEmpty: Bool { get }
var count: Int { get }

func makeIterator() -> Iterator // Inherited from Sequence.
func index(_ i: Index, offsetBy distance: Int) -> Index
func index(_ i: Index, offsetBy distance: Int, limitedBy limit: Index) -> Index?
func distance(from start: Index, to end: Index) -> Int
func index(after i: Index) -> Index
func formIndex(after i: inout Index)
}

```

The associated SubSequence type uses a recursive constraint to indicate that a SubSequence should be a Collection too. It ensures that the SubSequence's elements are the same type as those in the collection, and that a SubSequence's SubSequence has the same type. For example, String has Substring as its SubSequence, and Substring has itself as its SubSequence.

The associated Indices type is also a collection. Note that we left out the long list of constraints on Index for the sake of brevity. In short, they ensure the Index is both the Element and the Index of the Indices collection, and of the SubSequence's indices, and of the Indices's Indices.

With all the requirements above, conforming to Collection seems like a daunting task. Well, it turns out it's actually not that bad. Notice that all associated types except Index and Element have default values, so you don't need to care about those unless your collection type has special requirements. The same is true for most of the methods,

properties, and subscripts: protocol extensions on Collection provide the default implementations. Some of these extensions have associated type constraints that match the protocol's default associated types; for example, Collection only provides a default implementation of the `makeIterator` method if its Iterator is an `IndexingIterator<Self>`:

```
extension Collection where Iterator == IndexingIterator<Self> {  
    func makeIterator() -> IndexingIterator<Self>  
}
```

If you decide your type should have a different iterator type, you'd have to implement the method above.

Working out what's required and what's provided through defaults isn't exactly hard, but it's a lot of manual work, and unless you're careful not to overlook anything, it's easy to end up in an annoying guessing game with the compiler. The most frustrating part of the process may be that the compiler *has* all the information to guide you; the diagnostics just aren't very helpful.

For the time being, your best hope is to find the minimal conformance requirements spelled out in the documentation, as is in fact the case for Collection:

... To add Collection conformance to your type, you must declare at least the following requirements:

- The `startIndex` and `endIndex` properties.
- A subscript that provides at least read-only access to your type's elements.
- The `index(after:)` method for advancing an index into your collection.

So in the end, we end up with these requirements:

```
protocol Collection: Sequence {  
    /// A type representing the sequence's elements.  
    associatedtype Element  
    /// A type that represents a position in the collection.  
    associatedtype Index: Comparable  
    /// The position of the first element in a non-empty collection.  
    var startIndex: Index { get }
```

```

/// The collection's "past the end" position — that is, the position one
/// greater than the last valid subscript argument.
var endIndex: Index { get }
/// Returns the position immediately after the given index.
func index(after i: Index) -> Index
/// Accesses the element at the specified position.
subscript(position: Index) -> Element { get }
}

```

We can conform FIFOQueue to Collection like so:

```

extension FIFOQueue: Collection {
    public var startIndex: Int { return 0 }
    public var endIndex: Int { return left.count + right.count }

    public func index(after i: Int) -> Int {
        precondition(i >= startIndex && i < endIndex, "Index out of bounds")
        return i + 1
    }

    public subscript(position: Int) -> Element {
        precondition((startIndex..<endIndex).contains(position),
                    "Index out of bounds")
        if position < left.endIndex {
            return left[left.endIndex - position - 1]
        } else {
            return right[position - left.endIndex]
        }
    }
}

```

We use Int as our queue's Index type. We don't specify an explicit type alias for the associated type; just like with Element, Swift can infer it from the method and property definitions. Note that since the indexing returns elements from the front first, FIFOQueue.first returns the next item that will be dequeued (so it serves as a kind of peek).

With just a handful of lines, queues now have more than 40 methods and properties at their disposal. For example, we can iterate over queues:

```
var queue = FIFOQueue<String>()
```

```
for x in ["1", "2", "foo", "3"] {
    queue.enqueue(x)
}

for s in queue {
    print(s, terminator: " ")
} // 1 2 foo 3
```

We can pass queues to methods that take sequences:

```
var a = Array(queue) // ["1", "2", "foo", "3"]
a.append(contentsOf: queue[2...3])
a // ["1", "2", "foo", "3", "foo", "3"]
```

We can call methods and properties that extend Sequence:

```
queue.map { $0.uppercased() } // ["1", "2", "FOO", "3"]
queue.compactMap { Int($0) } // [1, 2, 3]
queue.filter { $0.count > 1 } // ["foo"]
queue.sorted() // ["1", "2", "3", "foo"]
queue.joined(separator: " ") // 1 2 foo 3
```

And we can call methods and properties that extend Collection:

```
queue.isEmpty // false
queue.count // 4
queue.first // Optional("1")
```

Array Literals

When writing a collection like `FIFOQueue`, it's nice to implement `ExpressibleByArrayLiteral` too. This will allow users to create a queue using the familiar `[value1, value2, etc]` syntax. The protocol requires us to implement an initializer, like so:

```
extension FIFOQueue: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: Element...) {
        self.init(left: elements.reversed(), right: [])
    }
}
```

For our queue logic, we want to reverse the elements to have them ready for use in the left-hand buffer. Of course, we could just copy the elements to the right-hand buffer, but since we’re going to be copying elements anyway, it’s more efficient to copy them in reverse order so that they don’t need reversing later when they’re dequeued.

Now queues can easily be created from literals:

```
let queue2: FIFOQueue = [1,2,3] // FIFOQueue<Int>(left: [3, 2, 1], right: [])
```

It’s important to underline the difference between literals and types in Swift here. [1,2,3] is *not* an array. Rather, it’s an *array literal* — something that can be used to create any type that conforms to ExpressibleByArrayLiteral. This particular literal contains other literals — integer literals — which can create any type that conforms to ExpressibleByIntegerLiteral.

These literals have “default” types — types that Swift will assume if you don’t specify an explicit type when you use a literal. So array literals default to Array, integer literals default to Int, float literals default to Double, and string literals default to String. But this only occurs in the absence of you specifying otherwise. For example, the queue declared above is a queue of integers, but it could’ve been a queue of some other integer type:

```
let byteQueue: FIFOQueue<UInt8> = [1,2,3]
// FIFOQueue<UInt8>(left: [3, 2, 1], right: [])
```

Often, the type of the literal can be inferred from the context. For example, this is what it looks like if a function takes a type that can be created from literals:

```
func takesSetOfFloats(floats: Set<Float>) {
    ...
}
```

```
takesSetOfFloats(floats: [1,2,3])
```

This literal will be interpreted as Set<Float>, and not as Array<Int>.

Associated Types

We’ve seen that Collection provides defaults for all but two of its associated types; types adopting the protocol only have to specify an Element and an Index type. While you don’t *have* to care much about the other associated types, it’s a good idea to take a brief

look at each of them to better understand what their specific purposes are. Let's go through them one by one.

Iterator — Inherited from Sequence. We already looked at iterators in detail in our discussion on [sequences](#). The default iterator type for collections is `IndexingIterator<Self>`. This is a simple struct that wraps the collection and uses the collection's own indices to step over each element.

Most collections in the standard library use `IndexingIterator` as their iterator. There should be little reason to write your own iterator type for a custom collection.

SubSequence — The type of a contiguous slice of the collection's elements. Subsequences are themselves collections. The default SubSequence type is `Slice<Self>`, which wraps the original collection and stores the slice's start and end index in terms of the base collection.

It can make sense for a collection to customize its SubSequence type, especially if it can be Self (i.e. a slice of the collection has the same type as the collection itself). Foundation's Data is an example of a such collection. We'll talk more about [subsequences](#) later in this chapter.

Indices — The return type of the collection's `indices` property. It represents a collection containing all indices that are valid for subscripting the base collection, in ascending order. Note that the `endIndex` isn't included because it signifies the "past the end" position and thus isn't a valid subscript argument.

The default type is the imaginatively named `DefaultIndices<Self>`. Like `Slice`, it's a simple wrapper for the base collection and a start and end index — it needs to keep a reference to the base collection to be able to advance the indices. This can lead to unexpected performance problems if users mutate the collection while iterating over its indices: if the collection is implemented using copy-on-write (as all collections in the standard library are), the extra reference to the collection can trigger unnecessary copies. If your custom collection can provide an alternative Indices type that doesn't need to keep a reference to the base collection, this is a worthwhile optimization. This is true for all collections whose index math doesn't rely on the collection itself, like arrays or our queue. If your index is an integer type, you can use `Range<Index>`:

```
extension FIFOQueue: Collection {  
    // ...  
    typealias Indices = Range<Int>  
    var indices: Range<Int> {
```

```
    startIndex..endIndex
}
}
```

Indices

An index represents a position in the collection. Every collection has two special indices: `startIndex` and `endIndex`. The `startIndex` designates the collection's first element, and `endIndex` is the index that comes *after* the last element. As a result, `endIndex` isn't a valid index for subscripting; you use it to form open ranges of indices (`someIndex..endIndex`) or to compare other indices against it, e.g. as the break condition in a loop (`while someIndex < endIndex`).

Up to this point, we've been using integers as the index into our collections. Array does this, and (with a bit of manipulation) our `FIFOQueue` type does too. Integer indices are intuitive, but they're not the only option. The only requirement for a collection's `Index` is that it must be `Comparable`, which is another way of saying indices have a defined order.

Take `Dictionary`, for instance. It would seem that the natural candidates for a dictionary's indices would be its keys; after all, the keys are what we use to address the values in the dictionary. But the key can't be the index because you can't advance it — there's no way to tell what the next index after a given key would be. Also, subscripting with an index is expected to give direct element access, without detours for searching or hashing.

Accordingly, `Dictionary.Index` is an opaque value that points to a position in the dictionary's internal storage buffer. It really is just a wrapper for a single `Int` offset, but that's an implementation detail of no interest to users of the collection. (In fact, the reality is somewhat more complex. The index also includes a mutation counter that allows the dictionary to detect when it's being accessed with an invalid index. In addition, dictionaries that get passed to or returned from Objective-C APIs use an `NSDictionary` as their backing store for efficient bridging, and the index type for those dictionaries is different. But you get the idea.)

This also explains why subscripting a `Dictionary` with an index doesn't return an optional value, whereas subscripting with a key does. The `subscript(_ key: Key)` we're so used to is an additional overload of the subscript operator that's defined directly on `Dictionary`. It returns an optional `Value`:

```
struct Dictionary {  
    ...  
    subscript(key: Key) -> Value?  
}
```

In contrast, subscripting with an index is part of the Collection protocol and *always* returns a non-optional value, because addressing a collection with an invalid index (like an out-of-bounds index on an array) is considered a programmer error, and doing so is supposed to trap:

```
protocol Collection {  
    subscript(position: Index) -> Element { get }  
}
```

Notice the return type, `Element`. A dictionary's `Element` type is the tuple type `(key: Key, value: Value)`, so for `Dictionary`, this subscript returns a key-value pair and not just a `Value`. This is also why iterating over a dictionary in a `for` loop produces key-value pairs.

In the section on [array indexing](#) in the Built-In Collections chapter, we discussed why it makes sense even for a “safe” language like Swift not to wrap every failable operation in an optional or error construct. [“If every API can fail, then you can’t write useful code.”](#) You need to have some fundamental basis that you can rely on, and trust to operate correctly,” otherwise your code gets bogged down in safety checks.

Index Invalidation

Indices may become invalid when the collection is mutated. Invalidation could mean that the index remains valid but now addresses a different element, or that the index is no longer a valid index for the collection, and using it to access the collection will trap. This should be intuitively clear when you consider arrays. When you append an element, all existing indices remain valid. When you remove the first element, an existing index to the last element becomes invalid. Meanwhile smaller indices remain “valid,” but the elements they point to have changed.

A dictionary index remains stable when new key-value pairs are added *until* the dictionary grows so much that it triggers a reallocation. This is because, as elements are inserted, the location of the indexed element in the dictionary’s storage buffer doesn’t

change until the buffer has to be resized, thereby forcing all elements to be rehashed. Removing elements from a dictionary invalidates indices.

An index should be a pure value that only stores the minimal amount of information required to describe an element's position. In particular, indices shouldn't keep a reference to their collection, if at all possible, because that interferes with copy-on-write optimizations when the collection is being mutated in a loop. Similarly, a collection usually can't distinguish one of its "own" indices from one that came from another collection of the same type. Again, this is trivially evident for arrays. Of course, you can use an integer index that was derived from one array to index another:

```
let numbers = [1,2,3,4]
let squares = numbers.map { $0 * $0 }
let numbersIndex = numbers.firstIndex(of: 4)! // 3
squares[numbersIndex] // 16
```

This also works with opaque index types, such as `String.Index`. In this example, we use one string's `startIndex` to access the first character of another string:

```
let hello = "Hello"
let world = "World"
let hellolidx = hello.startIndex
world[hellolidx] // W
```

However, the fact that you can do this doesn't mean it's generally a good idea. If we had used the index to subscript into an empty string, the program would've crashed because the index was out of bounds. Also, since characters are variable width, an index to the n^{th} character in one string may not point to a valid character boundary in another string. We talked about this in detail in the [Strings](#) chapter.

There are legitimate use cases for sharing indices between collections, though. The biggest one is working with slices. The Collection protocol requires that an index of the base collection must address the same element in a slice of that collection, so it's always safe to share indices with slices.

Advancing Indices

Swift 3 introduced [a major change](#) to the way index traversal is handled for collections. The task of advancing an index forward or backward (i.e. deriving a new index from a given index) is now a responsibility of the collection, whereas up until Swift 2, indices

were able to advance themselves. Where you used to write `someIndex.successor()` to step to the next index, you now write `collection.index(after: someIndex)`.

Why did the Swift team decide to make this change? In short, performance. It turns out that deriving an index from another often requires information about the collection's internals. It doesn't for arrays, where advancing an index is a simple addition operation. But a string index, for example, needs to inspect the actual character data, because characters have variable sizes in Swift.

In the old model of self-advancing indices, this meant the index had to store a reference to the collection's storage. That extra reference was enough to defeat the copy-on-write optimizations used by the standard library collections and would result in unnecessary copies when a collection was mutated during iteration.

By allowing indices to remain pure values, the new model doesn't have this problem. It's also conceptually easier to understand and can make implementations of custom index types simpler. In most cases, an index can likely be represented with one or two integers that efficiently encode the position to the element in the collection's underlying storage.

The downside of the new indexing model is a more verbose syntax.

Custom Collection Indices

As an example of a collection with non-integer indices, we'll build a way to iterate over the words in a string. When you want to split a string into its words, the easiest way is to use `split(separator:maxSplits:omittingEmptySubsequences:)`. This method is defined on `Collection`, and it turns a collection into an array of `SubSequences` split at the provided separator:

```
var str = "Still I see monsters"
str.split(separator: " ") // ["Still", "/", "see", "monsters"]
```

Each word in the returned array is of type `Substring`, which is `String`'s associated `SubSequence` type. Whenever you need to split a collection, the `split` method is almost always the right tool to use. It does have one disadvantage though: it eagerly computes the entire array. If you have a large string and only need the first few words, this isn't very efficient. To be more efficient, we build a `Words` collection that doesn't compute all the words upfront but instead allows us to iterate lazily. (The [Swift Algorithms package](#) provides a [lazy variant of split](#), among many other useful collection algorithms. You should use that one instead of our version in production code.)

Let's start by finding the range of the first word in a Substring. We'll use spaces as word boundaries, although it's an interesting exercise to make this configurable. A substring might start with an arbitrary number of spaces, which we skip over. `start` is the substring with the leading spaces removed. We then try to find the next space; if there's any space, we use that as the end of the word boundary. If we can't find any more spaces, we use `endIndex`:

```
extension Substring {
    var nextWordRange: Range<Index> {
        let start = drop(while: { $0 == " " })
        let end = start.firstIndex(where: { $0 == " " }) ?? endIndex
        return start.startIndex..
    }
}
```

Note that `Range` is half-open: the upper bound, `end`, isn't included in the word range.

A logical first choice for the index type of a `Words` collection would be `Int`: the index i would mean the i^{th} word in the collection. However, accessing an element through the index-based subscript needs to be an $O(1)$ operation, and to find the i^{th} word, we'd have to process the entire string (which is an $O(n)$ operation).

Another choice for the index type would be to use `String.Index`. The collection's `startIndex` would be `string.startIndex`, the index after that would be the index of the beginning of the next word, and so on. Unfortunately, the subscript implementation will have a similar problem: finding the end of the word is also $O(n)$.

Instead, we make our index a wrapper around `Range<Substring.Index>`. Because collection indices must be `Comparable`, we also have to implement that protocol for our custom index type. To compare two indices, we only compare the range's lower bounds. This doesn't work for comparing `Ranges` in general, but for our purpose, it suffices. And since `Comparable` inherits from `Equatable`, our index type must implement `==` as well, but the compiler synthesizes that for us.

By marking the `range` property and the initializer as `fileprivate`, we make `WordsIndex` an opaque type; users of our collection don't know the internal structure, and the only way to create an index is through the collection's interface:

```
struct WordsIndex: Comparable {
    fileprivate let range: Range<Substring.Index>
    fileprivate init(_ range: Range<Substring.Index>) {
```

```

    self.range = range
}

static func <(lhs: WordsIndex, rhs: WordsIndex) -> Bool {
    lhs.range.lowerBound < rhs.range.lowerBound
}
}

```

We're now ready to build our Words collection. It stores the underlying String as a Substring (we'll see why in the section on slicing) and provides a start index and an end index. The Collection protocol requires `startIndex` to have a complexity of $O(1)$. Unfortunately, computing it takes $O(n)$, where n is the number of spaces at the start of the string. Therefore, we compute it in the initializer and store it instead of defining it as a computed property. For the end index, we use an empty range that's outside the bounds of the underlying string:

```

struct Words {
    let string: Substring
    let startIndex: WordsIndex

    init(_ s: String) {
        self.init(s[...])
    }

    private init(_ s: Substring) {
        self.string = s
        self.startIndex = WordsIndex(string.nextWordRange)
    }

    public var endIndex: WordsIndex {
        let e = string.endIndex
        return WordsIndex(e..

```

Collection also requires us to provide a subscript that accesses elements. Here we can directly use the index's underlying range. Note that using the range of the word as the index makes the implementation $O(1)$:

```

extension Words {
    public subscript(index: WordsIndex) -> Substring {

```

```
        string[index.range]
    }
}
```

As the final Collection requirement, we need a way to compute the index following a given index. The upper bound of the index's range points to the position after the current word, so unless this position is equal to the string's endIndex (indicating that we reached the end of the string), we can take the substring from the upper bound onward and then look for the next word range:

```
extension Words: Collection {
    public func index(after i: WordsIndex) -> WordsIndex {
        guard i.range.upperBound < string.endIndex else { return endIndex }
        let remainder = string[i.range.upperBound...]
        return WordsIndex(remainder.nextWordRange)
    }
}
```

```
Array(Words(" hello world test ").prefix(2)) // ["hello", "world"]
```

With some effort, the Words collection could be changed to solve more general problems. First of all, we could make the word boundary configurable: instead of using a space, we could pass in a function, isWordBoundary: (Character) -> Bool. Second, the code isn't really specific to strings: we could replace String with any kind of collection. For example, we could reuse the same algorithm to lazily split Data into processable chunks. Again, the Swift Algorithms package already does all this with its lazy overloads of `split`.

Subsequences

The Collection protocol has an associated type, SubSequence, which is used for returning contiguous subranges of a collection:

```
extension Collection {
    associatedtype SubSequence: Collection = Slice<Self>
    where Element == SubSequence.Element,
          SubSequence == SubSequence.SubSequence
}
```

`SubSequence` is used as the return type for operations that return slices of the original collection:

- `prefix` and `suffix` — take the first or last n elements
- `prefix(while:)` — take elements from the start as long as they conform to a condition
- `dropFirst` and `dropLast` — return subsequences where the first or last n elements have been removed
- `drop(while:)` — drop elements until the condition ceases to be true, and then return the rest
- `split` — break up the sequence at the specified separator elements and return an array of subsequences

In addition, the `range-based subscript` operator takes a range of indices and returns the corresponding slice. The advantage of returning a subsequence rather than, say, an array of elements, is that no new memory allocation is necessary because subsequences share their base collection's storage.

The term “subsequence” has a historical background. Originally, `SubSequence` was an associated type of the `Sequence` protocol. It was moved up to `Collection` in Swift 5 because it didn't really solve many problems for `Sequence`, and it caused performance issues. The name was kept to maintain backward compatibility with existing code, but in the context of collections, the term “slice” is often more appropriate and is used interchangeably with subsequence.

By default, a `Collection` has `Slice<Self>` defined as its `SubSequence`, but many concrete types have custom implementations: for example, `String` has `Substring` as its subsequence, and `Array` uses a special `ArraySlice` type.

It can sometimes be convenient if `SubSequence == Self` — i.e. if subsequences have the same type as the base sequence — because this allows you to pass a slice everywhere the base collection is expected. The `Data` type in Foundation does this, so when you see a `Data` instance, you can't know if it's a standalone value (with `startIndex == 0` and `endIndex == count`) or a slice of a larger `Data` instance (with non-zero-based indices). The standard library collections have separate slice types, however; the main motivation for this is to avoid accidental memory “leaks” that can be caused by a tiny slice that

keeps its base collection (which may be very large) alive much longer than anticipated. Making slices their own types makes it easier to bind their lifetimes to local scopes.

As the examples from the standard library in the list above illustrate, `SubSequence` and `[SubSequence]` are good return types for slicing operations. The following example splits a collection into batches of n elements. It does this by looping over the collection's indices in steps of n , slicing off chunks, and adding those chunks to a new array of subsequences. If the number of elements isn't a multiple of n , the last batch will be smaller:

```
extension Collection {  
    public func split(batchSize: Int) -> [SubSequence] {  
        var result: [SubSequence] = []  
        var batchStart = startIndex  
        while batchStart < endIndex {  
            let batchEnd = index(batchStart, offsetBy: batchSize,  
                limitedBy: endIndex) ?? endIndex  
            let batch = self[batchStart ..< batchEnd]  
            result.append(batch)  
            batchStart = batchEnd  
        }  
        return result  
    }  
}  
  
let letters = "abcdefg"  
let batches = letters.split(batchSize: 3) // ["abc", "def", "g"]
```

Swift Algorithms provides this operation under the name [chunks\(ofCount:\)](#).

Because a subsequence is a collection as well (with the same `Element` type), we can seamlessly process the result of a slicing operation with the same collection operations we already know. For example, let's verify that the element counts of all batches equals the element count of the base collection:

```
let batchesCount = batches.map { $0.count }.reduce(0, +) // 7  
batchesCount == letters.count // true
```

With their very low memory overhead, subsequences are great for intermediate values. But because of the danger that a very small slice may accidentally keep a very large base collection alive, it's not recommended to store a subsequence permanently (e.g. in a

property) or to pass it to another function that might do so. To cut the tie between a subsequence and its base collection, we can create a brand-new collection and pass the subsequence into the initializer, e.g. `String(substring)` or `Array(arraySlice)`.

Slices

All collections get a default implementation of the slicing operation and have an overload for subscript that takes a `Range<Index>`. This is the equivalent of `words.dropFirst()`:

```
let words: Words = Words("one two three")
let onePastStart = words.index(after: words.startIndex)
let firstDropped = words[onePastStart..
```

Since operations like `words[somewhere.. (slice from a specific point to the end) and words[words.startIndex.. (slice from the start to a specific point) are common, there are variants in the standard library that perform these operations in a more readable way:`

```
let firstDropped2 = words.suffix(from: onePastStart)
// or:
let firstDropped3 = words[onePastStart...]
```

By default, the type of `firstDropped` won't be `Words` — it'll be `Slice<Words>`. `Slice` is a lightweight wrapper on top of any collection. The implementation looks something like this:

```
struct Slice<Base: Collection>: Collection {
    typealias Index = Base.Index
    let collection: Base

    var startIndex: Index
    var endIndex: Index

    init(base: Base, bounds: Range<Index>) {
        collection = base
        startIndex = bounds.lowerBound
        endIndex = bounds.upperBound
    }
}
```

```

func index(after i: Index) -> Index {
    return collection.index(after: i)
}

subscript(position: Index) -> Base.Element {
    return collection[position]
}

subscript(bounds: Range<Index>) -> Slice<Base> {
    return Slice(base: collection, bounds: bounds)
}
}

```

Slice is a perfectly good default subsequence type, but every time you write a custom collection, it's worth investigating whether or not you can make the collection its own SubSequence. For Words, this is easy to do:

```

extension Words {
    subscript(range: Range<WordsIndex>) -> Words {
        let start = range.lowerBound.range.lowerBound
        let end = range.upperBound.range.upperBound
        return Words(string[start..<end])
    }
}

```

The compiler infers the SubSequence type from the return type of the range-based subscript.

Using the same type for a collection and its SubSequence makes life easier for users of the collection because they only have to understand a single type instead of two. On the flipside, using distinct types for “root” collections and their slices makes it clear when a slice holds on to the memory of its base collection, which is why the standard library has `ArraySlice` and `Substring`.

Subsequences Share Indices with the Base Collection

A formal requirement of the Collection protocol is that indices of a subsequence can be used interchangeably with indices of the original collection. [The documentation states](#):

A collection and its slices share the same indices. An element of a collection is located under the same index in a slice as in the base collection, as long as neither the collection nor the slice has been mutated since the slice was created.

An important implication of this model is that, even when using integer indices, a collection's index won't necessarily start at zero. Here's an example of the start and end indices of an array slice:

```
let cities = ["New York", "Rio", "London", "Berlin",
    "Rome", "Beijing", "Tokyo", "Sydney"]
let slice = cities[2...4]
cities.startIndex // 0
cities.endIndex // 8
slice.startIndex // 2
slice.endIndex // 5
```

Accidentally accessing `slice[0]` in this situation will crash your program. This is another reason to always prefer constructs like `for x in collection` over manual index math.

We mentioned above that `Data` is a particularly dangerous example of this feature. Since `Data` uses integer indices, it's natural to assume the indices always start at 0, but this can't be the case, because `Data` is its own `SubSequence` type.

If you need access to the indices, `for index in collection.indices` is preferable to manual index math if possible — with one exception: if you mutate a collection while iterating over its indices, any strong reference the `indices` object holds to the original collection will defeat the copy-on-write optimizations and may cause an unwanted copy to be made. Depending on the size of the collection, this can have a significant negative performance impact. (Not all collections use an `Indices` type that strongly references the base collection, but many do because that's what the standard library's `DefaultIndices` type does.)

To avoid this unwanted copy when iterating over indices, you can replace the `for` loop with a `while` loop and advance the index manually in each iteration, thus avoiding the `indices` property. Just remember that if you do this, always start the loop at `collection.startIndex` and not at 0.

Specialized Collections

Like all well-designed protocols, Collection strives to keep its requirements as minimal as possible. To allow a wide array of types to become collections, the protocol shouldn't require conforming types to provide more than is absolutely necessary to implement the desired functionality.

Two particularly interesting limitations are that a Collection can't move its indices backward, and that it doesn't provide any functionality — such as inserting, removing, or replacing elements — for mutating the collection. That isn't to say that conforming types can't have these capabilities, of course, but only that the protocol makes no assumptions about them.

Some algorithms have additional requirements, though, and it'd be nice to have generic variants of them, even if only some collections can use them. For this purpose, the standard library includes four specialized collection protocols, each of which refines Collection in a particular way to enable new functionality (the quotes are from the [standard library documentation](#)):

- **BidirectionalCollection** — “A collection that supports backward as well as forward traversal.”
- **RandomAccessCollection** — “A collection that supports efficient random-access index traversal.”
- **MutableCollection** — “A collection that supports subscript assignment.”
- **RangeReplaceableCollection** — “A collection that supports replacement of an arbitrary subrange of elements with the elements of another collection.”

Let's discuss them one by one.

BidirectionalCollection

BidirectionalCollection adds a single but critical capability: the ability to move an index backward using the `index(before:)` method. This is sufficient to give your collection a default last property, which matches first:

```
extension BidirectionalCollection {  
    /// The last element of the collection.  
    public var last: Element? {
```

```
        return isEmpty ? nil : self[index(before: endIndex)]
    }
}
```

Collection could certainly provide a `last` property itself, but that wouldn't be a good idea. To get the last element of a forward-only collection, you have to iterate all the way to the end, i.e. an $O(n)$ operation. It'd be misleading to provide a cute little property for the last element — it takes a long time for a singly linked list with a million elements to fetch the last element.

An example of a bidirectional collection in the standard library is `String`. For Unicode-related reasons that we went into in the [Strings](#) chapter, a character collection can't provide random access to its characters, but you can move backward from the end, character by character.

`BidirectionalCollection` also adds more efficient implementations of some operations that profit from traversing the collection backward, such as `suffix`, `removeLast`, and `reversed`. The latter doesn't immediately reverse the collection, but instead returns a lazy view:

```
extension BidirectionalCollection {
    /// Returns a view presenting the elements of the collection in reverse
    /// order.
    /// - Complexity: O(1)
    public func reversed() -> ReversedCollection<Self> {
        return ReversedCollection(_base: self)
    }
}
```

Just as with the enumerated wrapper on `Sequence`, no actual reversing takes place. Instead, `ReversedCollection` holds the base collection and uses a custom index type that traverses the base collection in reverse order. The collection then reverses the logic of all index traversal methods so that moving forward moves backward in the base collection, and vice versa.

Value semantics play a big part in the validity of this approach. On construction, the wrapper “copies” the base collection into its own storage so that a subsequent mutation of the original collection won't change the copy held by `ReversedCollection`. This means that it has the same observable behavior as the version of `reversed` that returns an array.

Most types in the standard library that conform to Collection also conform to BidirectionalCollection. However, types like Dictionary and Set don't — mostly as the idea of forward and backward iteration makes little sense for inherently unordered collections.

RandomAccessCollection

A RandomAccessCollection provides the most efficient element access of all — it can jump to any index in constant time. To do this, conforming types must be able to (a) move an index any distance, and (b) measure the distance between any two indices, both in $O(1)$ time. RandomAccessCollection redeclares its associated Indices and SubSequence types with stricter constraints — both must be random-access themselves — but otherwise adds no new requirements over BidirectionalCollection. Adopters must take care to meet the documented $O(1)$ complexity requirements, however. You can do this either by providing implementations of the `index(_:offsetBy:)` and `distance(from:to:)` methods, or by using an `Index` type that conforms to Strideable, such as `Int`.

At first, it might seem like RandomAccessCollection doesn't add much. Even a simple forward-traverse-only collection like our Words can advance an index by an arbitrary distance. But there's a big difference. For Collection and BidirectionalCollection, `index(_:offsetBy:)` operates by incrementing the index successively until it reaches the destination. This clearly takes linear time — the longer the distance traveled, the longer it'll take to run. Random-access collections, on the other hand, can just move straight to the destination.

This ability is key in a number of algorithms. For example, when you implement a generic binary search, it's crucial this algorithm is constrained to random-access collections only — otherwise, it'd be far less efficient than just searching through the collection from start to end.

A random-access collection can compute the distance between its `startIndex` and `endIndex` in constant time, which means the collection can also compute `count` in constant time out of the box.

MutableCollection

A mutable collection supports in-place element mutation. The single new requirement it adds to Collection is that the single-element subscript now must also have a setter. We can add conformance for our queue type:

```

extension FIFOQueue: MutableCollection {
    public var startIndex: Int { return 0 }
    public var endIndex: Int { return left.count + right.count }

    public func index(after i: Int) -> Int {
        i + 1
    }

    public subscript(position: Int) -> Element {
        get {
            precondition((0..

```

Notice that the compiler won't let us add the subscript setter in an extension to an existing Collection; it's not allowed to provide a setter without a getter, and we're not allowed to redefine the existing getter, so we have to replace the existing Collection-conforming extension. Now the queue is mutable via subscripts:

```

var playlist: FIFOQueue = ["Shake It Off", "Blank Space", "Style"]
playlist.first // Optional("Shake It Off")
playlist[0] = "You Belong With Me"
playlist.first // Optional("You Belong With Me")

```

MutableCollection conformance is a requirement for many algorithms that work on a collection in place. Examples in the standard library include in-place sorting, reversing, and the swapAt method.

Relatively few types in the standard library adopt `MutableCollection`; of the three major collection types, only `Array` does. `MutableCollection` allows changing the values of a collection's elements, but not the length of the collection or the order of the elements. This last point explains why `Dictionary` and `Set` do *not* conform to `MutableCollection`, although they're certainly mutable data structures.

Dictionaries and sets are *unordered* collections — the order of the elements is undefined as far as the code using the collection is concerned. However, *internally*, even these collections have a stable element order that's defined by their implementation. When you mutate a `MutableCollection` via subscript assignment, the index of the mutated element must remain stable, i.e. the position of the index in the indices collection must not change. `Dictionary` and `Set` can't satisfy this requirement because their indices point to the bucket in their internal storage where the corresponding element is stored, and that bucket could change when the element is mutated.

`String` isn't a `MutableCollection` either because characters in a string don't have a fixed size in the string's buffer. As a result, replacing a character can take linear time, as the tailend of the buffer may have to be moved a few bytes backward or forward to make room for the replacement character. Additionally, changing one character may form a new grapheme cluster with adjacent combining characters, thereby changing the string's count in the process.

RangeReplaceableCollection

For operations that require adding or removing elements, use the `RangeReplaceableCollection` protocol. This protocol requires two things:

- An empty initializer — This is useful in generic functions, as it allows a function to create new empty collections of the same type.
- A `replaceSubrange(_:_with:)` method — This takes a range to replace and a collection to replace it with.

`RangeReplaceableCollection` is a great example of the power of protocol extensions. You implement one flexible method, `replaceSubrange`, and from that you get a whole bunch of derived methods for free:

- `append(_:)` and `append(contentsOf:)` — Replace `endIndex.. (i.e. replace the empty range at the end) with the new element/elements.`

- **remove(at:)** and **removeSubrange(_:)** — Replace $i \dots i$ or subrange with an empty collection.
- **insert(at:)** and **insert(contentsOf:at:)** — Replace $i \dots i$ (i.e. replace the empty range at that point in the array) with a new element/elements.
- **removeAll** — Replace $\text{startIndex} \dots \text{endIndex}$ with an empty collection.

These methods are requirements of the protocol, which means that when a specific collection type can use knowledge about its implementation to perform these functions more efficiently, it can provide custom versions that will take priority over the default protocol extension ones.

We chose to have a simple inefficient implementation for our queue type. As we stated when defining the data type, the left stack holds the element in reverse order. To have a simple implementation, we need to reverse all the elements and combine them into the right array so that we can replace the entire range at once:

```
extension FIFOQueue: RangeReplaceableCollection {
    mutating public func replaceSubrange<C: Collection>(
        _ subrange: Range<Int>,
        with newElements: C) where C.Element == Element
    {
        right = left.reversed() + right
        left.removeAll()
        right.replaceSubrange(subrange, with: newElements)
    }
}
```

You might like to try implementing a more efficient version, which looks at whether or not the replaced range spans the divide between the left and right stacks. There's no need for us to implement the empty init in this example, since the `FIFOQueue` struct already has one by default.

Unlike `BidirectionalCollection` and `RandomAccessCollection`, where the latter extends the former, `RangeReplaceableCollection` doesn't inherit from `MutableCollection`; they form distinct hierarchies. An example of a standard library collection that does conform to `RangeReplaceableCollection` but isn't a `MutableCollection` is `String`. The reasons for this boil down to what we said above about indices having to remain stable during a single-element subscript mutation, which `String` can't guarantee. We talk more about this in the [Strings](#) chapter.

Composing Capabilities

The specialized collection protocols can be elegantly composed into a set of constraints that exactly matches the requirements of a particular algorithm. As an example, take the `sort` method in the standard library for sorting a collection in place (unlike its non-mutating sibling, `sorted`, which returns the sorted elements in an array). Sorting in place requires the collection to be mutable. If you want the sort to be fast, you also need random access. Last but not least, you need to be able to compare the collection's elements to each other.

Combining these requirements, the `sort` method is defined in an extension to `MutableCollection`, with `RandomAccessCollection` and `Element: Comparable` as additional constraints:

```
extension MutableCollection
    where Self: RandomAccessCollection, Element: Comparable {
    /// Sorts the collection in place.
    public mutating func sort() { ... }
}
```

Lazy Sequences

The standard library provides two protocols for lazy evaluation: `LazySequenceProtocol` and `LazyCollectionProtocol`. *Lazy evaluation* means the results are only computed when needed, in contrast to *eager evaluation*, which is the default in Swift. A lazy sequence creates a separation between the producer and consumer of the sequence: you don't construct the entire sequence upfront. Instead, once the consumer asks for the next element, the lazy sequence produces it. This can be used both to achieve a different programming style, and for performance reasons.

We already saw a number of lazy sequences in this chapter. We showed three variants of constructing the Fibonacci sequence: using a custom iterator, using `AnySequence`, and using the `sequence(first:next:)` method. Each of these variants produces an infinite, lazy stream of values: the next value only gets computed when the consumer calls `next`. This allows the consumer to decide how many elements it needs. For example, we can take the first element of that stream, or a prefix.

One problem with lazily computed sequences is that some transformations require computing the entire sequence. For example, recall our definition of `standardIn`,

single-pass sequence that reads strings from the standard input line by line. We might want to only print lines that match a specific condition. In a functional style, we'd probably use filter to achieve this. However, because the return type of filter on Sequence is an array (which must be computed eagerly), the implementation has no other possibility than processing the entire standard input before it can return the result. In other words, the result is no longer lazy. The following code will print all the lines that have more than three words, but only *after* the standard input sends an end-of-file (EOF) signal:

```
let filtered = standardIn.filter {
    $0.split(separator: " ").count > 3
}
for line in filtered { print(line) }
```

Ideally, we'd like our program to print matching lines as they arrive. One solution is to switch to a more imperative style:

```
for line in standardIn {
    guard line.split(separator: " ").count > 3 else { continue }
    print(line)
}
```

If we want to use the functional style with method chaining instead of a for loop, we have to produce the filtered items as they arrive, rather than after all the input has been read. In other words: we need to construct a lazy sequence. The standard library provides a .lazy property on Sequence that helps us achieve this. We can write the following code:

```
let filtered = standardIn.lazy.filter {
    $0.split(separator: " ").count > 3
}
for line in filtered { print(line) }
```

The return type of .lazy is LazySequence<Self>. This means that standardIn.lazy has the type LazySequence<AnySequence<String>>. The LazySequence stores the underlying sequence internally, but it doesn't do any additional processing. The filter method on LazySequence then returns a LazyFilterSequence<AnySequence<String>>. Internally, it stores the underlying sequence and the predicate function that's passed to filter.

In our code example above, this means that the lines get read from the standard input as we iterate over filtered. Instead of waiting for the EOF marker, the lines get printed immediately. Furthermore, no intermediate array is constructed.

It's also interesting to see how the code changes as the requirements of our program change. For example, let's only print the first line that has more than three words. Using lazy, we can call .first while leaving our filtered definition unchanged:

```
let filtered = standardIn.lazy.filter {
    $0.split(separator: " ").count > 3
}
print(filtered.first ?? "<none>")
```

Without lazy, this would read all the lines from the standard input until receiving the EOF signal and only then print the first matching line. With lazy, it only reads until the first match and leaves the rest of the standard input unprocessed. To change the code to print the first two matching lines, we can change the call to first to prefix(2). Making changes to these functional pipelines is often shorter and clearer than changing the same code in an imperative style. For example, here's the same code, once with lazy, and once written imperatively:

```
// Functional.
let result = Array(standardIn.lazy.filter {
    $0.split(separator: " ").count > 3
}.prefix(2))

// Imperative.
var result: [String] = []
for line in standardIn {
    guard line.split(separator: " ").count > 3 else { continue }
    result.append(line)
    if result.count == 2 { break }
}
```

Lazy Processing of Collections

A lazy sequence can also be more efficient when chaining multiple methods on a regular collection type, such as an Array. If you prefer a more functional style, you might be tempted to write code like this:

```
/*_*/(1.. $<100$ ).map { $0 * $0 }.filter { $0 > 10 }.map { "\($0)" }
```

When you're used to functional programming, the code above is clear and easy to read. However, it's also inefficient: each call to map and filter creates an intermediate array that gets destroyed in the next step. By inserting .lazy at the beginning of the chain, no intermediate arrays are constructed, and the code runs much faster:

```
/*_*/(1.. $<100$ ).lazy.map { $0 * $0 }.filter { $0 > 10 }.map { "\($0)" }
```

The LazyCollectionProtocol extends LazySequence by requiring that the conforming type is a Collection as well. In a lazy sequence, we can only produce the next element lazily, but in a collection, we can also compute individual elements lazily. For example, when you map over a lazy collection and then access an element using a subscript, the map transformation is only applied to this one element (here, at index 50):

```
let allNumbers = 1.. $<1_000_000$ 
let allSquares = allNumbers.lazy.map { $0 * $0 }
print(allSquares[50]) // 2500
```

Using lazy collections isn't always a performance win; there are a few things to keep in mind. When accessing elements through a subscript, the value gets computed each time you access it. Likewise, depending on the computations the collection must perform, subscript access might not be $O(1)$ anymore. For example, consider the following code:

```
let largeSquares = allNumbers.lazy.filter { $0 > 1000 }.map { $0 * $0 }
print(largeSquares[50]) // 2500
```

Before the print statement executes, neither filter nor map have done any real work. Because of the filter operation, index 50 in largeSquares doesn't correspond to index 50 in allNumbers. To find the correct element, filter needs to compute the first 51 elements on every access through the subscript, and this is clearly not a constant-time operation.

Recap

The Sequence and Collection protocols form the foundation of Swift's collection types. They provide dozens of common operations for conforming types and act as constraints for your own generic functions. The specialized collection types, such as MutableCollection or RandomAccessCollection, give you fine-grained control over the functionality and performance requirements of your algorithms.

The high level of abstraction necessarily makes the model complex, so don't feel discouraged if not everything makes sense immediately. It takes practice to become comfortable with the strict type system, especially since, more often than not, discerning what the compiler wants to tell you is an art form that forces you to carefully read between the lines. The reward is an extremely flexible system that can handle everything from a pointer to a memory buffer to a destructively consumed stream of network packets.

This flexibility means that once you've internalized the model, chances are that a lot of code you'll come across in the future will instantly feel familiar because it's built on the same abstractions and supports the same operations. And whenever you create a custom type that fits in the Sequence or Collection framework, consider adding the conformance. It'll make life easier both for you and for other developers who work with your code.

Concurrency

12

In Swift 5.5, fundamental concurrency features were added to the language. This allows us to write concurrent code with strong compiler support, making a certain class of bugs impossible.

Async/await is the most prominent change, and it enables us to use Swift's structured programming techniques, such as the built-in error handling, for asynchronous code as if we were writing synchronous code. Compared to completion handlers, `async/await` is simpler and easier to reason about.

In this chapter, we'll load [Swift Talk](#) episodes, collections, and related data from the network as an example. We predefined `Episode` and `Collection` structs that both conform to `Codable` and `Identifiable`. For example, here's the `Episode` type:

```
struct Episode: Identifiable, Codable {
    var id: String
    var poster_url: URL
    var collection: String
    // ...
    static let url = URL(string: "https://talk.objc.io/episodes.json")!
}
```

Loading the episodes from the network and parsing them as JSON looks like this using `async/await`:

```
func loadEpisodes() async throws -> [Episode] {
    let session = URLSession.shared
    let (data, _) = try await session.data(from: Episode.url)
    return try JSONDecoder().decode([Episode].self, from: data)
}
```

Here's the same example without `async/await`, using a completion handler instead:

```
func loadEpisodesCont(
    _ completion: @escaping (Result<[Episode], Error>) -> ())
{
    let session = URLSession.shared
    let task = session.dataTask(with: Episode.url) { data, _, err in
        completion(Result {
            guard let d = data else {
                throw (err ?? UnknownError())
            }
        })
    }
}
```

```
        }
        return try JSONDecoder().decode([Episode].self, from: d)
    })
}
task.resume()
}
```

Note that in the first example (using `async/await`), the execution is from top to bottom, just like in normal structured programming. While the data is being loaded from the network, the function is **suspended**. Meanwhile, in the second example (using completion handlers), the execution branches: one part continues running after resuming the data task, whereas the other part (the body of the completion handler) runs asynchronously when the network request has finished (or failed). Looking at the types of our functions, we can see that the first function returns an array or throws an error. With the completion handler, it's only by convention that we assume the completion handler will be called exactly once.

A completion handler can also be called a **continuation**. The name describes the handler's purpose as the point where execution continues when `loadEpisodesCont` has finished its work. Likewise, the `session.dataTask` method doesn't return the data; rather it calls the provided continuation (in this case, the closure expression). `Async/await` also uses continuations under the hood. In the above example, the part after `await` is the continuation.

The concurrency system in Swift is only in its beginnings. More features will be added in the future, and the existing features will be refined and improved. Additionally, some compile-time checks aren't yet fully implemented. To make sure your code is future-proof, add the following Swift compiler flags: `-Xfrontend -warn-concurrency` and `-Xfrontend -enable-actor-data-race-checks`. These will give you useful warnings when you use unsafe constructs.

Async/Await

Before `async/await`, we wrote our non-blocking code using completion handlers (continuations) and delegates. This has been standard practice for many years in the Apple ecosystem. However, this pattern can lead to deeply nested continuations that are hard to follow. Even worse, standard structured programming constructs — such as Swift's built-in error handling, and `defer` — cannot be used.

When we compare the two code examples above, a few things stand out. First, the `async/await` code is shorter (three lines vs. ten lines for the completion handler-based approach). Second, completion handlers are harder to read; the order of execution isn't from top to bottom, as the continuation executes after the function has returned. Finally, when writing code with completion handlers, it's easy to make mistakes — for example, it's easy to forget to call the completion handler in the error case. By looking at the type of the function, it's unclear how often the completion handler gets called; you have to consult the documentation and trust that it's up to date.

Essentially, the `async/await` version is shorter and easier to understand. It's also more precise; by looking at the type, we know that it'll return either an error or an array of episodes, and that it'll return exactly once.

Combining `async/await` code is much simpler than combining code with completion handlers, especially when considering error handling. For example, let's extend our previous example to download the poster image of the first episode. The only thing we have to do is add another line to our function:

```
func loadFirstPoster() async throws -> Data {  
    let session = URLSession.shared  
    let (data, _) = try await session.data(from: Episode.url)  
    let episodes = try JSONDecoder().decode([Episode].self, from: data)  
    let (imageData, _) = try await session.data(from: episodes[0].poster_url)  
    return imageData  
}
```

In comparison, it's much harder to add loading of the first poster image to the code using completion handlers. We now have to take a lot of care to ensure our error handling is correct. If we can't load the initial data or if we have a parsing error, we need to make sure we immediately call our completion handler with the error. Otherwise, we can continue loading the poster image and call our completion handler once it's done.

The implementation using completion handlers is much more complex: it isn't obvious that it's correct (are we calling the completion handler exactly once for all different scenarios?), and the control flow is much harder to follow. If we do make a mistake (for example, not calling the completion handler), we can't rely on the compiler to warn us about it:

```
func loadFirstPosterCont(_ completion: @escaping (Result<Data, Error>) -> ()) {  
    let session = URLSession.shared  
    let task = session.dataTask(with: Episode.url) { data, _, err in
```

```

do {
    guard let d = data else {
        throw (err ?? UnknownError())
    }
    let episodes = try JSONDecoder().decode([Episode].self, from: d)
    let inner = session.dataTask(with: episodes[0].poster_url) { data, _, err in
        completion(Result {
            guard let d = data else {
                throw (err ?? UnknownError())
            }
            return d
        })
    }
    inner.resume()
} catch {
    completion(.failure(error))
}
task.resume()
}

```

We've now seen that both the implementation and the interface of `async/await` code are much simpler than code that uses completion handlers, and the compiler can eliminate an entire class of bugs (though not all). Yet another advantage of `async/await` over completion handlers is that we can use `defer` to perform any actions that should be done before exiting the scope. However, note that the body of a `defer` statement doesn't allow concurrent code. For example, you can't asynchronously update a model object inside a `defer` statement.

How Asynchronous Functions Execute

To understand the execution of an asynchronous function (from here on out referred to as an “`async` function”), we split it up into separate parts, where each part is delimited by a potential **suspension point**. In other words, we split the function up at each `await` statement. Consider the function from before, now annotated with the parts:

```

func loadFirstPoster() async throws -> Data {
    // Part 1
    let session = URLSession.shared
    let (data, _) = try await session.data(from: Episode.url)

```

```
// Part 2
let episodes = try JSONDecoder().decode([Episode].self, from: data)
let imageData = try await session.data(from: episodes[0].poster_url).0
// Part 3
return imageData
}
```

If we rewrite our function to use completion handlers, the parts correspond to each of the synchronous blocks within the function. Under the hood, Swift rewrites every async function containing suspension points to continuations. The first part of the function above executes normally, but parts two and three will both be continuations.

When executing the code above, “Part 1” is executed synchronously. This unit of work is called a **job**. The function is then suspended as it waits for the data to load from the network. This loading is done in a separate job. Once the data has loaded, `loadFirstPoster` resumes and a new job runs, which synchronously executes the code marked as “Part 2.” To load the image data, the function is suspended again and another job is scheduled. Once that job finishes, our function can resume and finally return its value.

Swift’s concurrency model is called **cooperative multitasking**. In short, this means functions should never block the current thread, but voluntarily suspend instead. A function can only be suspended at a potential suspension point (marked with `await`). When a function is suspended, this doesn’t mean the current thread is blocked: instead, control is given back to the scheduler, and other jobs (corresponding to other tasks) can run on the thread in the meantime. At a later point in time, the scheduler resumes the function by calling its continuation. For example, in the function above, the function suspends before “Part 2” while the data is being loaded from the network. Other jobs might execute in between, and once the data is available, the function resumes.

Note that a suspended function isn’t guaranteed to resume on its original thread. The Swift runtime maintains a thread pool for asynchronous tasks to run on and executes jobs on these threads as they become available. The particular thread an async function is running on shouldn’t be (and usually isn’t) important — unless we’re talking about the main thread, which still gets special treatment.

The potential thread hopping between jobs means we can’t rely on thread-local values to be the same after a suspension point. Therefore, APIs that use thread-local values, such as the `Progress` class in Foundation, aren’t

compatible with `async` functions. The concurrency-aware replacement for thread-local storage is called **task-local storage**. We won't discuss it in this chapter, but you can read up on it in the corresponding [Swift Evolution proposal](#).

The current thread isn't the only datum that can change at a suspension point, however. More generally, you must assume on resumption that *all non-local state may have changed* because other code had the chance to run while the function was suspended (non-local state includes global variables and properties on `self`, unless `self` has value semantics). This can cause bugs that are hard to find and that the compiler can't catch either. We'll come back to this point in the [Actor Reentrancy](#) section.

The concurrency model is called cooperative because it relies on the individual functions working together: no function should ever block its thread while waiting for expensive I/O operations or by executing long-running work. Instead, functions should suspend themselves by using other `async` functions for I/O, or by calling `Task.yield()` in between long-running work to give other jobs the chance to execute.

`Async/await` is best suited for I/O-bound waiting, because Swift's concurrency system is designed for efficient task switching. Suspending one function and resuming another on the freed-up thread is a lot faster than switching to another thread or bringing up a new thread. If functions can wait for slow I/O without occupying a thread, the system can keep the CPU cores busy with little overhead.

The concurrency system also supports cancellation, which is cooperative as well. We'll talk more about cancellation [later in this chapter](#).

You may have noticed that the call to `JSONDecoder().decode` in our example is *not* `async` because this API currently doesn't have an `async` variant. If we expect to process large amounts of data, the decoding step could take several 100 milliseconds or even longer, and during this time, our function isn't a good concurrency citizen because it doesn't give other tasks the chance to run.

Having said that, briefly occupying a thread isn't the end of the world, especially if the CPU is doing something useful (as it is during JSON decoding). Unless all running functions start "misbehaving" at the same time, the scheduler can still dispatch jobs to other CPU cores — the cooperative thread pool is designed to use roughly as many threads as the machine has CPU cores.

Keep in mind, however, that the scheduler won't spawn new threads to keep the program responsive, as that could lead to *thread explosion*, which has been a source of performance problems in Grand Central Dispatch (GCD). The flipside is that a number of non-cooperating functions can bring the entire thread pool to a halt, so we should really avoid blocking if possible.

In theory, each function part between two suspension points will be synchronously executed as a separate job. Every call to an async function also creates a separate job, which possibly executes on a different actor. However, sometimes multiple jobs can be "fused" together and effectively executed as one big uninterrupted job.

For example, Foundation has a bytes property on FileHandle to asynchronously read the bytes from a file. This is implemented using the `AsyncBytes` type. `AsyncBytes` is an `AsyncSequence` (the asynchronous variant of `Sequence`), with a `next` method that returns a single byte. The `next` method is roughly implemented like this:

```
mutating func next() async {
    if !buffer.isEmpty {
        return buffer.removeFirst()
    } else {
        return await reloadBufferAndNext()
    }
}
```

Due to job fusion, calling the `next` method will effectively be like a regular synchronous call, unless the buffer is empty, in which case the method suspends and only returns after reading more data. Even though we'll have to write `await` when calling `next`, it truly is a *potential* suspension point: in most cases, the buffer won't be empty, and the current task won't be suspended.

Interfacing with Completion Handlers

In existing projects, you might have code that uses completion handlers. This could be your own code, third-party code, or Apple code that hasn't yet been updated to `async/await`. Using `withCheckedContinuation` (or one of three closely related variants), you can wrap any function that takes a completion handler in an `async` function.

For example, let's imagine we have a function for loading a single episode with a completion handler:

```
func loadEpisodeCont(id: Episode.ID,  
    _ completion: (Result<Episode, Error>) -> () {  
    // ...  
}
```

To turn this into an async API, we write a new async function and wrap the invocation of the original function in a call to `withCheckedThrowingContinuation`. The latter will execute its body immediately and then suspend until we call `cont.resume` inside the body:

```
func loadEpisode(id: Episode.ID) async throws -> Episode {  
    try await withCheckedThrowingContinuation { cont in  
        loadEpisodeCont(id: id) {  
            cont.resume(with: $0)  
        }  
    }  
}
```

Because the completion handler in our original `loadEpisodeCont` method might be called with an error value (i.e. the `Result` is a `.failure` case), we need to mark our async wrapper method as `throws`, and we need to use `withCheckedThrowingContinuation` rather than `withCheckedContinuation`. The inclusion of “Checked” means that both of these functions perform runtime checks to ensure we call `resume` exactly once. Calling it more than once is a runtime error. If we discard the continuation before calling it, we also get a warning at runtime.

`withUnsafeContinuation` and `withUnsafeThrowingContinuation` are two additional variants, and they’re a little bit more efficient at runtime, because they skip the safety checks of their checked counterparts: you need to make sure to call the continuation exactly once. Not calling the continuation causes the task to never resume, and calling it more than once is undefined behavior. It’s good practice to write your code with the checked variants and make sure they work before switching to the unsafe variants (if you really need the performance).

The `with[...].Continuation` functions are useful beyond interfacing with existing completion handler-based code. In essence, they allow you to manually suspend a task and resume it at a later point. In our experience, you need to take great care to get this right in any but the most simple scenarios. As an example, the `AsyncStream` type from the standard library [uses `withUnsafeContinuation`](#) to suspend a consumer while waiting for more items to be produced.

By the way, rather than writing `loadEpisode` as an `async` function, we could also write it as an `async` `init` instead:

```
extension Episode {
    init(id: Episode.ID) async throws {
        self = try await withCheckedThrowingContinuation { cont in
            loadEpisodeCont(id: id, cont.resume(with:))
        }
    }
}
```

When you want to access your `async` methods from Objective-C, there are some limitations. You can mark `async` methods as `@objc` to make them visible to Objective-C as methods with a completion handler, but you can't mark the `init` variant as `@objc` because `async` initializers aren't bridged. There's a general limitation that global Swift functions can't be marked as `@objc`, so they have to be exposed as methods on a class that's already visible to Objective-C.

Structured Concurrency

We saw above how `async/await` makes asynchronous code structured, enabling us to use Swift's built-in control flow constructs: conditionals, loops, error handling, and `defer` statements all work like they do in synchronous code. However, `async/await` on its own does *not* introduce concurrency, i.e. multiple tasks executing simultaneously. For that, we need a way to create new tasks.

Tasks

A **task** is the fundamental execution context in Swift's concurrency model. Every asynchronous function is executing in a task (and so are the synchronous functions called by `async` functions). Tasks serve roughly the same purpose threads do in traditional multithreaded code. Like a thread, a task on its own has no concurrency; it runs one function at a time. When the running task encounters an `await`, it can suspend its execution, giving up its thread and yielding control to the scheduler in the Swift runtime. The scheduler can then run another task on the same thread. When it's time to resume the first task, the task will pick up exactly where it left off (possibly on a different thread).

Child Tasks vs. Unstructured Tasks

When we call an async function with await, the called function will run in the same task as the caller. Creating a new task always requires an explicit action. We can create two kinds of tasks:

- **Child tasks** — These form the basis of structured concurrency. We create a child task with one of the structured concurrency constructs: `async let` or task groups. Child tasks are organized in a tree and have scoped lifetimes. We'll discuss child tasks in detail below.
- **Unstructured tasks** — These are standalone tasks that become the root of a new independent task tree. We create an unstructured task by calling either the `Task` initializer or the factory method, `Task.detached`. The lifetime of an unstructured task is independent of the lifetime of the current task. We'll cover unstructured tasks in the [Unstructured Concurrency](#) section.

The Task Tree

The central idea of structured concurrency is to extend the concepts of structured programming (clear control flow and scoped lifetimes) to multiple tasks running concurrently. This is achieved by organizing tasks in a tree structure and imposing rules on the lifetimes of these tasks:

- **Child tasks run concurrently with each other** — The current task can create one or more child tasks that run in parallel, and the current task becomes the child tasks' parent. The parent task also runs concurrently with the child tasks.
- **Child tasks cannot outlive their parent task** — Each child task's lifetime is limited to the scope it's created in, just like the lifetime of a local variable ends at the end of its scope. The parent task can't exit the scope without waiting for all child tasks to complete, just like a synchronous function can't return until all functions it has called have returned.
- **Cancellation propagates downward from parent to children** — This ensures that a single cancellation cancels the complete task subtree below it, no matter how deep the tree is.

The result is that an async function can *temporarily* branch into several concurrently executing subtasks while limiting the concurrency to the current scope. In turn, the

author of the function can be certain that no dangling resources (running tasks) are being kept alive after the function has returned.

Child tasks also inherit their parent task's priority (unless explicitly overridden) and task-local values. Because the dependencies between tasks in the task tree are known, the scheduler can prioritize child tasks (and *their* child tasks, if any) if a higher-priority parent task is waiting for them.

async let

The `async let` syntax is the quickest way to create a child task. Continuing with the example from the [Async/Await](#) section, the following function loads the episodes and episode collections from the network *concurrently*:

```
// loadCollections has the same structure as loadEpisodes above.
```

```
func loadCollections() async throws -> [Collection] { ... }
```

```
func loadEpisodesAndCollections() async throws -> ([Episode], [Collection]) {
    async let episodes = loadEpisodes()
    async let collections = loadCollections()
    return try await (episodes, collections)
}
```

The `async let episodes = loadEpisodes()` syntax creates an **asynchronous binding**.

When execution reaches this line, the runtime creates a new child task and executes the `loadEpisodes()` call in this task. The child task starts running immediately. Meanwhile, the parent task continues running — note that we didn't have to write `await` even though `loadEpisodes` is an `async` function. In the following line, a second child task is launched to execute `loadCollections()`.

Next, the parent task awaits the `async` bindings to collect the results of the child tasks. This is the `try await (episodes, collections)` expression in our example, and it's here that the compiler requires us to write `await` (and `try`, if applicable) to acknowledge that the parent task may suspend until the child tasks have finished. In general, the parent task can perform other (synchronous or asynchronous) work before awaiting the child tasks, because parent and child tasks are running concurrently. In our example, there's nothing else for the parent task to do, so it immediately awaits the child tasks.

Let's cover the most important rules for asynchronous bindings:

Accessing an `async let` value for the second time won't suspend a second time. Once a parent task has awaited an `async` binding, its value is immediately available for subsequent accesses. Reading it again won't restart the child task and won't suspend the parent task (even though you have to write `await` in both places). For example, the following code only suspends once and prints the same number twice:

```
// IO-heavy random number generator, therefore async.
```

```
func requestRandomNumber() async -> Int { ... }
```

```
async let rand = requestRandomNumber()
await print(rand) // suspends
await print(rand) // doesn't suspend, doesn't rerun request
```

Async bindings don't have a separate identity in the type system. That is, unlike many other languages, Swift *doesn't* model `async let` bindings as futures or promises — the type of `async let` episodes is `[Episode]`, not `Future<[Episode]>` or something like `episodes: async [Episode]`. This is an intentional restriction: if child tasks returned `Future` values, programmers could pass these futures around, potentially letting them escape the current scope, which would defeat the lifetime guarantees made by structured concurrency.

Unawaited `async` lets will be implicitly canceled and awaited at the end of the scope. The requirement to await unawaited child tasks follows from the structured concurrency guarantee that child tasks can't outlive their parents. Any child task that hasn't completed must be allowed to finish before the parent task can exit the scope in which it created the child task. Here's an example of a program that navigates to one of two pages, depending on user input. To minimize wait times, we start two child tasks to preload the contents of both pages while waiting for the user's action (in a third child task). When the user input arrives, we choose to await only one of the two child tasks:

```
func waitForUserInput() async -> NavigationAction { ... }
func loadPage(at pageIndex: Int) async -> Page { ... }
func navigate(currentPage: Page) async -> Page {
    async let nextAction = waitForUserInput()
    async let nextPage = loadPage(at: currentPage.index + 1)
    async let previousPage = loadPage(at: currentPage.index - 1)
    switch await nextAction {
        case .nextPage:
            return await nextPage
        // Implicitly cancels and awaits previousPage
```

```
case .previousPage:  
    return await previousPage  
    // Implicitly cancels and awaits nextPage  
}  
}
```

In each branch of the switch statement, the unawaited child task will be implicitly canceled *and* awaited before the function returns. Note that cancellation doesn't necessarily mean that the child task completes quickly, because cancellation, like suspension, is cooperative in Swift's concurrency model. We'll have more to say about cancellation later in this chapter. By the way, task groups (which we'll cover in the next section) exhibit subtly different cancellation behavior when exiting their scopes: a task group won't implicitly cancel unawaited child tasks on normal (i.e. non-throwing) exit.

An `async` binding creates a single task, even if it contains multiple `async` calls. The entire expression on the right side of the equals sign is wrapped in a task. For example, the following code creates a single child task, which then executes the two `async` function calls *sequentially*:

```
// A single child task  
async let (num1, num2) = (requestRandomNumber(), requestRandomNumber())  
await print(num1) // Awaits num1 and num2.  
await print(num2) // num2 is already available.
```

The system always awaits a child task as a single unit, even if our code awaits only part of the child task. That is, `await num1` in the code above will suspend until the entire child task has completed, even if the result of `num1` is available earlier. Because of these semantics, `await num1` also makes `num2` available — a subsequent `await num2` will therefore not need to suspend again.

In contrast to the previous snippet, the following code executes the two function calls *concurrently* because they're in separate child tasks:

```
// Two concurrent child tasks  
async let num3 = requestRandomNumber()  
async let num4 = requestRandomNumber()  
await print(num3) // Awaits only num3.  
await print(num4) // Awaits only num4.
```

async let is a convenient, lightweight syntax when the number of child tasks is known at compile time. For a dynamic number of child tasks, we need another tool: task groups.

Task Groups

A task group provides a dynamic amount of concurrency, i.e. the number of child tasks is determined at runtime. As an example, this function downloads the poster images for an array of episodes concurrently:

```
func loadPosterImages(for episodes: [Episode]) async throws
    -> [Episode.ID: Data]
{
    let session = URLSession.shared
    return try await withThrowingTaskGroup(of: (id: Episode.ID, image: Data).self)
    { group in
        for episode in episodes {
            group.addTask {
                let (imageData, _) = try await session.data(from: episode.poster_url)
                return (episode.id, imageData)
            }
        }
        return try await group.reduce(into: [:]) { dict, pair in
            dict[pair.id] = pair.image
        }
    }
}
```

We create a task group with one of two functions — withTaskGroup(of:returning:body:) or withThrowingTaskGroup — depending on the throwing behavior of the child tasks (Swift's type system can't express these two cases as a single function). The withTaskGroup function takes both the result type for the child tasks and a closure that provides us with a TaskGroup instance. Note that calling withTaskGroup doesn't create a child task yet — the closure runs in the parent task.

We then call TaskGroup.addTask {...} (or addTaskUnlessCancelled) for each child task we want to create, usually in a loop. Child tasks begin executing immediately and in any order. The closure we pass to addTask returns a value that becomes the result of that child task, but note that we don't receive these results right away. Instead, the task group collects the results of all child tasks and provides them to us as an AsyncSequence. In

our example, we call `reduce` on the task group to store the results in a dictionary, but we could've used an asynchronous `for` loop just as well:

```
return try await withThrowingTaskGroup(of: id: Episode.ID, image: Data).self)
{ group in
    // ...
    var result: [Episode.ID: Data] = [:]
    for try await (id, imageData) in group {
        result[id] = imageData
    }
    return result
}
```

This structure of the task group closure — one loop to launch the child tasks, followed by another loop to collect or process their results — is a typical pattern. It resembles the popular [MapReduce algorithm](#) where a parent node divides its work into chunks and distributes the chunks to independent workers. As the workers deliver their results, the parent combines them into an aggregate result. We can recognize the structured concurrency paradigm of introducing concurrency selectively and containing its effects in this.

Consider how much more difficult it'd be to implement `loadPosterImages` using completion handlers. We're not showing the code here, but we put our attempt [on GitHub](#). Two major aspects of this version are tricky to implement correctly. First, we need a way to collect the results of the concurrently running `URLSession` tasks. When these “child tasks” complete and call their respective completion handlers, we add their results to a shared local array. We have to protect this array with a serial dispatch queue (or a lock) to prevent data races when two completion handlers try to access it simultaneously (we'll talk more about data races in the [Actors](#) section). Second, we need to delay calling the function's completion handler until all network tasks have completed. We do this by waiting for a `DispatchGroup` that tracks how many “child tasks” are still in progress. We can't wait synchronously because we don't want to block the current thread, so we have to dispatch the waiting to a separate dispatch queue. In our structured concurrency version, the task group handles all this manual bookkeeping and synchronization for us.

Next, let's cover the most important rules for task groups, like we did for `async let`:

Child task results are delivered in completion order, not in submission order. This is why our example returns a dictionary that maps from an episode ID to the image data

— and not just a plain array. Alternatively, the function could reorder the child task results as they come in from the task group.

Child tasks must all have the same result type. It has to be this way because TaskGroup must have a single element type when delivering the results. If you need to run child tasks with distinct result types, you have three options:

0. Use `async let` if the number of child tasks is static.
1. Use multiple task groups for child tasks of different element types.
2. Define a common type that can represent all results, e.g. an enum with one case per child task type.

Child tasks added to the group can't outlive the scope of the task group closure. If there are still unawaited child tasks when we exit the task group closure, the runtime will implicitly wait for them to complete (and discard their results) before continuing. Again, this follows from the structured concurrency rule that a child task's lifetime is bound to its scope.

Unlike `async let`, a task group won't implicitly cancel unawaited child tasks on exit (unless the task group throws an error). If we don't want this behavior, we must call `TaskGroup.cancelAll()` to manually cancel all remaining tasks. Not canceling by default makes it easier to use a task group for initiating Void-returning tasks, like so:

```
await withThrowingTaskGroup(of: Void.self) { group in
    for document in modifiedDocuments {
        group.addTask { try await document.save() }
    }
    // Implicitly awaits all child tasks.
}
```

This task group doesn't collect its child tasks' results (Void) at all, yet the group closure won't return until all child tasks have finished. Note that the group will silently discard any error thrown by a child task, so this isn't a good pattern for failable tasks. The following modified example checks for errors, but it has notably different semantics:

```
try await withThrowingTaskGroup(of: Void.self) { group in
    for document in modifiedDocuments {
        group.addTask { try await document.save() }
    }
}
```

```
// Explicitly await child tasks, catch errors.  
for try await _ in group {  
    // Do nothing.  
}  
}
```

The purpose of the for try await loop is to catch any error thrown by a child task and propagate it up. When the task group exits by throwing, it *will* implicitly cancel all remaining tasks, just like `async let`. If we don't want this, we should prevent errors from reaching the task group by handling them in the child task closures; handling them in the task group closure isn't sufficient, because `AsyncSequence` prescribes that iteration ends on the first error, so we'd miss any subsequent error.

Task groups don't throttle the amount of concurrency. If we call the `loadPosterImages` function with an array of 500 episodes, it'll launch 500 child tasks, potentially resulting in 500 simultaneous network requests. For optimum performance, it's probably best to limit the number of concurrent child tasks to a smaller number. Currently, `TaskGroup` doesn't provide an API to limit the "width" of its concurrency. We can implement this ourselves though: we start with a small number of child tasks and immediately await their results. Then, every time the task group delivers a result, we can start another child task until all inputs have been processed. Adding new tasks to the group while it's already producing results is valid and expected.

Sendable Types and Functions

The type of the closure passed to `TaskGroup.addTask` is `@escaping @Sendable () async -> ChildTaskResult`. The `@Sendable` attribute describes that this function is being passed across concurrent execution contexts — here, parent and child task — and must therefore be concurrency-safe. Sendable closures have some restrictions (checked by the compiler) when it comes to capturing state to make sure they don't introduce data races:

- Any captures must be sendable themselves. Types can declare themselves to be sendable by conforming to the `Sendable` marker protocol, which is an empty protocol with no requirements. Non-public structs and enums implicitly conform to `Sendable` if all of their components are sendable. Actors are also sendable by default because they're designed to protect their state against concurrent accesses.

For mutable or non-final classes, the compiler can't check if the class is safe to be shared across concurrency domains, so it rejects a plain Sendable conformance. We can still make such a class sendable by writing class C: @unchecked Sendable, which disables the compiler checks. Now we're responsible for making sure the class is thread-safe, e.g. by protecting all state accesses with a lock.

- All captures must be by value, whereas Swift's default is to capture state by reference. Captures of immutable values declared with let are implicitly by value; any other capture must explicitly be made by value via a capture list.

Note that the second rule precludes any *mutable* state capture. For instance, this alternative formulation of our task group code is illegal because the child task closures are capturing the parent task's mutable state:

```
return await withThrowingTaskGroup(of: (id: Episode.ID, image: Data).self)
{ group in
    var result: [Episode.ID: Data] = [:]
    for episode in episodes {
        group.addTask {
            let (imageData, _) = try await session.data(from: episode.poster_url)
            // Error: Mutation of captured var 'result' in concurrently-executing code
            result[episode.id] = imageData
        }
    }
    return result
}
```

Here, we're trying to mutate the result dictionary concurrently from the child tasks, which results in a data race. This is the same problem we discussed above for the completion handler-based version of the loadPosterImages function. However, unlike before, the @Sendable annotation on the closure enables the compiler to catch this mistake right away. We've already seen the correct way to write this: use a second loop to iterate over the child task results. Since this loop runs in the parent task, it's free to mutate the local state.

The same @Sendable restrictions apply to `async let`. You can think of the right side of an `async let` assignment as being wrapped in a @Sendable closure that runs in a different execution context and therefore can't access the non-sendable state of the enclosing context or mutate a captured variable.

In Swift 5.6, the compiler doesn't yet catch all possible Sendable violations. This is because existing libraries (including Apple's frameworks) haven't yet been audited and

annotated for concurrency safety, so the compiler can't know what types are safe to pass across concurrency domains. If any usage of e.g. Date or Data in a sendable function triggered an error because of a missing annotation in Foundation, it'd be almost impossible for developers to adopt `async/await`.

The `-Xfrontend -warn-concurrency` compiler flag we mentioned in the introduction of this chapter enables stricter compiler diagnostics for `Sendable` violations. We found the additional warnings useful for understanding when an `async` function runs in a different concurrency context and how the compiler prevents data races. We think it's a good idea to enable the warnings — at least temporarily while writing `async` code, even if you decide to disable them again due to too much noise. It's also possible to retroactively make a third-party type `sendable` with an `unchecked` extension, like so:

```
// Should be safe because Date is a plain value.  
extension Date: @unchecked Sendable {}
```

But note that this is inherently unsafe because we're telling the compiler to disable concurrency checking for this type. It can be a workable makeshift solution, however, to silence the compiler.

At the time of writing, the whole topic of `Sendable` checking is still in flux. Eventually, in Swift 6, `Sendable` violations will become hard errors. For the transition period (which could take years, as we need to wait until all libraries have been audited), there exists a [migration path](#) that allows both module authors to make their code compatible with pre- and post-concurrency worlds, and clients to selectively silence warnings on a per-module basis. Clients can use the `@preconcurrency import A` directive to suppress concurrency warnings for types from module A. The compiler is smart enough to resurface the warnings once module A is updated with `Sendable` annotations.

Cancellation

Being able to cancel an asynchronous operation is a common requirement for almost any program. Yet, like error handling, implementing cancellation is difficult in traditional completion handler-based code. Take the `loadFirstPosterCont` function from the [Async/Await](#) section as an example. Here's what we'd need to change to support cancellation:

- Provide a way for the caller to signal that the operation has been canceled. We could do this by returning a thread-safe “cancellation token” that provides a public `cancel` method and allows us to query the cancellation state. The

`URLSessionDataTask` object itself isn't suitable — it does have a `cancel` method, but once the network task has completed, it can no longer be canceled, so we can't use it to cancel the inner download task reliably.

- When the cancellation token receives a cancellation request, cancel the URL session operations. This is usually done by passing a closure that runs on cancellation to the cancellation token. Additional bookkeeping is required to pass the inner download task to this cancellation handler.
- Take care not to start new tasks after the operation has been canceled. In our example, we should insert a manual check for cancellation after the (potentially time-consuming) JSON decoding step and return early if needed.
- Make sure to return a suitable error code to the caller on cancellation. `URLSession` completes with the error code `URLError.Code.cancelled` when canceled; we'd have to come up with our own error for the manual cancellation check and possibly transform the URL error into something more appropriate for our needs.

The resulting code is messy, making the intent of the code hard to understand because a large proportion is boilerplate to deal with cancellation and errors.

Compare this to the corresponding async code, which supports cancellation out of the box:

```
func loadFirstPoster() async throws -> Data {  
    let session = URLSession.shared  
    let (data, _) = try await session.data(from: Episode.url)  
    let episodes = try JSONDecoder().decode([Episode].self, from: data)  
    return try await session.data(from: episodes[0].poster_url).0  
}
```

When the task this function is running in gets canceled, the function will abort with an error. We'll take a closer look at how this works below.

Cancellation Is Cooperative

We saw earlier that Swift's concurrency system is cooperative, and this applies to cancellation as well: canceling a task has no effect unless the code that's running in the task periodically checks if it has been canceled and finishes early if necessary. It's an important design goal of the system to give functions the chance to clean up after themselves on cancellation. This is why the system can't just terminate a canceled task.

The unit that's being canceled is always a task, and not a particular function in that task. If you have access to the Task handle of an unstructured task, you can call its `cancel` method to cancel it. It's also possible to cancel the current task with this piece of code:

```
withUnsafeCurrentTask { task in
    task?.cancel()
}
```

At its most basic level, canceling a task just sets a flag in the task's metadata. A function that wants to support cancellation should occasionally check `Task.isCancelled` or call `try Task.checkCancellation()` — the latter aborts directly with a suitable error if `isCancelled` is true. Adding these checks to our example function could look like this:

```
func loadFirstPoster2() async throws -> Data {
    try Task.checkCancellation()
    let session = URLSession.shared
    let (data, _) = try await session.data(from: Episode.url)
    try Task.checkCancellation()
    let episodes = try JSONDecoder().decode([Episode].self, from: data)
    try Task.checkCancellation()
    return try await session.data(from: episodes[0].poster_url).0
}
```

This isn't too bad, but it's not even necessary in this example because the `URLSession` APIs we're calling already perform similar cancellation checks internally. If the task is canceled, the currently active download will stop and throw an error, which our function passes on to the caller. Depending on how big the data models we expect to download are, the JSON decoding step between the two network requests can be problematic because `JSONDecoder` currently doesn't support cancellation. So if JSON decoding takes, say, 500 milliseconds, our function could continue running for that duration after receiving the cancellation signal, which isn't ideal. To fix this, we'd have to use a JSON decoder with cancellation support, or write our own.

It's worth noting that `JSONDecoder` *could* support cancellation even though it's a fully synchronous API. Synchronous functions can use the same `Task.isCancelled` or `try Task.checkCancellation()` calls to check for cancellation — they'll do the correct thing if the synchronous function is nested inside an `async` context, and return default values (`false`) otherwise.

In summary, in the common case where we write a function that spends most of its time calling other (system) APIs that already handle cancellation, we get cancellation support

essentially for free, as long as we pass cancellation errors on to our caller. However, if we perform long-running computations or call multiple non-cancelable APIs in sequence, we should add manual cancellation checks to our code. For instance, the following function computes a potentially large amount of random numbers. We add a cancellation check on every one-thousandth iteration through the loop:

```
func makeRandomNumbers(in range: ClosedRange<Int>, count: Int) throws
    -> [Int]
{
    var result: [Int] = []
    result.reserveCapacity(count)
    for i in 1...count {
        // Check for cancellation periodically.
        if i.isMultiple(of: 1000) {
            try Task.checkCancellation()
        }
        result.append(Int.random(in: range))
    }
    return result
}
```

If this were an async function, we'd also want to periodically call `Task.yield()` in the same fashion to give the runtime the chance to schedule another task on our thread.

Cancellation Uses the Error Path

By convention, a function that detects it's been canceled should throw a `CancellationError`, which is a new standard library type. Callers can check for this error to distinguish cancellation from other errors. Using errors for cancellation has the benefit that cancellation doesn't introduce new control flow paths, but note that the specific error is only a convention — callers can't rely on it. For example, `URLSession` continues to throw its customary `URLError.Code.cancelled` error on cancellation. If we don't want to expose the `URLSession` behavior to our callers, we can catch the "wrong" error and replace it with a `CancellationError`:

```
func loadFirstPoster3() async throws -> Data {
    do {
        let session = URLSession.shared
        let (data, _) = try await session.data(from: Episode.url)
        let episodes = try JSONDecoder().decode([Episode].self, from: data)
```

```
    return try await session.data(from: episodes[0].poster_url).0
} catch URLError.Code.cancelled {
    // Replace URLError.Code.cancelled.
    throw CancellationError()
} catch {
    // Propagate all other errors.
    throw error
}
}
```

Non-throwing functions can also participate in cancellation. They can't throw an error, of course, so authors must choose an appropriate "empty" return value, such as nil. In most cases, it's probably better to make the function throwing instead. Canceled functions are even allowed to return a *partial* result — for example, `makeRandomNumbers` could abort by returning the partially filled array up to the point when it detects it's been canceled. If you decide to do this, make sure to document the behavior clearly, because callers may not expect to receive partial results.

Cancellation and Structured Concurrency

So far, we've looked at cancellation in a single task, but the real power of the cancellation system comes from its integration with structured concurrency. Cancellation signals automatically flow down the task tree: when a parent task is canceled, its child tasks will see their `isCancelled` flags switch to true. Again, this won't stop the child tasks immediately, as each task is responsible for performing periodic cancellation checks on its own. In addition, cancellation must respect the fundamental rule of structured concurrency that child tasks can't outlive their parent — the canceled parent task will keep running until all of its child tasks have completed.

Propagating cancellation down the task tree means that canceling a top-level task (e.g. in response to a user action) will tell all child tasks working on that operation to abort, no matter how deeply they're nested — without a single line of code. It's immensely powerful, but it only works if we use structured concurrency (unstructured tasks must be canceled manually) and if we design our functions with cancellation in mind.

As another example of how important it is to design code for cancellation, let's write a function that runs an async operation with a timeout:

```
func asyncWithTimeout<R>(
    nanoseconds timeout: UInt64,
```

```

do work: @escaping () async throws -> R
) async throws -> R {
    return try await withThrowingTaskGroup(of: R.self) { group in
        // Start actual work.
        group.addTask { try await work() }
        // Start timeout timer.
        group.addTask {
            try await Task.sleep(nanoseconds: timeout)
            // We've reached the timeout.
            throw CancellationError()
        }
        // First one to the finish line wins, cancel the other task.
        let result = try await group.next()!
        group.cancelAll()
        return result
    }
}

```

And here's a usage example that loads an episode image with a timeout of one second — if the download takes longer, the function will abort with a `CancellationError`:

```

let imageData = try await asyncWithTimeout(nanoseconds: 1_000_000_000,
    do: loadFirstPoster3)

```

In `asyncWithTimeout`, we use a task group to start two concurrent child tasks: one for the actual work, and one that sleeps for the desired timeout duration. We then wait for the first child task to finish. If the work task “wins,” we’ll return its result and cancel the timeout task. If the timeout task is faster, `group.next()` will throw a `CancellationError`. Since the task group exits with an error, it’ll implicitly cancel the unfinished work task.

The `group.cancelAll()` call after retrieving the first result from the group is important. Recall that task groups don’t implicitly cancel unawaited child tasks on successful completion. Without `cancelAll()`, the task group would silently wait for the timeout to run down before returning. Forgetting to add this line is an easy mistake to make, because it doesn’t change the output of your program — only its performance. In fact, the authors of the Swift Evolution proposal for structured concurrency made this mistake several times.

The `asyncWithTimeout` function also illustrates how important it is that functions support the cooperative cancellation model. If the timeout “wins,” the task group will

cancel the work task, but it then still has to wait for the work task to return. If the work task doesn't check for cancellation (like the `JSONDecoder` example above) and takes a long time to complete, the `asyncWithTimeout` function could take a lot longer than the specified timeout duration to complete.

Cancellation Handlers

The `withTaskCancellationHandler(operation:onCancel:)` function lets us install a cancellation handler for the current task. The function takes two closures: an `async` operation (the work) that runs in the current task; and the cancellation handler, which is invoked *immediately* when the current task is canceled. The purpose of the handler is to run cleanup code on cancellation, e.g. for releasing resources or for passing the cancellation signal on to other objects that don't integrate with Swift's concurrency system.

For example, it's likely that the Foundation framework uses a cancellation handler in its implementation of the `async/await`-based `URLSession` API. In this case, the `onCancel` closure's task would be to forward the cancellation event to the `URLSession` system so that the network request can be canceled.

Implementing this is trickier than it sounds, because the cancellation handler runs in a different concurrency context than the task that's being canceled. Specifically, the cancellation handler and the main operation can't access shared mutable state (the `onCancel` closure is marked `@Sendable` to make the compiler aware of this). We need to use a class with manual synchronization (a lock or dispatch queue) to share the `URLSessionDataTask` instance between the two contexts in a thread-safe manner. We can't use an actor, because neither the cancellation handler nor the continuation closure are `async`. The code is too long to print here, but you can check it out on [GitHub](#).

Unstructured Concurrency

At times, the defining feature of structured concurrency — confinement to a scope — becomes a hindrance. We need another tool that allows us to opt out of structured concurrency and start a task that can outlive the current scope. This tool is the `Task.init(priority:operation:)` initializer. We usually call this initializer with a trailing closure, like this:

```
let task = Task {  
    return try await loadEpisodesAndCollections()  
}
```

Calling `Task.init` launches a new, independent task whose lifetime isn't bound to the current scope. The task starts running immediately, concurrent with the originating task. The new task becomes the root of a separate task tree, which will exist until the task completes. Because there's no parent-child relationship between the origin task and the new task, we call this **unstructured concurrency**.

An unstructured task won't be canceled when its origin task is canceled. The `Task` instance returned by the initializer allows us to manually cancel the task or retrieve its result using [try] `await task.value` (the try is only required when the task closure can throw). The `value` property is `async` because the task may not have completed when we retrieve its result. A `Task` instance is Swift's version of a future or promise object.

Note that launching an unstructured task from a structured concurrency context (e.g. inside a task group) is often an anti-pattern because it allows a part of an operation to escape the structured concurrency world. That's not to say that creating an unstructured task is never a good idea — it's clearly necessary for concurrent work that must outlive the current scope. But the cost of losing automatic cancellation, error propagation, and lifetime management is substantial. Furthermore, we pay for unstructured tasks with loss of clarity, as the task hierarchy is no longer apparent from the code structure. The constraints imposed by structured concurrency are purposefully designed to make our code easier to understand; we shouldn't break out of them unless we absolutely need to.

The Swift runtime keeps a strong reference to all tasks until they complete, so we do *not* need to store the task instance just to keep the task alive. Also, the closure passed to `Task.init` doesn't require explicit `self....` annotations when strongly capturing `self`. This is achieved through a new, unofficial compiler attribute, `_implicitSelfCapture`. The rationale for not requiring `self` is that most tasks won't create a permanent reference cycle because they'll complete eventually, at which point they'll release their captured variables. If we create a task that potentially never finishes (e.g. a `NotificationCenter` observer that publishes incoming notifications as an `AsyncSequence`), we should take care to capture variables weakly if they can cause reference cycles.

Unstructured vs. Detached Tasks

Unstructured tasks launched with Task { ... } inherit the origin context's priority (unless overridden), task-local values, and — importantly — actor isolation. So if the function that launches the task is running on a specific actor, the new task will also be isolated to that actor. This is often desirable because it gives the unstructured task access to the surrounding actor-isolated state, like in this contrived example:

```
@MainActor class ViewModel: ObservableObject {  
    @Published var counter = 0  
  
    func incrementAsync() {  
        Task {  
            // Task can access actor-isolated state.  
            counter += 1  
        }  
    }  
}
```

We'll say more about actor isolation in the [Actors](#) section later in this chapter. Note that if the task closure calls other async functions with a different actor isolation (e.g. from a @MainActor annotation), the task will switch to the other actor. "Normal" async functions without any actor affiliation generally run on whatever the current actor context is, but the exact behavior hasn't been specified. At the time of writing, the Swift team is working on [formalizing the rules](#) so that it's always statically known in which context a function runs.

If we don't want the new task to run in the current execution context, we can launch a **detached task** using Task.detached { ... } instead. A detached task works much like an unstructured task and has the same API, but it doesn't inherit the current task's priority, task-local values, or actor isolation.

You may wonder when you should use a detached task rather than an unstructured one. Our advice is to treat Task { ... } as the default choice and only use Task.detached if you have a specific reason to leave the current actor context. One such reason is if you're on the main actor (more on that [later](#)) and want to launch an operation that isn't supposed to occupy the main thread.

In the remainder of this chapter, we'll use "unstructured task" as a generic term for both unstructured and detached tasks. We can ignore the distinction unless we specifically discuss actor isolation.

Starting Async Work from Synchronous Contexts

Unstructured tasks can be launched from any context — even from a non-async function. In fact, `Task { ... }` and `Task.detached { ... }` are the only ways to call an async function from a synchronous context. We sometimes say that the `async` attribute is “contagious,” because making a function `async` requires all of its callers to become `async` too.

Every `async` function that’s being executed must have an unstructured task at the root of the current task tree. This task may be hidden in a third-party library or the Swift runtime, but it’s there. Take this example of one of the simplest `async/await` programs we can write:

```
@main struct Program {  
    static func main() async throws {  
        print("Sleeping for one second")  
        try await Task.sleep(nanoseconds: 1_000_000_000)  
        print("Done")  
    }  
}
```

When we declare the `main()` function of an executable to be `async`, the compiler generates code that calls the `_runAsyncMain` function inside the Swift runtime. If you look at [the source code for this function](#), you’ll find a call to `Task.detached` in there.

Error Handling for Unstructured Tasks

As we’ll discuss in detail in the [Error Handling](#) chapter, one of Swift’s fundamental design goals is to make it hard for programmers to ignore errors — we must either handle an error on the spot, or explicitly forward it to our caller, which then has to handle or propagate it, and so on. Unfortunately, the unstructured task API opens a new loophole that makes it too easy to ignore errors in certain cases. As an example, consider the following unstructured task that saves the program’s current state to disk:

```
Task {  
    try await save()  
}
```

If `save` throws an error, that error will be silently discarded because no one ever awaits this task's result. Ideally, the compiler should recognize this and force us to handle the error case. We don't even get a warning because the Task initializer is declared with the `@discardableResult` attribute, which allows us to discard its return value. Without `@discardableResult`, we'd at least be forced to write `_ = Task { ... }`, making the fire-and-forget nature of the task explicit.

Actors

Concurrent programming has to deal with the inherent problem of potential conflicts over shared resources, no matter if you use old-school threads, GCD queues, or Swift tasks. Whenever code is executed concurrently (in a time-shared manner or in parallel), you have to make sure to protect shared resources, such as an object's properties, memory in general, files, or a database connection.

Traditionally, we've used a whole variety of locking APIs for this purpose, and more recently, GCD's serial queues have become a popular way to ensure conflict-free access of shared resources. With the introduction of Swift's native concurrency model, the language gained a new mechanism for resource isolation: actors.

Resource Isolation

An actor is a reference type that protects its state by only allowing access to its mutable properties directly on `self` (exception: let constants can be safely accessed across actor boundaries), and by isolating the execution of its function parts on a serial execution context. It's important to keep in mind that an actor doesn't isolate the execution of entire methods: the general rule is that only the function parts between suspension points can be considered atomic operations. In the special case of a method that doesn't call other async functions with `await`, the method runs as a single job and becomes an atomic operation (we'll talk more about this in the [Reentrancy](#) section).

The actor's access model provides resource isolation to prevent **data races**. A [data race](#) occurs when two threads try to access the same location in memory at the same time and at least one of them is a write operation. This can lead to reading non-sensical data, e.g. if the reading thread reads from the memory in question halfway through the write operation of the other thread. Even worse, if two threads write to the same memory location at the same time, data corruption can be the consequence.

Actors provide protection against data races on their properties. The access model of an actor as described above strictly prevents a property from being accessed simultaneously by two threads, no matter if it's for reading or writing. However, note that using actors won't make all your concurrent code magically correct — data races are just one type of the broader category of **race conditions**.

Race conditions occur when the correct behavior of our code depends on the execution sequence of multiple threads. In the context of an actor, we can say that a race condition occurs when the proper behavior of the actor depends on the execution sequence of multiple isolated function parts.

While the isolated nature of each function part prevents data races as described above, we don't have control over the execution sequence of multiple function parts across suspension points. For example, modifying an actor's property before an await and relying on this property to still have the same value after the await is a race condition. The consequences of a race condition depend on the specifics of the situation, but it could result in unintended behavior or even data corruption (on a higher level than an actual memory location being corrupted, as in the case of a data race).

Bugs resulting from race conditions — also called heisenbugs — are particularly difficult to debug due to their non-deterministic nature. Therefore it's important to be aware of the possibility of race conditions, even in actors, and to avoid them by careful design. We'll discuss the reason why an actor doesn't protect you from race conditions in the Reentrancy section.

With this basic understanding of the safety that actors do and don't provide, let's look at an example. Imagine we want to implement a queue for a web crawler that looks like this:

```
class CrawlerQueue {  
    var pending: [URL] = []  
    var finished: [URL: String] = [:]  
}
```

Let's suppose that the web crawler uses several concurrent workers to process the pending URLs, and we want to share the crawler queue across all workers. A worker should be able to retrieve a pending URL from the queue, append newly found URLs to the queue, and store the crawl result for a processed URL. This makes the `CrawlerQueue` instance with its two properties, `pending` and `finished`, a shared resource among the concurrent workers that has to be protected from data races.

Using dispatch queues, we might have implemented the necessary synchronization like this:

```
class CrawlerQueue {  
    private var pending: Set<URL> = []  
    private var finished: [URL: String] = [:]  
  
    private var queue = DispatchQueue(label: "crawler-queue")  
  
    public func getURL() -> URL? {  
        queue.sync {  
            self.pending.popFirst()  
        }  
    }  
  
    public func enqueue(_ url: URL) {  
        queue.async {  
            guard self.finished[url] == nil else { return }  
            self.pending.insert(url)  
        }  
    }  
  
    public func store(_ contents: String, for url: URL) {  
        queue.async {  
            self.finished[url] = contents  
        }  
    }  
}
```

All properties are now private to the queue and can only be accessed through public methods, which use a private serial dispatch queue to prevent data races on the pending and finished properties.

The actor implementation of the queue looks similar, minus the code to dispatch onto the private serial queue:

```
actor CrawlerQueue {  
    var pending: Set<URL> = []  
    var finished: [URL: String] = [:]  
  
    public func getURL() -> URL? {
```

```

    pending.popFirst()
}

public func enqueue(_ url: URL) {
    guard finished[url] == nil else { return }
    pending.insert(url)
}

public func store(_ contents: String, for url: URL) {
    finished[url] = contents
}
}

```

The pending and finished properties can only be accessed in the actor's methods by default (the properties are **isolated** to the actor's execution context), and each actor method is executed on the actor's serial executor, which ensures we don't run into data races on its properties.

Reentrancy

Instead of providing separate methods on CrawlerQueue to fetch a URL, enqueue new URLs, and store the result for a URL, let's provide a unified API for all these tasks so that we can't accidentally forget one of the calls:

```

actor CrawlerQueue {
    var pending: Set<URL> = []
    var finished: [URL: String] = [:]

    func process(_ handler: (URL) async -> (contents: String, links: [URL])) async {
        guard let url = pending.popFirst() else { return }
        let (contents, links) = await handler(url)
        finished[url] = contents
        for link in links {
            guard finished[link] == nil else { continue }
            pending.insert(link)
        }
    }
}

```

The `process` method gets called with an `async` function as its parameter. This function takes the URL to process, and it returns the result in the form of a string and an array of new URLs to be added to the queue.

`process` is explicitly marked as `async`, because otherwise we wouldn't be able to call the `async` handler with `await`. Once we call `await handler(url)`, Swift has two imperfect alternatives of how to proceed: either it can allow other code to run on the actor while the result of `handler` is being awaited, or it can prevent other actor code from running as long as the `handler` hasn't returned.

In the latter case, we could easily run into deadlocks. For example, it'd be enough for the `handler` function to call any other method on the `CrawlerQueue` actor to bring the execution to a halt. For this reason, Swift chooses reentrancy as the default behavior for actors: once the current job is suspended (when we call `await handler(url)`), other code might run on the actor before the job resumes. If the `handler` function were to make another call to the actor internally, code execution can still continue.

With reentrancy being the default, it's important to make sure not to rely on assumptions of the actor's state being the same before and after a suspension point. Likewise, we can't assume that an `async` method is executed as an atomic operation. For example, our job queue is in a peculiar state while `process` waits for the `handler` to finish: the URL that's being processed has been removed from the set of pending URLs, but it hasn't been added to the dictionary of finished jobs. Therefore, we have a window of opportunity for the same URL to be added to the pending URLs again, although it's currently being processed.

This problem can only manifest itself because the actor's `process` method is reentrant — if actor methods were non-reentrant, no other code on the actor could interfere while the `process` method executes, and we wouldn't have to worry about the state of the actor changing while waiting for a suspension point.

To remedy this situation, we can add a third property to the queue to keep track of the URLs currently being processed:

```
actor CrawlerQueue {
    var pending: Set<URL> = []
    var inProcess: Set<URL> = []
    var finished: [URL: String] = [:]

    func process(_ handler: (URL) async -> (String, [URL])) async {
        guard let url = pending.popFirst() else { return }
    }
}
```

```

inProcess.insert(url)
let (contents, links) = await handler(url)
finished[url] = contents
inProcess.remove(url)
for link in links {
    guard finished[link] == nil, !inProcess.contains(link) else { continue }
    pending.insert(link)
}
}
}

```

Once a URL is in the queue, it's always being tracked one way or another — either as pending, in process, or finished. Therefore, `process` can now safely be executed in a reentrant fashion.

Actor Performance

To the outside world, all public actor methods are `async` methods: they have to be called using `await` to allow for a switch to the actor's own execution context. For example, the function below calls two actors sequentially. We say that the task **hops** between actors:

```

// Counter is an actor.
let counter1 = Counter()
let counter2 = Counter()

func incrementAll() async {
    await counter1.increment()
    await counter2.increment()
}

```

An actor hop necessarily has some overhead, but Swift's concurrency system is designed to make actor hopping much faster than switching threads or dispatch queues, so you don't have to worry too much about the performance cost. Since the task that calls an actor method (the task that executes the `incrementAll` function in our example) has to wait for the actor method to complete, the actor can run on the task's current thread. If the actor isn't busy at the time of the call, the actor hop consists of setting a flag in the actor that "locks" it for other callers and then executing a normal function call. When the method returns, the actor is "unlocked" again. The context switch becomes more expensive under **contention**, i.e. if the actor is currently executing some other task. In that case, our task must suspend and give up its thread while it waits for the actor to

become available. You'll get the best performance if you can design your program in such a way that actors can be expected to be uncontended most of the time.

The cost of actor hopping also increases for switches to and from the main actor. Because the main actor runs on the main thread, hopping from another execution context (which generally runs on a thread of the cooperative thread pool) to the main actor or vice versa also involves a relatively costly thread switch.

The Main Actor

Sometimes it's not necessary to create your own actor to synchronize the access of some state. Especially with GUI applications, you often just want to make sure that a particular method gets called on the main thread or that a specific property is only being accessed from the main thread. Since the main thread (or the main queue in GCD terms) has always had a special role on Apple's platforms, it's now also exposed in the form of a global actor, the `MainActor`. Under the hood, the main actor uses the main thread as its serial execution context.

The special thing about a global actor is that it can be used as an attribute to annotate other types, properties, or functions. For example, instead of defining the `CrawlerQueue` as an actor itself, we could also use the system-defined `@MainActor` attribute to make sure its methods and properties are only being accessed on the main thread:

```
@MainActor
final class CrawlerQueue {
    var pending: Set<URL> = []
    var inProcess: Set<URL> = []
    var finished: [URL: String] = [:]

    func process(_ handler: (URL) async -> (String, [URL])) async {
        // ...
    }
}
```

Of course, this implementation of the crawler queue will do more work on the main thread compared to its actor-based counterpart, but especially for publishing changes to the UI, this technique is very useful.

Marking an entire type as `@MainActor` is shorthand for marking all properties and methods individually. The example above is semantically equal to the following:

```
final class CrawlerQueue {  
    @MainActor var pending: Set<URL> = []  
    @MainActor var inProcess: Set<URL> = []  
    @MainActor var finished: [URL: String] = [:]  
  
    @MainActor func process(_ handler: (URL) async -> (String, [URL])) async {  
        // ...  
    }  
}
```

Sometimes you run into the situation where you'd want to mark an entire type as `@MainActor`, with just one or two exceptions. Instead of typing all the annotations manually, we can also take the inverse approach: annotate the type as `@MainActor`, and then use the `nonisolated` keyword to opt out of the main actor isolation for specific methods or properties. For example, we could use this technique to add a non-isolated initializer to the otherwise MainActor-isolated crawler queue:

```
@MainActor  
final class CrawlerQueue {  
    nonisolated init(_ initialURLs: [URL]) {  
        self.pending = Set(initialURLs)  
    }  
    // ...  
}
```

In most cases, the `@MainActor` attribute does what you'd expect: it ensures that the annotated method or property is only accessed on the main thread. However, there are some subtle edge cases that are worth thinking about. Specifically, there are three distinct scenarios we have to look at: annotating an `async` method with `@MainActor`, annotating non-`async` methods, and annotating properties (which you can think of as non-`async` getters and setters).

0. **Async `@MainActor` methods:** When we annotate an `async` method as `@MainActor`, we can be sure that the code of this method runs on the main thread. The compiler forces us to call the `async` method with `await`, thus giving it the opportunity to switch to the main actor's execution context. This will even work if such a method is called from Objective-C code, because `async` functions are exposed as functions with completion handlers, allowing the runtime to switch the execution context.

1. **Non-async @MainActor methods:** When we annotate a non-async method as @MainActor, the situation is a bit more complicated. The compiler will still force us either to not call this method from anything but the main actor's execution context, or to call it with await from any other context. However, since the method is synchronous, it cannot be dispatched onto the correct execution context at runtime. All checks have to be performed at compile time, and they can be defeated e.g. from Objective-C code.

One common example of this problem occurs when we conform a class to a protocol and then mark a protocol method as @MainActor. We might assume that this ensures the method always runs on the main actor, but that's not necessarily the case. For example, the delegate methods of URLSession are being called on a private queue of the session. Annotating such a delegate method as @MainActor has no effect with regard to the thread it's running on, and the compiler isn't able to warn us about this.

2. **@MainActor properties:** @MainActor annotations on properties are also a compile-time check. If we try to access a main actor-isolated property from another execution context, the compiler will prevent us from doing so. Unlike main actor-isolated synchronous method calls, we can't insert an await before the property access to allow for a context switch. We have to make sure to already be on the main execution context; otherwise, the compiler will throw an error. Just like with synchronous methods, it's important to keep in mind that the compiler checks around main actor-isolated properties can be defeated.

Reasoning about Execution Contexts

When you read Swift code that uses GCD for concurrency, you reason about which code is executed on which queue essentially by executing the code “in your head,” tracing its path from dispatch to dispatch at runtime. Additionally, you'll have to consult the documentation of Apple-provided or third-party APIs to determine if completion handlers or delegate methods are being called on the main queue or some other queue.

Within this model, which queue you're currently on is determined at runtime. If you look at any particular piece of code, you cannot say for certain on which queue it'll run when the code executes (at least if the code you're looking at doesn't do its own dispatching right at that point).

With Swift's model of actor-isolated functions, the reasoning process no longer depends solely on runtime behavior. You can determine on which actor's execution context an async function will run by inspecting its lexical scope. If a method is defined within an

actor, you know that this code will run isolated on that actor's execution context, no matter what happens around it at runtime. Similarly, any `async` function that's marked as being actor-isolated with e.g. the `@MainActor` attribute, is guaranteed to run isolated on that actor as well. If the `async` function you're looking at isn't explicitly actor-isolated in any of these two ways, it'll be executed on a generic executor not associated with any actor (eventually, as detailed in [this proposal](#)).

However, if you're looking at a synchronous function, the execution context is still determined at runtime: you have to trace back its potential call stacks to the nearest `async` functions to know which execution context it might run on.

Recap

The concurrency model is the biggest change to the language since the introduction of protocol extensions in Swift 2. `Async/await` lets us write asynchronous code in the same style as synchronous code, using the language's normal control flow constructs.

Structured concurrency applies the 60-year-old insights from structured programming to concurrent code, and actors prevent data races by serializing access to memory or other resources.

None of these concepts on their own are new, but Swift combines them into one coherent concurrency model and deeply integrates them into the language. This allows the compiler to check for many common concurrency mistakes, hopefully resulting in fewer bugs and safer programs.

As we write this, Swift 5.5 is only a few months old. Some things are still in flux, and we naturally don't have much experience using the new concurrency model in production yet. As we worked with the new features while writing this chapter, we continually learned new things. We think it's fair to assume that it'll take us another year or two of real-world use to *really* understand the new model. We feel confident enough that what we wrote in this chapter is correct and solid advice, but we're also certain that our understanding will continue to evolve.

Error Handling

13

As programmers, we constantly have to deal with things going wrong: the network connection may go down, a file we expected to exist may not be there, and so on. Handling failures well is one of those intangible factors that distinguishes good programs from bad ones, and yet we often tend to see error handling as a subordinate task — something to be added later (and which then often gets cut as a deadline looms).

And we get it: error handling can be messy, and coding for the happy path is usually more fun. Therefore, it's all the more important that a programming language provides a good model that supports programmers in this task. Here are some of the things Swift's built-in error handling architecture with throw, try, and catch provides:

- **Safety** — Swift makes it impossible for programmers to ignore errors accidentally.
- **Conciseness** — The code for throwing and catching errors doesn't overwhelm the code for the happy path.
- **Universality** — A single error throwing and handling mechanism can be used everywhere, including in asynchronous code. The new `async/await` pattern for asynchronous functions fully supports `throw/try`-based error handling. (Before `async/await`, the common idiom of using callbacks for concurrency didn't integrate with the language's native error handling approach at all.)
- **Propagation** — An error shouldn't have to be handled where it occurred, because the logic for recovering from an error is often far removed from the place where the error originated. Swift's error handling architecture makes it easy to communicate errors up the call stack to the appropriate level. Intermediate functions (functions that call throwing functions, but neither throw nor handle errors themselves) can pass errors through without requiring big syntax changes.
- **Documentation** — The compiler enforces both throwing functions and their call sites to be annotated, making it easy for programmers to see where errors can occur. The type system does *not* expose *which* errors a function can throw, though.

We'll revisit these points throughout the chapter.

Error Categories

The terms “error” and “failure” can mean all sorts of things. Let's try to come up with some categories of “things that can go wrong,” differentiated by how we commonly handle them in code:

Expected errors — These are failures the programmer expects (or should expect) to happen during normal operation. These include things like network issues (a network connection is never 100 percent reliable), or when a string the user has entered is malformed. We can further segment expected errors by the complexity of the failure reason.

→ **Trivial errors** — Some operations have exactly one expected failure condition. For example, when you look up a key in a dictionary, the key is either present (success) or absent (failure). In Swift, we tend to return optionals from functions that have a single clear and commonly used “not found” or “invalid input” error condition. Returning a rich error value wouldn’t give the caller more information than what’s already present in the optional value.

Assuming the failure reason is obvious to the caller, optionals perform well in terms of conciseness (partly thanks to syntactic sugar for optionals), safety (we have to unwrap the value before we can use it), documentation (functions have optional return types), propagation (optional chaining), and universality (optionals are ubiquitous).

→ **Rich errors** — Networking and file system operations are examples of tasks that require more substantial error information than “something went wrong.” There are many different things that can fail in these situations, and programmers will regularly want to react differently depending on the type of failure (e.g. a program may want to retry a request when it times out but display an error to the user if a URL doesn’t exist). Errors of this type are the main focus of this chapter.

While most failable standard library APIs return trivial errors (i.e. optionals), the Codable system uses rich errors. Encoding and decoding have many different error conditions, and precise error information is valuable for clients to figure out what went wrong. The methods for encoding and decoding are annotated with throws to tell callers to prepare for handling errors.

Unexpected errors — A condition that the programmer didn’t anticipate occurred and that makes it difficult or impossible to continue. This usually means that an assumption the programmer made (“this can never happen”) turned out to be false. Examples where the standard library follows this pattern include accessing an array with an out-of-bounds index, creating a range with an upper bound that’s smaller than the lower bound, integer overflow, and integer division by zero.

The usual way to deal with an unexpected error in Swift is to let the program crash, because continuing with an unknown program state would be unsafe. Moreover, these situations are considered **programmer errors** that should be caught in testing — it’d be inappropriate to handle them e.g. by displaying an error to the user.

In code, we use assertions (i.e. `assert`, `precondition`, or `fatalError`) to validate our expectations and trap if an assumption doesn't hold. We looked at these functions in the [Optionals chapter](#). Assertions are a great tool for identifying bugs in your code. Used correctly, they show you at the earliest possible moment when your program is in a state you didn't expect. They're also a useful documentation tool: every `assert` or `precondition` call makes the author's (usually implicit) assumptions about program state visible to other readers of the code.

Assertions should never be used to signal expected errors — doing so would make graceful handling of these errors impossible because programs can't recover from assertions. The opposite — using optionals or throwing functions to point out programmer errors — should also be avoided, because it's better to catch a wrong assumption at the source than let it permeate through other layers of the program.

The Result Type

Before we look at Swift's built-in error handling in more detail, let's discuss the `Result` type. `Result` was added to the standard library in Swift 5, but variants of it have been popular in the Swift community since the very first release of Swift. Understanding how `Result` is used to communicate errors will clarify how Swift's error handling works when you take away the syntactic sugar.

Recall from the [Enums chapter](#) that `Result` is an enum with a shape similar to `Optional` — like `Optional`, `Result` has two cases. The cases have different names — `success` and `failure` — but they have the same function `some` and `none` have for `Optional`. The difference to `Optional` is that `Result.failure` also has an associated value, which allows `Result` instances to carry rich error information:

```
enum Result<Success, Failure: Error> {
    case success(Success)
    case failure(Failure)
}
```

Also note that the generic parameter for the failure case's payload is constrained to the `Error` protocol to communicate its intended use as an error value.

The differences between `Optional` and `Result` should remind you of the distinction between trivial and rich errors that we made above. This is no coincidence; we can use `Result` for rich errors in the same manner `Optional` is used to return trivial errors.

Suppose we're writing a function to read a file from disk. As a first try, we could define the interface using an optional. Because reading a file might fail, we want to be able to return nil:

```
func contentsOrNil(ofFile filename: String) -> String?
```

The function signature above is simple, but it doesn't tell us anything about why reading the file failed. Does the file not exist? Or do we not have the right permissions? This is an example where the failure reason matters. Let's define an enum for the possible error cases:

```
enum FileError: Error {
    case fileDoesNotExist
    case noPermission
}
```

Now we can change the type of our function to return a Result, i.e. either a string (success) or a FileError (failure):

```
func contents(ofFile filename: String) -> Result<String, FileError>
```

The caller of the function can switch over the return value and react differently based on the specific error that was returned. In the code below, we try to read the file and print the contents if reading succeeds. In the failure case, we print a tailormade error message for each possible error:

```
let result = contents(ofFile: "input.txt")
switch result {
    case let .success(contents):
        print(contents)
    case let .failure(error):
        switch error {
            case .fileDoesNotExist:
                print("File not found")
            case .noPermission:
                print("No permission")
        }
}
```

Notice that neither of the two nested switch statements requires a default case — the compiler can verify that we're switching exhaustively over all possible values. This

works because `FileError` is an enum. Had we used `Result<String, Error>` as the function's return type, we'd have had to include a default case.

Throwing and Catching

Swift's built-in error handling shares many aspects with the `Result`-based approach in the previous section, despite having a very different syntax. Instead of giving a function a `Result` return type to indicate that it can fail, we now mark it as `throws`. For every throwing function, the compiler will verify that the caller either catches the error or propagates it. Converting the `contents(ofFile:)` function from above to the `throws` syntax looks like this:

```
func contents(ofFile filename: String) throws -> String
```

When calling a throwing function, our code won't compile unless we annotate the call with `try`. The `try` keyword signals that a function can throw an error, both to the compiler and to readers of the code.

Calling a throwing function also forces us to decide how we want to deal with errors. We can either handle an error by using `do/catch`, or we can propagate the error up the call stack by annotating the calling function itself with `throws`. There can be more than one `catch` clause, and `catch` clauses support pattern matching for catching specific error types or values. In the example below, we explicitly catch a `fileDoesNotExist` case and then handle all other errors in a `catchall` clause. Within the `catchall` clause, the compiler automatically makes a variable named `error` available (much like the implicit `newValue` variable in a property's `willSet` handler):

```
do {
    let result = try contents(ofFile: "input.txt")
    print(result)
} catch FileError.fileDoesNotExist {
    print("File not found")
} catch {
    print(error)
    // Handle any other error.
}
```

The error handling syntax probably looks familiar to you. Many other languages use the same try, catch, and throw keywords for exception handling. Despite the resemblance, error handling in Swift doesn't incur the runtime cost that's often associated with exceptions. The compiler treats throw like a regular return, making both code paths very fast.

If we want to expose more information in our errors, we can use an enum with associated values. For example, a file parsing library might model its error conditions like this:

```
enum ParseError: Error {
    case wrongEncoding
    case warning(line: Int, message: String)
}
```

An enum is a popular choice for modeling error values, but note that we could've also used a struct or class; any type that conforms to the Error protocol can be used as an error in a throwing function. And because the Error protocol has no requirements, any type can choose to conform to it without extra implementation work.

For quick tests or prototypes, we sometimes find it useful to conform String to Error, which can be done with this one-liner: extension String: Error {}.

Doing this allows us to treat any error message directly as an error value, as in throw "File not found". It isn't something we'd recommend for production code, not least because conforming a type you don't own to a protocol you don't own isn't advisable (see the [Protocols](#) chapter for more on this). But it's a nice little hack for a REPL session or similar environments.

The type of our parsing function looks like this:

```
func parse(text: String) throws -> [String]
```

Now, if we want to parse a string, we can again use pattern matching to distinguish between the error cases. In the warning case, we can bind the line number and warning message to variables, like we'd do in a switch statement:

```
do {
    let result = try parse(text: "{\"message\": \"We come in peace\" }")
```

```
    print(result)
} catch ParseError.wrongEncoding {
    print("Wrong encoding")
} catch let ParseError.warning(line, message) {
    print("Warning at line \(line): \(message)")
} catch {
    preconditionFailure("Unexpected error: \(error)")
}
```

If you squint at the overall shape of this piece of code, with its separate sections for the success and failure paths and the use of pattern matching to bind values, it looks remarkably similar to switching over a `Result` value. The parallels are no accident — Swift’s error handling is essentially a nicer syntax for creating and unwrapping `Result`-like values.

Typed and Untyped Errors

Something about the `do/catch` code in the previous section doesn’t feel quite right. Even if we’re absolutely sure that the only error that could happen is of type `ParseError` (which we’re handling exhaustively), we still need to write a final `catch` clause to convince the compiler that we caught all possible errors.

This is because Swift’s native error handling uses *untagged errors* — we can only mark a function as throws, but we can’t specify *which* errors it’ll throw. As a result, the compiler always requires a catchall clause to prove all errors are caught exhaustively. Making error handling untagged was a deliberate design decision by the Swift Core Team. The rationale is that exhaustive error handling is impractical and undesirable in most situations; usually, you probably only care about one or two specific errors (if at all) and are fine with handling all other errors in a catchall clause.

The `Result` type, on the other hand, uses *tagged errors*: `Result` has two generic parameters, `Success` and `Failure`, and the latter specifies the concrete type of an error value. Earlier, this feature allowed us to switch exhaustively over a `Result<String, FileError>`; as another example, here’s a variant of the `parse(text:)` function where we replaced the `throws` annotation with a `Result<[String], ParseError>` return type. Thanks to the concrete error type, this function also enables exhaustive switching over its failure cases:

```
func parse(text: String) -> Result<[String], ParseError>
```

Why did the Swift Core Team decide to accept this mismatch between using untyped errors for the built-in error handling and a Result type with typed errors? After all, the Core Team could also have opted for a variant of Result with an untyped failure case, i.e. one where the failure case could be any Error value. Well, it turns out the Result type we have is really a hybrid that supports both patterns. If you don't want a concrete error type, you can specify `Result<..., Error>` to accept any error value.

So Result really gives us the choice between typed and untyped errors. The tradeoff is that a Result with an untyped failure type requires a little more typing than would otherwise be necessary because we have to drag the Error parameter along. If that bothers you, you can always make a type alias for a Result type with untyped errors:

```
typealias UResult<Success> = Result<Success, Error>
```

By the way, the fact that we can write `Result<..., Error>` takes advantage of some special compiler magic for the Error protocol. We saw that the Failure parameter is constrained to Error:

```
enum Result<Success, Failure: Error>
```

Since Swift protocols normally don't conform to themselves, a variable of type `Result<..., Error>` wouldn't satisfy the `Failure: Error` constraint. To allow Result to be used with untyped errors in this manner, the Swift team added a special case to the compiler that allows self-conformance for Error (but for no other protocol). We discuss this in more detail in the [Protocols](#) chapter.

Now that we have a Result type with support for concrete error types, it's quite possible this feature will also be added to the native error handling model in a future Swift version. Until then, using Result with typed errors is a good choice for code where you want the compiler to verify that you caught all possible errors. If and when we get typed errors everywhere, the ability to specify the concrete error type a function can throw will almost certainly become an optional feature and not a required one. This is because typed errors come with significant downsides:

- Concrete error types make composing throwing functions and aggregating errors much more difficult. Any function that called multiple other throwing functions would either have to propagate multiple error types up the call stack or come up with a new custom error type for aggregating the errors from the lower levels. This would quickly get out of hand. We'll come back to this later in the [Chaining Errors](#) section.

- Strictly typed errors make libraries less extensible. For example, every time a function were to add a new error condition, it'd be a source-breaking change for all clients that catch errors exhaustively. To maintain binary compatibility with different library versions, clients would have to add default cases to all do/catch statements anyway, similar to what they must do for non-frozen enums. This issue alone is reason enough for why frameworks like Cocoa probably won't ever have typed errors.
- Unlike exhaustive switching over enums, exhaustive catching of all possible error conditions is usually neither necessary nor practical. Think about how many different things can go wrong when you execute a network request — it's nearly impossible for a programmer to come up with a meaningful reaction to every possible issue. Most programs will probably handle a handful of errors explicitly and have a generic error handler, which may just log the error or present it to the user, for the rest.

Because errors are untyped, it's important to document the types of errors your functions can throw. Xcode supports a Throws keyword in documentation markup for this purpose. Here's an example:

```
/// Opens a text file and returns its contents.  
///  
/// - Parameter filename: The name of the file to read.  
/// - Returns: The file contents, interpreted as UTF-8.  
/// - Throws: 'FileError' if the file does not exist or  
///           the process doesn't have read permissions.  
func contents(ofFile filename: String) throws -> String
```

The Quick Help popover that appears when you Option-click on the function name will now include an extra section for the thrown errors.

Non-Ignorable Errors

In the introduction of this chapter, we identified safety as one factor of a good error handling system. A big benefit of using the built-in error handling is that the compiler will make sure you can't ignore the error case when calling a function that might throw. With **Result**, this isn't always the case.

For example, consider Foundation methods such as `Data.write(to:options:)` (for writing bytes to a file) or `FileManager.removeItem(at:)` (for deleting a file):

```
extension Data {  
    func write(to url: URL, options: Data.WritingOptions = []) throws  
}  
  
extension FileManager {  
    func removeItem(at URL: URL) throws  
}
```

If these methods used Result-based error handling instead, their declarations would look like this:

```
extension Data {  
    func write(to url: URL, options: Data.WritingOptions = []) -> Result<(), Error>  
}  
  
extension FileManager {  
    func removeItem(at URL: URL) -> Result<(), Error>  
}
```

What's special about these methods is that we call them for their side effects and not for their return values — in fact, neither method has a meaningful return value except “the operation did or didn't succeed.” With the Result-based variants, it's all too easy for programmers to (accidentally or intentionally) ignore any failures by writing code such as this:

```
_ = FileManager.default.removeItem(at: url)
```

When calling the throws-based variants, on the other hand, the compiler forces us to prefix the call with `try`. The compiler also requires that we either wrap that call in a `do/catch` block or propagate the error up the call stack. This makes it immediately clear to the programmer and to other readers of the code that the operations can fail, and the compiler will force us to handle the error.

Although `Result<(), Error>` may not be a good choice for a function's return type, it's commonly used for callback-based error reporting (where `throws` isn't available, as we'll see in the [Errors and Callbacks](#) section) when the success

case has no meaningful payload. The empty tuple, or `Void`, is a type that has exactly one possible value, `()` (confusingly, the same spelling is used for the type and its only value). Hence, the success case carries no additional information other than “the operation succeeded.”

Error Conversions

Converting between throws and Optionals

Errors and optionals are both common ways for functions to signal that something went wrong. In the introduction of this chapter, we gave you some advice on how to decide which pattern you should use for your own functions. You’ll end up working a lot with both errors and optionals, and passing results to other APIs will often make it necessary to convert back and forth between throwing functions and optional values.

The `try?` keyword allows us to ignore the error of a throwing function and convert the return value into an optional; the optional tells us if the function succeeded or not:

```
if let result = try? parse(text: input) {  
    print(result)  
}
```

Using `try?` means we receive less information than before: we only know if the function returned a successful value or if it returned some error — specific information about that error gets thrown away. To go the other way, from an optional to a function that throws, we have to provide an error value that should be used in case the optional is `nil`. Here’s an extension on `Optional` that unwraps itself and throws the given error if it finds `nil`:

```
extension Optional {  
    /// Unwraps `self` if it is non-`nil`.  
    /// Throws the given error if `self` is `nil`.  
    func or(error: Error) throws -> Wrapped {  
        switch self {  
            case let x?: return x  
            case nil: throw error  
        }  
    }  
}
```

And here's a usage example:

```
do {
    let int = try Int("42").or(error: ReadIntError.couldNotRead)
} catch {
    print(error)
}
```

This can be useful in conjunction with multiple try statements, or when you're working inside a function that's already marked as throws. It can also be a helpful pattern in unit tests — so much so that the XCTest framework provides a convenience API for it. XCTest can automatically fail a test if you throw an error inside the test method (which you must mark as throws to make it compile). If your test relies on an optional being non-nil to continue, you can call let nonOptional = try XCTUnwrap(someOptional) to unwrap the optional or fail the test if the optional is nil in a single line.

The existence of the try? keyword may appear contradictory to Swift's philosophy that ignoring errors shouldn't be allowed. However, you still have to explicitly write try? so that the compiler forces you to acknowledge your actions and make them explicit to other readers of the code. try? is a legitimate option when you're not interested in the error message.

There's a third variant of try: try!. This is used when you know there can't possibly be an error result. Just like force-unwrapping an optional value that's nil, try! causes a crash if your assumptions turn out to be wrong at runtime.

Converting between throws and Result

We saw that Result and error handling using throws are effectively two sides of the same coin. Differences in the handling of typed and untyped errors aside, you can think of a Result value as the reified outcome (i.e. a value you can store or pass around) of a throwing function. Given this duality, it's no surprise that the standard library provides ways to convert between the two representations.

To call a throwing function and wrap its outcome in a Result, use the init(catching:) initializer, which takes a throwing function and converts it into a Result. The implementation looks like this:

```
extension Result where Failure == Swift.Error {  
    /// Creates a new result by evaluating a throwing closure, capturing the  
    /// returned value as a success, or any thrown error as a failure.  
    init(caught body: () throws -> Success) {  
        do {  
            self = .success(try body())  
        } catch {  
            self = .failure(error)  
        }  
    }  
}
```

And here's an example:

```
let encoder = JSONEncoder()  
let encodingResult = Result { try encoder.encode([1, 2]) } // success(5 bytes)  
type(of: encodingResult) // Result<Data, Error>
```

This is useful if you want to delay handling the error to a later time or pass the result to another function.

The reverse operation is called `Result.get()`. It evaluates (i.e. switches over) the `Result` and treats the failure case as an error to be thrown. This is [the implementation](#):

```
extension Result {  
    public func get() throws -> Success {  
        switch self {  
            case let .success(success):  
                return success  
            case let .failure(failure):  
                throw failure  
        }  
    }  
}
```

Chaining Errors

It's common to call multiple failable functions in a row. For example, an operation may be divided into multiple subtasks, where the result of one subtask becomes the input for

the next. Every single subtask can fail with an error, so if an error is thrown, the entire operation should abort immediately.

Chaining throws

Not all error handling systems handle the above use case well, but Swift's built-in error handling shines in this regard. There's no need for nested if statements or similar constructs to unwrap return values before passing them to the next function; we simply place all function calls into a single do/catch block (or wrap them in a throwing function). The first error that occurs breaks the chain and switches control to the catch block (or propagates the error to the caller).

Here's an example of an operation with three subtasks:

```
func complexOperation(filename: String) throws -> [String] {
    let text = try contents(ofFile: filename)
    let segments = try parse(text: text)
    return try process(segments: segments)
}
```

Chaining Result

Let's compare the clean try-based example to the equivalent code when using the `Result` type. Chaining multiple functions that return a `Result` is a lot of work if we do it manually; we first call the first function and switch over its return value, and if it's a `.success`, we can pass the unwrapped value to the second function and start over. As soon as one function returns a `.failure`, the chain breaks and we short-circuit by immediately returning the failure to the caller:

```
func complexOperation1(filename: String) -> Result<[String], Error> {
    let result1 = contents(ofFile: filename)
    switch result1 {
        case .success(let text):
            let result2 = parse(text: text)
            switch result2 {
                case .success(let segments):
                    return process(segments: segments)
                .mapError { $0 as Error }
            }
    }
}
```

```
    case .failure(let error):
        return .failure(error as Error)
    }
    case .failure(let error):
        return .failure(error as Error)
    }
}
```

This gets ugly quickly because each additional function in the chain requires another nested switch statement. Note as well that we had to duplicate the same failure path in each switch.

Before we attempt to refactor this code, let's take another look how we're handling the failure cases in the above code. These are the signatures of the functions that represent the three subtasks in our example:

```
func contents(ofFile filename: String) -> Result<String, FileNotFoundError>
func parse(text: String) -> Result<[String], ParseError>
func process(segments: [String]) -> Result<[String], ProcessError>
```

Each of these functions has a different failure type: `FileError`, `ParseError`, and `ProcessError`. As we work our way through the chain of subtasks, we not only have to care about transforming the success types (from `String` to `[String]` and again to `[String]`); we must also take care of transforming the failure types into an aggregate type (which is `Error` in this example, but could be another concrete type). We can see the error transformations in three places in the code:

- The two `return .failure(error as Error)` lines convert the `error` value from its concrete type to `Error`. We could've omitted the `as Error` part here — the compiler would've added it implicitly, but adding it illustrates what's really going on.
- For the final step of the chain, we can't just write `return process(segments: segments)` because the return type of `process(segments)` isn't compatible with the required return type, `Result<[String], Error>` — we have to use the `mapError` method (which is part of the `Result` type) to convert the failure type once more.

Regardless of the complexity introduced by the strict error types, we should refactor the mess of nested `switch` statements. Fortunately, `Result` includes functionality to do just that. The pattern we used multiple times in the above code — switching over a `Result` and either invoking the next step in the chain with the unwrapped success value, or

aborting when we encounter a failure — is precisely what `Result`'s `flatMap` method does. Its structure is identical to the `flatMap` method for optionals that we covered in the [Optionals chapter](#).

Replacing the switches with `flatMap` cleans the code up significantly. In fact, the end result is quite elegant, if not nearly as clean as the `throws`-based example:

```
func complexOperation2(filename: String) -> Result<[String], Error> {
    return contents(ofFile: filename).mapError { $0 as Error }
        .flatMap { text in
            parse(text: text).mapError { $0 as Error }
        }
        .flatMap { segments in
            process(segments: segments).mapError { $0 as Error }
        }
}
```

Note that we still have to deal with the incompatible failure types. `Result`'s `map` and `flatMap` methods only transform the success case and leave the type of the failure case unchanged. And chaining multiple `map` or `flatMap` operations only works if all involved `Result` types have the same Failure type. We achieve this in our example with several `mapError` calls; their task is to convert the specific errors to `Error`.

The example we discussed in this section nicely illustrates that strictly typed errors can be problematic and may often cause more trouble than they're worth. The `flatMap`-based code would certainly be more readable without the `mapError` calls. What's more is the aggregation function ended up erasing the concrete error types anyway, so the actual error handling code higher up the call stack can't even take advantage of the types.

Errors and Callbacks

Swift's built-in error handling integrates seamlessly with asynchronous APIs that use the new `async/await` model: everything we've discussed so far equally applies to `async` functions or methods. However, the built-in error handling does *not* work well for callback-based asynchronous APIs. Let's look at a function that asynchronously computes a large number and calls back our code when the computation has finished:

```
func compute(callback: (Int) -> ())
```

We can call the function by providing a callback function. The callback receives the result as its only parameter:

```
compute { number in
    print(number)
}
```

How do we integrate errors into this design? If an optional provides sufficient information for the error (i.e. there's only one simple error condition), we could specify that the callback receives an optional integer, which would be `nil` in case of a failure:

```
func computeOptional(callback: (Int?) -> ())
```

Now, in our callback, we must unwrap the optional, e.g. by using the `??` operator:

```
computeOptional { numberOrNil in
    print(numberOrNil ?? -1)
}
```

What if we want to report more specific errors to the callback though? This function signature seems like a natural solution:

```
func computeThrows(callback: (Int) throws -> ())
```

But this doesn't do what we want; this type has a totally different meaning. Instead of saying that the computation might fail, it expresses that the callback itself could throw an error. The problem becomes clearer when we try to rewrite this wrong attempt into a version that uses `Result`:

```
func computeResult(callback: (Int) -> Result<(), Error>)
```

This isn't correct either — we need to wrap the `Int` argument in a `Result` and not in the callback's return type.

Finally, this is the correct solution:

```
func computeResult(callback: (Result<Int, Error>) -> ())
```

The incompatibility of the native error handling with callback-based APIs illustrates a key difference between `throws` and `Optional / Result`: only the latter are values we can pass around freely, whereas `throws` isn't as flexible. We like how [Joshua Emmons said it](#):

See, throw, like return, only works in one direction; up. We can throw an error “up” to the caller, but we can’t throw an error “down” as a parameter to another function we call.

The ability to pass an error “down” to a continuation function is exactly what we need in callback-based asynchronous contexts. Unfortunately, there’s no clear way to write the variant above with throws. The best we can do is wrap the Int inside another throwing function. This makes the signature more complicated:

```
func compute(callback: () throws -> Int) -> ()
```

And using this variant becomes more difficult for the caller too. To get the integer out, the callback now has to call the throwing function. This is where the caller must perform the error checking:

```
compute { (resultFunc: () throws -> Int) in
    do {
        let result = try resultFunc()
        print(result)
    } catch {
        print("An error occurred: \(error)")
    }
}
```

This works, but it’s definitely not idiomatic Swift; Result is the way to go for error handling with callbacks. And starting with Swift 5.5, the compute function arguably should be written as an async function instead, which greatly simplifies error handling:

```
func compute() async throws -> Int

do {
    print(try await compute())
} catch {
    print("An error occurred: \(error)")
}
```

Cleaning Up Using defer

Many programming languages have a try/finally construct, where the block marked with finally is always executed when the function returns, regardless of whether or not an error was thrown. The defer keyword in Swift has a similar purpose but works a bit differently. Like finally, a defer block is always executed when a scope is exited, regardless of the reason of exiting — whether it's because a value is successfully returned, because an error happened, or any other reason. This makes defer a good option for performing required cleanup work. Unlike finally, a defer block doesn't require a leading try or do block, and it's more flexible in terms of where you place it in your code.

Let's go back to the contents(ofFile:) function from the beginning of this chapter and look at a possible implementation that uses defer:

```
func contents(ofFile filename: String) throws -> String {  
    let file = open(filename, O_RDONLY)  
    defer { close(file) }  
    return try load(file: file)  
}
```

The defer block in the second line ensures that the file is closed when the function returns, regardless of whether it completes successfully or throws an error.

While defer is often used together with error handling, it can be useful in other contexts too — for example, when you want to keep the code for initialization and cleanup of a resource (such as opening and closing a file) close together. Putting related lines next to each other can make your code significantly more readable, especially in longer functions.

If there are multiple defer statements in the same scope, they're executed in reverse order; you can think of them as a stack. At first, it might feel strange that the defer blocks run in reverse order. However, it should quickly make sense if we look at this example of executing a database query:

```
let database = try openDatabase(...)  
defer { closeDatabase(database) }  
let connection = try openConnection(database)  
defer { closeConnection(connection) }  
let result = try runQuery(connection, ...)
```

This code has to open a database and a connection to the database first before it can finally run the query. If an error is thrown — for example, during the `runQuery` call — cleaning up the resources needs to happen the other way around; we want to close the connection first and the database second. Because the `defer` statements run in reverse, this happens automatically.

A `defer` block is executed just before program control is transferred outside the scope the `defer` statement appears in. Even the value of a `return` statement is evaluated before any `defer` blocks in the same scope get to run. You can take advantage of this behavior to mutate a variable after returning its previous value to the caller. In the following example, the `increment` function uses `defer` to increment the value of the captured `counter` variable *after* returning its previous value:

```
var counter = 0

func increment() -> Int {
    defer { counter += 1 }
    return counter
}

increment() // 0
counter // 1
```

If you browse through the standard library source code, you'll see this pattern from time to time. Writing the same logic without `defer` would have required declaring a local variable to store the value of `counter` temporarily.

There are some situations in which `defer` statements don't get executed: when your program segfaults, or when it raises a fatal error (e.g. using `fatalError` or by force-unwrapping a `nil`), all execution halts immediately.

Rethrowing

The existence of throwing functions poses a problem for functions that take other functions as arguments, such as `map` or `filter`. In the [Built-In Collections](#) chapter, we discussed the type of a hypothetical `filter` method on `Array` (the real `filter` is defined on `Sequence` and is slightly more complicated):

```
func filter(_ isIncluded: (Element) -> Bool) -> [Element]
```

This definition works, but it has one drawback: the compiler won't accept any throwing function to be passed in as the predicate because the `isIncluded` parameter isn't marked as `throws`.

Let's walk through an example where this limitation becomes an issue. We start by writing a function that checks a file for some notion of validity (which factors the function uses to decide if a file is valid or not isn't important for the example). The `checkFile` function either returns a Boolean (true for valid, false for invalid), or it throws an error if something goes wrong when checking the file:

```
func checkFile(filename: String) throws -> Bool
```

Suppose we have an array of file names and want to filter out the invalid files. Naturally, we'd like to use `filter` to do this, but the compiler won't allow it because `checkFile` is a throwing function:

```
let filenames: [String] = ...
// Error: Call can throw but is not marked with 'try'
let validFiles = filenames.filter(checkFile)
```

We could work around the issue by handling the error locally inside the `filter` predicate:

```
let validFiles = filenames.filter { filename in
    do {
        return try checkFile(filename: filename)
    } catch {
        return false
    }
}
```

But this is inconvenient, and it may not even be what we want — the above code silently ignores errors by catching them and returning `false`, but what if we want to abort the entire operation when an error occurs?

One solution would be for the standard library to annotate the predicate function with `throws` in the declaration of `filter`:

```
func filter(_ isIncluded: (Element) throws -> Bool) throws -> [Element]
```

This would work, but it'd be equally inconvenient because now *every* invocation of `filter` becomes a throwing call that requires a `try` (or `try!`) annotation. Doing this for every

higher-order function in the standard library would lead to code that's covered in try keywords, thereby defeating the main purpose of try, which is to allow readers of the code to quickly distinguish between throwing and non-throwing calls.

Another alternative is to define two versions of filter: one that throws, and one that doesn't. With the exception of the try annotation, their implementations would be identical. We could then rely on the compiler to pick the best overload for each invocation. This is better because it keeps call sites clean, but it's still quite wasteful.

Luckily, Swift has a better solution in the form of the rethrows keyword. Annotating a function with rethrows tells the compiler that this function will only throw an error when its function parameter throws an error. Thus, the true method signature for filter looks like this:

```
func filter(_ isIncluded: (Element) throws -> Bool) rethrows -> [Element]
```

The predicate function is still marked as throws, indicating that callers may pass in a throwing function. In its implementation, filter must call the predicate with try. The rethrows annotation guarantees that filter will propagate errors thrown by the predicate function up the call stack, but filter will never throw an error on its own. This allows the compiler to waive the requirement that filter be called with try when the caller passes in a non-throwing predicate function.

Almost all sequence and collection functions in the standard library that take a function argument are annotated with rethrows, with one important exception: lazy collection methods, which we discussed in the [Collection Protocols](#) chapter, generally don't have support for throwing because the lazy collections store the transformation function for later evaluation. Supporting throwing in this context would require a try annotation on any call of a Collection API that might go to a lazy collection, once more defeating the purpose of try as a marker for actual throwing calls.

Bridging Errors to Objective-C

Objective-C has no mechanism that's similar to throws and try. (Objective-C *does* have exception handling that uses these same keywords, but exceptions in Objective-C should only be used to signal programmer errors. You'll rarely catch an Objective-C exception in a normal app.)

Instead, the common pattern in Cocoa is that a method returns `NO` or `nil` when an error occurs. Failable methods also take a reference to an `NSError` pointer as an extra argument; they can use this pointer to pass concrete error information back to the caller. For example, the `contentsOfFile:` method would look like this in Objective-C:

```
- (NSString *)contentsOfFile:(NSString *)filename error:(NSError **)error;
```

Swift automatically translates methods that follow this pattern to the `throws` syntax. The `error` parameter gets removed since it's no longer needed, and `BOOL` return types are changed to `Void`. The method above gets imported like this:

```
func contents(ofFile filename: String) throws -> String
```

The automatic conversion works for all Objective-C methods that use this structure. Other `NSError` parameters — for example, in asynchronous APIs that pass an error back to the caller in a completion block — are bridged to the `Error` protocol, so you don't generally need to interact with `NSError` directly.

If you pass a Swift error to an Objective-C method, it'll be bridged back to `NSError`. Since all `NSError` objects must have a domain string and an integer error code, the runtime will generate default values if necessary, using the qualified type name as the domain and numbering enum cases from zero for the error code. Optionally, you can provide your own values by conforming your type to the `CustomNSError` protocol.

For example, we could extend our `ParseError` like this:

```
extension ParseError: CustomNSError {
    static let errorDomain = "io.objc.parseError"

    var errorCode: Int {
        switch self {
            case .wrongEncoding: return 100
            case .warning(_, _): return 200
        }
    }

    var userInfo: [String: Any] {
        return [:]
    }
}
```

In a similar manner, you can add conformance to one or both of the following protocols to provide better interoperability with Cocoa conventions:

- **LocalizedError** — This provides localized messages describing the error (`errorDescription`), why the error occurred (`failureReason`), tips for how to recover (`recoverySuggestion`), and additional help text (`helpAnchor`).
- **RecoverableError** — This describes an error the user can recover from by presenting one or more `recoveryOptions` and performing the recovery when the user requests it. This is mostly used in macOS apps using AppKit.

Even without conforming to `LocalizedError`, every type that conforms to `Error` has a `localizedDescription` property, which you can override in your own types. However, because `localizedDescription` isn't a requirement of the `Error` protocol, the property isn't dynamically dispatched. Unless you also conform to `LocalizedError`, your custom `localizedDescription` won't be used by Objective-C APIs or by values wrapped in an `Error` existential. When writing Cocoa apps, you should always implement the `LocalizedError` protocol for error types that get passed to Cocoa APIs. For more information on dynamic dispatch and existentials, refer to the [Protocols chapter](#).

Recap

Swift gives us many choices for handling the unexpected in our code. When we can't possibly continue, we can use `fatalError` or an assertion. When we're not interested in the kind of error, or if there's only one kind of error, we can use optionals. When we need more than one kind of error or want to provide additional information, we can use Swift's native error handling model or the `Result` type.

When Apple introduced the error handling model in Swift 2.0, a lot of people in the community were skeptical. The fact that `throws` uses untyped errors was seen as a needless deviation from the strict typing in other parts of the language. We were skeptical too, but in hindsight, we think the Swift team was proven correct because fine-grained error handling is often unnecessary. Now that we have a `Result` type with a generic failure type, there's a good chance that strongly typed error handling will be added as an opt-in feature in the future.

Error handling is a good example of Swift being a pragmatic language that optimizes for the most common use case first. Keeping the syntax familiar for developers who are used to C-style languages is a more important goal than adhering to the "purer" functional style based on `Result` and `flatMap` — although those are now also available in

the standard library. The design of the error handling model follows a common theme for Swift: the goal is to wrap safe, “functional” concepts in a friendly, imperative syntax (another example is the mutability model for value types). The introduction of the `async/await`-style concurrency model, including support for `throws`-style error handling in asynchronous functions, is an additional step in that direction.

Encoding and Decoding

14

Serializing a program's internal data structures into some kind of data interchange format and vice versa is one of the most common programming tasks. Swift calls these operations *encoding* and *decoding*.

The Codable system (named after its base “protocol,” which is really a type alias) is a standardized design for encoding and decoding data that all custom types can opt into. It's designed around three central goals:

- **Universality** — It should work with structs, enums, and classes.
- **Type safety** — Interchange formats such as JSON are often weakly typed, whereas your code should work with strongly typed data structures.
- **Reducing boilerplate** — Developers should have to write as little repetitive “adapter code” as possible to let custom types participate in the system. The compiler should generate this code automatically.

Types declare their ability to be (de)serialized by conforming to the Encodable and/or Decodable protocols. Each of these protocols has just one requirement — Encodable defines an `encode(to:)` method in which a value encodes itself, and Decodable specifies an initializer for creating an instance from serialized data:

```
/// A type that can encode itself to an external representation.  
public protocol Encodable {  
    /// Encodes this value into the given encoder.  
    public func encode(to encoder: Encoder) throws  
}  
  
/// A type that can decode itself from an external representation.  
public protocol Decodable {  
    /// Creates a new instance by decoding from the given decoder.  
    public init(from decoder: Decoder) throws  
}
```

Because most types that adopt one will also adopt the other, the standard library provides the Codable type alias as shorthand for both:

```
public typealias Codable = Decodable & Encodable
```

All basic standard library types — including Bool, the number types, and String — are Codable out of the box, as are optionals, arrays, dictionaries, sets, and ranges containing Codable elements. Furthermore, many common data types used by Apple's frameworks

— including Data, Date, URL, CGPoint, and CGRect — have adopted Codable. Lastly, the Swift compiler can synthesize the conformance to Codable for structs, classes, and enums if the types of their properties or associated values conform to Codable.

The flipside of relying on both the built-in coding capabilities of all these types and the compiler-generated Codable conformance is a lack of control over the serialized data format. The Codable system works best if you're looking for an easy way to serialize (and deserialize) your data but you don't have special requirements for how exactly the data has to be represented in its serialized format. If you want to interface with external data formats, e.g. a JSON API you don't control, Codable can still be a good fit if the format only deviates slightly from Swift's defaults. If the data format has too many incompatibilities with the defaults of the Codable system, it's still possible to build your serialization code on top of the Codable architecture, but you'll have to write a lot of custom encoding and decoding code to make this work.

Below, we'll first look at how the Codable system can be used for out-of-the-box serialization. Then, we'll explore how property wrappers can be used to selectively customize the serialized data format. Finally, we'll look into the encoding and decoding process in depth.

A Minimal Example

Let's begin with a minimal example of how you'd use the Codable system to encode an instance of a custom type to JSON.

Automatic Conformance

Making one of your own types codable can be as easy as conforming it to Codable. If all of the type's stored properties are themselves codable, the Swift compiler will automatically generate code that implements the Encodable and Decodable protocols. This Coordinate struct stores a GPS location:

```
struct Coordinate: Codable {  
    var latitude: Double  
    var longitude: Double  
    // Nothing to implement here.  
}
```

Because both stored properties are already codable, adopting the Codable protocol is enough to satisfy the compiler. Now we can write a Placemark struct that takes advantage of Coordinate’s Codable conformance:

```
struct Placemark: Codable {  
    var name: String  
    var coordinate: Coordinate  
}
```

Enums equally benefit from the Codable code synthesis if they don’t have associated values, or if their associated values conform to Codable. For example, we could define the following Surrounding enum with associated values, add it to an alternative version of Placemark, and get all the codable conformances for free:

```
enum Surrounding: Codable {  
    case land  
    case inlandWater(name: String)  
    case ocean(name: String)  
}  
  
struct Placemark2: Codable {  
    var name: String  
    var coordinate: Coordinate  
    var surrounding: Surrounding  
}
```

The code the compiler synthesizes isn’t visible, but we’ll dissect it piece by piece a little later in this chapter. For now, treat the generated code like you would a default implementation for a protocol in the standard library, such as `Sequence.drop(while:)` — you get the default behavior for free, but you have the option of providing your own implementation.

The only material difference between code generation and “normal” default implementations is that the latter are part of the standard library, whereas the logic for the Codable code synthesis lives in the compiler. Moving the code into the standard library would require more capable reflection APIs than Swift currently has, and even if they existed, runtime reflection has its own tradeoffs (for one, reflection is typically slower).

Nonetheless, shifting as much of the language definition as possible out of the compiler and into libraries remains a stated goal for Swift. Someday, we’ll probably get a macro

system that's powerful enough to move the entire Codable system into the standard library, but that's at least several years off. Until then, compiler code synthesis is a pragmatic solution to this problem, and one that has applications other than Codable — the same design is used for automatic Equatable and Hashable conformance for structs and enums, as well as Caselterable conformance for enums.

Encoding

Swift ships with two built-in encoders, JSONEncoder and PropertyListEncoder (these are defined in Foundation and not in the standard library). In addition, Codable types are compatible with Cocoa's NSKeyedArchiver. We'll focus on JSONEncoder because JSON is the most common data exchange format on the web.

Here's how we can encode an array of Placemark values to JSON:

```
let places = [
    Placemark(name: "Berlin", coordinate:
        Coordinate(latitude: 52, longitude: 13)),
    Placemark(name: "Cape Town", coordinate:
        Coordinate(latitude: -34, longitude: 18))
]

do {
    let encoder = JSONEncoder()
    let jsonData = try encoder.encode(places) // 129 bytes
    let jsonString = String(decoding: jsonData, as: UTF8.self)
/*
[{"name":"Berlin","coordinate":{"longitude":13,"latitude":52}},
 {"name":"Cape Town","coordinate":{"longitude":18,"latitude":-34}}]
*/
} catch {
    print(error.localizedDescription)
}
```

The actual encoding step is exceedingly simple: create and (optionally) configure the encoder, and pass it the value to encode. The JSON encoder returns a collection of bytes in the form of a Data instance, which we then convert into a string for display.

In addition to a property for configuring the output formatting (pretty-printed and/or keys sorted lexicographically), JSONEncoder provides customization options for

formatting dates (including ISO 8601 or Unix epoch timestamps) and Data values (e.g. Base64), in addition to options for how exceptional floating-point values (infinity and `Nan`) should be treated. We can even use the encoder's `keyEncodingStrategy` option to specify that keys should be converted to snake case, or we can pass in our own custom key conversion function. These options always apply to the entire hierarchy of values being encoded, i.e. you can't use them to specify that a `Date` in one type should follow a different encoding scheme than one in another type. If you need that kind of granularity, you have to write custom `Codable` implementations for the affected types or customize the encoding of specific properties using a property wrapper (we'll show an example of this [below](#)).

It's worth noting that all of these configuration options are specific to `JSONEncoder`. Other encoders will have different options (or none at all). Even the `encode(_)` method is encoder-specific and not defined in any of the protocols. Other encoders might decide to return a `String` or even a `URL` to the encoded file instead of a `Data` value.

In fact, `JSONEncoder` doesn't even conform to the `Encoder` protocol. Instead, it's a wrapper around a private class that implements the protocol and does the actual encoding work. It was designed in this way because the top-level encoder should provide an entirely different API (namely, a single method to start the encoding process) than the `Encoder` object that's being passed to codable types during the encoding process does. Separating these tasks cleanly means clients can only access the APIs that are appropriate in any given situation — for example, a codable type can't reconfigure the encoder in the middle of the encoding process because the public configuration API is only exposed by the top-level encoder. Apple later formalized the concept of top-level encoders and decoders in the `Combine` framework, which includes `TopLevelEncoder` and `TopLevelDecoder` protocols. Moving these into the standard library has been suggested, but it hasn't happened yet.

Decoding

The decoding counterpart to `JSONEncoder` is `JSONDecoder`. Decoding follows the same pattern as encoding: create a decoder and pass it something to decode. `JSONDecoder` expects a `Data` instance containing UTF-8-encoded JSON text, but as we just saw with encoders, other decoders may have different interfaces:

```
do {
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark].self, from: jsonData)
    // [Berlin (lat: 52.0, lon: 13.0), Cape Town (lat: -34.0, lon: 18.0)]
```

```
    type(of: decoded) // Array<Placemark>
    decoded == places // true
} catch {
    print(error.localizedDescription)
}
```

Notice that `decoder.decode(_:from:)` takes two arguments. In addition to the input data, we also have to specify the type we expect to get back (here it's `[Placemark].self`). This allows for full compile-time type safety. The tedious conversion from weakly typed JSON data to the concrete data types we use in our code happens behind the scenes.

Making the decoded type an explicit argument of the decoding method is a deliberate design choice. This wasn't strictly necessary, as the compiler would've been able to infer the correct type automatically in many situations, but the Swift team decided that the increase in clarity and avoidance of ambiguity was more important than maximum conciseness.

Even more so than during encoding, error handling is extremely important during decoding. There are so many things that can go wrong — from missing data (a required field is missing in the JSON input), to type mismatches (the server unexpectedly encodes numbers as strings), to fully corrupted data. Check out the [documentation for the DecodingError type](#) to see what other errors you can expect.

Customizing the Encoded Format

Sometimes the data format used by the built-in encoders and decoders just needs a little bit of tweaking to match your requirements, e.g. to communicate with a particular JSON API. We already saw above that `JSONEncoder` offers multiple configuration options for common variations of how data is represented in the JSON format. If these options don't provide what you need, you should look into using a property wrapper to customize the serialized format for a particular property.

For example, we might want to represent the `Double` values in the `Coordinate` type as strings. For this, we'll implement a `CodedAsString` property wrapper and conform it to `Codable` "manually," i.e. we'll implement the `init(from:)` initializer and the `encode(to:)` method ourselves:

```
@propertyWrapper
struct CodedAsString: Codable {
```

```

var wrappedValue: Double
init(wrappedValue: Double) {
    self.wrappedValue = wrappedValue
}

init(from decoder: Decoder) throws {
    let container = try decoder.singleValueContainer()
    let str = try container.decode(String.self)
    guard let value = Double(str) else {
        let error = EncodingError.Context(
            codingPath: container.codingPath,
            debugDescription: "Invalid string representation of double value"
        )
        throw EncodingError.invalidValue(str, error)
    }
    wrappedValue = value
}

func encode(to encoder: Encoder) throws {
    var container = encoder.singleValueContainer()
    try container.encode(String(wrappedValue))
}

```

Although this seems like a lot of code at first, most of it's just boilerplate or error handling. Only two lines (the call to decode in init, and the call to encode in encode(to:)) are specific to the data transformation from double to string (or vice versa). In the next section, we'll look at the encoding and decoding process in detail and explain how to work with coding containers.

Applying this property wrapper to the latitude and longitude values in our Coordinate struct is straightforward:

```

struct Coordinate: Codable {
    @CodedAsString var latitude: Double
    @CodedAsString var longitude: Double
}

let jsonData = try encoder.encode(places)
let jsonString = String(decoding: jsonData, as: UTF8.self)
/*

```

```
[{"name": "Berlin", "coordinate": {"longitude": "13.0", "latitude": "52.0"}}, {"name": "Cape Town", "coordinate": {"longitude": "18.0", "latitude": "-34.0"}}]  
*/
```

Using property wrappers to customize the encoding and decoding process of certain properties has two big advantages: first, the default coding behavior still applies to all other properties (there are none in our Coordinate example, but in real life, you often just want to customize one of several properties). And second, we can easily reuse a transformation like `CodedAsString` in other places.

If you need more customization than property wrappers on individual properties can provide, you have to manually implement `init(from:)` and `encode(to:)`, which we'll describe in detail in the next section.

The Encoding Process

If *using* the Codable system is all you're interested in and the default behavior works for you, you can stop reading now. But to understand how to customize the way types get encoded, we need to dig a little deeper. How does the encoding process work? What code does the compiler actually synthesize when we conform a type to Codable?

When you initiate the encoding process, the encoder calls the `encode(to: Encoder)` method of the value that's being encoded, passing itself as the argument. It's then the value's responsibility to encode itself into the encoder in any format it sees fit.

In our example above, we pass an array of `Placemark` values to the JSON encoder:

```
let jsonData = try encoder.encode(places)
```

The encoder (or rather its private Encoder-conforming class) will now call `places.encode(to: self)`. How does the array know how to encode itself in a format the encoder understands?

Containers

Let's look at the Encoder protocol to see the interface an encoder presents to the value being encoded:

```

/// A type that can encode values into a native format for external representation.
public protocol Encoder {
    /// The path of coding keys taken to get to this point in encoding.
    var codingPath: [CodingKey] { get }
    /// Any contextual information set by the user for encoding.
    var userInfo: [CodingUserInfoKey: Any] { get }
    /// Returns an encoding container appropriate for holding
    /// multiple values keyed by the given key type.
    func container<Key: CodingKey>(keyedBy type: Key.Type)
        -> KeyedEncodingContainer<Key>
    /// Returns an encoding container appropriate for holding
    /// multiple unkeyed values.
    func unkeyedContainer() -> UnkeyedEncodingContainer
    /// Returns an encoding container appropriate for holding
    /// a single primitive value.
    func singleValueContainer() -> SingleValueEncodingContainer
}

```

Ignoring `codingPath` and `userInfo` for a moment, it's apparent that an `Encoder` is essentially a provider of *encoding containers*. A container is a sandboxed view into the encoder's storage. By creating a new container for each value that's being encoded, the encoder can make sure the values don't overwrite each other's data.

There are three types of containers:

- **Keyed containers** — These encode key-value pairs. Think of a keyed container as a special kind of dictionary. Keyed containers are by far the most prevalent container.

The keys in a keyed encoding container are strongly typed, which provides type safety and autocompletion. The encoder will eventually convert the keys to strings (or integers) when it writes its target format (such as JSON), but that's hidden from client code. Changing the keys your type provides is the easiest way to customize how it encodes itself. We'll see an example of this below.

- **Unkeyed containers** — These encode multiple values sequentially, omitting keys. Think of an array of encoded values. Because there are no keys to identify value, decoding containers must take care to decode values in the same order in which they were encoded.

→ **Single-value containers** — These encode a single value. You'd use these for types that are wholly defined by a single property. Examples include primitive types like `Int` and enums that are `RawRepresentable` as primitive values.

For each of the three container types, there's a protocol that defines the interface through which the container receives values to encode. Here's the definition of `SingleValueEncodingContainer`:

```
/// A container that can support the storage and direct encoding of a single
/// non-keyed value.
public protocol SingleValueEncodingContainer {
    /// The path of coding keys taken to get to this point in encoding.
    var codingPath: [CodingKey] { get }

    /// Encodes a null value.
    mutating func encodeNil() throws

    /// Base types.
    mutating func encode(_ value: Bool) throws
    mutating func encode(_ value: Int) throws
    mutating func encode(_ value: Int8) throws
    mutating func encode(_ value: Int16) throws
    mutating func encode(_ value: Int32) throws
    mutating func encode(_ value: Int64) throws
    mutating func encode(_ value: UInt) throws
    mutating func encode(_ value: UInt8) throws
    mutating func encode(_ value: UInt16) throws
    mutating func encode(_ value: UInt32) throws
    mutating func encode(_ value: UInt64) throws
    mutating func encode(_ value: Float) throws
    mutating func encode(_ value: Double) throws
    mutating func encode(_ value: String) throws

    mutating func encode<T: Encodable>(_ value: T) throws
}
```

As you can see, the protocol mainly declares a bunch of `encode(_)` overloads for various types: `Bool`, `String`, and the integer and floating-point types. There's also a variant for encoding a null value. Every encoder and decoder must support these primitive types, and all `Encodable` types must ultimately be reducible to one of these types. [The Swift Evolution proposal](#) that introduced the `Codable` system says:

These ... overloads give strong, static type guarantees about what is encodable (preventing accidental attempts to encode an invalid type), and provide a list of primitive types that are common to all encoders and decoders that users can rely on.

Any value that isn't one of the base types ends up in the generic `encode<T: Encodable>` overload. In it, the container eventually calls the argument's `encode(to: Encoder)` method, and the entire process will start over one level down until only primitive types are left. But the container is free to treat types with special requirements differently. For instance, it's at this point that `JSONEncoder` checks if it's encoding a `Data` value that must observe the configured encoding strategy, such as `Base64` (the default behavior for `Data` is to encode itself into an unkeyed container of `UInt8` bytes).

`UnkeyedEncodingContainer` and `KeyedEncodingContainerProtocol` have the same structure as `SingleValueEncodingContainer`, but they expose a few additional capabilities, such as being able to create nested containers. If you want to write an encoder and decoder for another data format, most of the work consists of implementing these containers.

How a Value Encodes Itself

Going back to our example, the top-level type we're encoding is `Array<Placemark>`. An unkeyed container is a perfect match for an array (which is, after all, a sequential list of values), so the array asks the encoder for one. The array then iterates over its elements and encodes each element into the container. Here's how this process looks in code:

```
extension Array: Encodable where Element: Encodable {
    public func encode(to encoder: Encoder) throws {
        var container = encoder.unkeyedContainer()
        for element in self {
            try container.encode(element)
        }
    }
}
```

The array elements are `Placemark` instances. We've seen that, for non-primitive types, the container will continue calling each value's `encode(to:)` method.

The Synthesized Code

This brings us to the code the compiler synthesizes for the Placemark struct when we add Codable conformance. Let's walk through it step by step.

Coding Keys

The first thing the compiler generates is a private nested enum named CodingKeys:

```
struct Placemark {  
    // ...  
  
    private enum CodingKeys: CodingKey {  
        case name  
        case coordinate  
    }  
}
```

The enum contains one case for every stored property of the struct. The cases are the keys for a keyed encoding container. Compared to string keys, these strongly typed keys are much safer and more convenient to use because the compiler will find typos. However, encoders must eventually be able to convert keys to strings or integers for storage. Handling these translations is the task of the CodingKey protocol:

```
/// A type that can be used as a key for encoding and decoding.  
public protocol CodingKey {  
    /// The string to use in a named collection (e.g. a string-keyed dictionary).  
    var stringValue: String { get }  
    /// The value to use in an integer-indexed collection  
    /// (e.g. an int-keyed dictionary).  
    var intValue: Int? { get }  
    init?(stringValue: String)  
    init?(intValue: Int)  
}
```

All keys must provide a string representation. Optionally, a key type can also provide a conversion to and from integers. Encoders can choose to use integer keys if that's more efficient, but they're also free to ignore them and stick with string keys (as JSONEncoder does). The default compiler-synthesized code only produces string keys.

The encode(to:) Method

This is the code the compiler generates for the Placemark struct's encode(to:) method:

```
struct Placemark: Codable {  
    // ...  
    func encode(to encoder: Encoder) throws {  
        var container = encoder.container(keyedBy: CodingKeys.self)  
        try container.encode(name, forKey: .name)  
        try container.encode(coordinate, forKey: .coordinate)  
    }  
}
```

The main difference to the array version is that Placemark encodes itself into a *keyed* container. A keyed container is the correct choice for most composite data types (structs and classes) with more than one property (notable exception: Range uses an unkeyed container to encode its lower and upper bounds). Notice how the code passes CodingKeys.self to the encoder when it requests the keyed container. All subsequent encode commands into this container must specify a key of the same type. Since the key type is usually private to the type that's being encoded, it's nearly impossible to accidentally use another type's coding keys when implementing the encode(to:) method manually.

The end result of the encoding process is a tree of nested containers the JSON encoder can translate into its target format: keyed containers become JSON objects ({ ... }), unkeyed containers become JSON arrays ([...]), and single-value containers get converted to numbers, Booleans, strings, or null, depending on their data type.

The init(from:) Initializer

When we call try decoder.decode([Placemark].self, from: jsonData), the decoder creates an instance of the type we passed in (here it's [Placemark]) using the initializer defined in Decodable. Like encoders, decoders manage a tree of *decoding containers*, which can be any of the three familiar kinds: keyed, unkeyed, or single-value containers.

Each value being decoded then walks recursively down the container hierarchy and initializes its properties with the values it decodes from its container. If, at any step, an error is thrown (e.g. because of a type mismatch or a missing value), the entire process fails with an error.

Here's how the compiler-generated decoding initializer looks for Placemark:

```
struct Placemark: Codable {
    // ...
    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        name = try container.decode(String.self, forKey: .name)
        coordinate = try container.decode(Coordinate.self, forKey: .coordinate)
    }
}
```

Raw-Representable and Enums

We saw above how Swift's code synthesis generates the Codable implementation for structs (and classes, which work in the same way). However, there's an exception: if a type conforms to the RawRepresentable protocol and its RawValue is one of the primitive codable types (namely Bool, String, Float, Double, or one of the integer types), the raw value is encoded directly into a single-value container.

A common case of raw-representable types you might want to encode is that of enums. Enums even have a special syntax for making them raw-representable without having to manually implement an initializer and the rawValue property (see the [Enums](#) chapter for more details). Let's look at how a simplified, raw-representable version of the Surrounding enum from above is encoded:

```
enum Surrounding2: String, Codable {
    case land
    case inlandWater
    case ocean
}

struct Placemark2: Codable {
    var name: String
    var coordinate: Coordinate
    var surrounding: Surrounding2
}

let berlin = Placemark2(
    name: "Berlin",
    coordinate: Coordinate(latitude: 52, longitude: 13),
```

```

        surrounding: .land
    )
let data = try JSONEncoder().encode(berlin)
String(decoding: data, as: UTF8.self)
/*
{"name":"Berlin","coordinate":{"longitude":13,"latitude":52},
"surrounding":"land"}
*/

```

The raw value of the `Surrounding2` value (which is the name of the enum case by default) is used verbatim as the value for the "surrounding" key in the JSON dictionary. The compiler also uses the same encoding format for raw-representable structs or classes. However, note that not all enums are encoded that way: if an enum isn't raw-representable, its value is encoded in a keyed container. The key is the name of the enum case, and the value is a dictionary containing the associated values of the enum case:

```

enum Surrounding3: Codable {
    case land
    case inlandWater(name: String)
    case ocean(name: String)
}

struct Placemark3: Codable {
    var name: String
    var coordinate: Coordinate
    var surrounding: Surrounding3
}

let greatBlueHole = Placemark3(
    name: "Great Blue Hole",
    coordinate: Coordinate(latitude: 17.32278, longitude: -87.534444),
    surrounding: .ocean(name: "Caribbean Sea")
)
let data2 = try JSONEncoder().encode(greatBlueHole)
String(decoding: data2, as: UTF8.self)
/*
{"name":"Great Blue Hole",
"coordinate":{"longitude":-87.53444399999999,
"latitude":17.322780000000002},
"surrounding": "ocean"
}

```

```
"surrounding": {"ocean": {"name": "Caribbean Sea"}}}  
*/
```

Just as with structs, you can define your own coding keys for an enum to rename or skip cases. Furthermore, you can define `<CaseName>CodingKeys` types to customize the labels of associated values (or skip an associated value, if it has a default value). For example, we could add an `OceanCodingKeys` enum to `Surrounding3` to rename the label of the associated value from "name" to something else.

Manual Conformance

If your type has special requirements, you can always implement the `Encodable` and `Decodable` requirements yourself. What's nice is that the automatic code synthesis isn't an all-or-nothing thing — you can pick and choose what to override and take the rest from the compiler.

Custom Coding Keys

The easiest way to control how a type encodes itself is to write a custom `CodingKeys` enum (it doesn't have to be an enum, by the way, although only enums get synthesized implementations for the `CodingKey` protocol). Providing custom coding keys is a quick and declarative way of changing how a type is encoded. It allows us to:

- **rename fields** in the encoded output by giving them an explicit string value, or
- **skip fields altogether** by omitting their keys from the enum.

To assign different names, we also have to give the enum an explicit `raw value type` of `String`. For example, this will map `name` to "label" in the JSON output while leaving the coordinate mapping unchanged:

```
struct Placemark2: Codable {  
    var name: String  
    var coordinate: Coordinate  
  
    private enum CodingKeys: String, CodingKey {  
        case name = "label"
```

```
    case coordinate
}

// Compiler-synthesized encode and decode methods
// will use overridden CodingKeys.

}
```

And this will skip the placemark's name and only encode the GPS coordinates because we didn't include the name key in the enum:

```
struct Placemark3: Codable {
    var name: String = "(Unknown)"
    var coordinate: Coordinate

    private enum CodingKeys: CodingKey {
        case coordinate
    }
}
```

Notice the default value we had to assign to the name property. Without it, code generation for Decodable will fail when the compiler detects that it can't assign a value to name in the initializer.

Skipping properties during encoding can be useful for transient values that can easily be recomputed or aren't important to store, such as caches or memoized expensive computations. The compiler is smart enough to filter out lazy properties on its own, but if you use normal stored properties for transient values, this is how you can do it yourself.

Custom encode(to:) and init(from:) Implementations

If you need more control, there's always the option to implement encode(to:) and/or init(from:) yourself. As an example, we'll customize how our placemark type handles missing values during decoding. Here's an alternative definition of the Placemark type in which the coordinate property is optional:

```
struct Placemark4: Codable {
    var name: String
```

```
    var coordinate: Coordinate?  
}
```

By default, `JSONDecoder` will initialize an optional property of the target type with `nil` if no corresponding value exists in the input data. Accordingly, our server can now send us JSON data where the "coordinate" field is missing:

```
let validJSONInput = """  
[  
  { "name" : "Berlin" },  
  { "name" : "Cape Town" }  
]  
"""
```

When we ask `JSONDecoder` to decode this input into an array of `Placemark4` values, we'll get placemarks whose `coordinate` property is `nil`. So far so good. Now suppose the server might also send an empty JSON object instead of skipping the value altogether to signify a missing optional value:

```
let invalidJSONInput = """  
[  
  {  
    "name" : "Berlin",  
    "coordinate": {}  
  }  
]  
"""
```

When we try to decode this, the decoder, expecting the "latitude" and "longitude" fields inside the `coordinate` object, trips over the empty object and fails with a `.keyNotFound` error:

```
do {  
  let inputData = invalidJSONInput.data(using: .utf8)!  
  let decoder = JSONDecoder()  
  _ = try decoder.decode([Placemark4].self, from: inputData)  
} catch {  
  print(error.localizedDescription)  
  // The data couldn't be read because it is missing.  
}
```

To make this work, we can override the Decodable initializer and explicitly catch the error we expect:

```
struct Placemark4: Codable {
    var name: String
    var coordinate: Coordinate?

    // encode(to:) is still synthesized by compiler.

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.name = try container.decode(String.self, forKey: .name)
        do {
            self.coordinate = try container.decodeIfPresent(Coordinate.self,
                forKey: .coordinate)
        } catch DecodingError.keyNotFound {
            self.coordinate = nil
        }
    }
}
```

Now the decoder can successfully decode the faulty JSON:

```
do {
    let inputData = invalidJSONInput.data(using: .utf8)!
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark4].self, from: inputData)
    decoded // [Berlin (nil)]
} catch {
    print(error.localizedDescription)
}
```

Note that other errors, such as fully corrupted input data or any problem with the name field, will still throw.

This sort of customization is a nice option if only one or two types are affected, but it doesn't scale well. If a type has dozens of properties, you'll have to write manual code for each field, even if you only need to customize a single one. What makes the example above especially tricky to implement is that there are two variants of the JSON we want to interpret as nil for the coordinate property: either the "coordinate" key is entirely absent in the JSON, or the value of "coordinate" is an empty JSON object.

Since the "coordinate" field might be absent, we cannot use a property wrapper as a more elegant, reusable way of accepting an empty JSON object as a nil value: even if the wrappedValue of our property wrapper is optional, the property wrapper itself isn't, and the synthesized decoding code expects the key for this property to be present.

If we ease our requirements a little and assume that the "coordinate" key is always present in the JSON data, we can handle the "empty object as nil" case without manually implementing CodingKeys and init(from:) on Coordinate. Instead, we'll build a NilWhenKeyNotFound property wrapper that conforms to Decodable:

```
@propertyWrapper
struct NilWhenKeyNotFound<Value: Decodable>: Decodable {
    var wrappedValue: Value?

    init(wrappedValue: Value?) {
        self.wrappedValue = wrappedValue
    }

    init(from decoder: Decoder) throws {
        do {
            let container = try decoder.singleValueContainer()
            self.wrappedValue = try container.decode(Value.self)
        } catch DecodingError.keyNotFound {
            self.wrappedValue = nil
        }
    }
}
```

Now we can add this property wrapper to the coordinate property. If the decoding of the coordinate throws a keyNotFound error, coordinate will be nil:

```
struct Placemark5: Decodable {
    var name: String
    @NilWhenKeyNotFound var coordinate: Coordinate?
}
```

Note that we only conformed Placemark5 to Decodable, since our NilWhenKeyNotFound property wrapper currently only supports decoding. However, adding the Encodable conformance is just a matter of implementing encode(to:), which has to create an empty keyed container in case wrappedValue is nil.

For more tips on how to work with messy data in the confines of the Codable system, you might want to read Dave Lyon's article [on the topic](#). Dave came up with a generic protocol-based solution for exactly this issue. And if you have control over the input, it's always better to fix the problem at the source (make the server send valid JSON) than it is to doctor with malformed data at a later stage.

Common Coding Tasks

In this section, we'll discuss some common tasks you might want to solve with the Codable system, along with potential problems you might run into.

Making Types You Don't Own Codable

Suppose we want to replace our Coordinate type with CLLocationCoordinate2D from the Core Location framework. CLLocationCoordinate2D has the exact same structure as Coordinate, so it makes sense not to reinvent the wheel.

The problem is that CLLocationCoordinate2D doesn't conform to Codable. As a result, the compiler will now (correctly) complain that it can't synthesize the Codable conformance for Placemark5 any longer because one of its properties isn't Codable itself:

```
import CoreLocation

struct Placemark5: Codable {
    var name: String
    var coordinate: CLLocationCoordinate2D
}

// Error: cannot automatically synthesize 'Decodable'/'Encodable'
// because 'CLLocationCoordinate2D' does not conform.
```

Can we make CLLocationCoordinate2D codable despite the fact that the type is defined in another module? Adding the missing conformance in an extension produces an error:

```
extension CLLocationCoordinate2D: Codable {}

// Error: implementation of 'Encodable' cannot be automatically
// synthesized in an extension in a different file to the type.
```

Swift will only generate code for conformances that are specified either on the type definition itself or in an extension in the same file — in this instance, we'd have to implement the protocols manually. But even if that limitation didn't exist, retroactively adding Codable conformance to a type we don't own is probably not a good idea. What if Apple decides to provide the conformance itself in a future SDK version? It's likely that Apple's implementation won't be compatible with ours, which means values encoded with our version wouldn't decode with Apple's code, and vice versa. This is a problem because a decoder can't know which implementation it should use — it only sees that it's supposed to decode a value of type CLLocationCoordinate2D.

Itai Ferber, a developer at Apple who wrote large chunks of the Codable system, [gives this advice](#):

I would actually take this a step further and recommend that any time you intend to extend someone else's type with Encodable or Decodable, you should almost certainly write a wrapper struct for it instead, unless you have reasonable guarantees that the type will never attempt to conform to these protocols on its own.

In the next section, we'll see an example that uses a wrapper struct. As for our current problem, let's go with a slightly different (but equally safe) solution: we'll provide our own Codable implementation for Placemark5, in which we'll encode the latitude and longitude values directly. This effectively hides the existence of the CLLocationCoordinate2D type from the encoders and decoders; from their perspective, it looks as if the latitude and longitude properties were defined directly on Placemark5:

```
extension Placemark5 {  
    private enum CodingKeys: String, CodingKey {  
        case name  
        case latitude = "lat"  
        case longitude = "lon"  
    }  
  
    func encode(to encoder: Encoder) throws {  
        var container = encoder.container(keyedBy: CodingKeys.self)  
        try container.encode(name, forKey: .name)  
        // Encode latitude and longitude separately.  
        try container.encode(coordinate.latitude, forKey: .latitude)  
        try container.encode(coordinate.longitude, forKey: .longitude)  
    }  
}
```

```

init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container.decode(String.self, forKey: .name)
    // Reconstruct CLLocationCoordinate2D from lat/lon.
    self.coordinate = CLLocationCoordinate2D(
        latitude: try container.decode(Double.self, forKey: .latitude),
        longitude: try container.decode(Double.self, forKey: .longitude)
    )
}
}

```

This example provides us with a good idea of the boilerplate code we'd have to write for every type if the compiler didn't generate it for us (and the synthesized implementation for the CodingKey protocol is still missing here).

Alternatively, we could've used a *nested container* to encode the coordinates.

KeyedDecodingContainer has a method named nestedContainer(keyedBy:forKey:) that creates a separate keyed container (with a separate coding key type) and stores it under the provided key. We'd add a second enum for the nested keys and encode the latitude and longitude values into the nested container (we're only showing the Encodable implementation here; Decodable follows the same pattern):

```

struct Placemark6: Encodable {
    var name: String
    var coordinate: CLLocationCoordinate2D

    private enum CodingKeys: CodingKey {
        case name
        case coordinate
    }

    // The coding keys for the nested container.
    private enum CoordinateCodingKeys: CodingKey {
        case latitude
        case longitude
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(name, forKey: .name)
    }
}

```

```

    var coordinateContainer = container.nestedContainer(
        keyedBy: CoordinateCodingKeys.self, forKey: .coordinate)
    try coordinateContainer.encode(coordinate.latitude, forKey: .latitude)
    try coordinateContainer.encode(coordinate.longitude, forKey: .longitude)
}
}

```

With this approach, we'd have effectively recreated the way the `Coordinate` type encodes itself inside our original `Placemark` struct, but without exposing the nested type to the `Codable` system at all. The resulting JSON is identical in both cases.

As you can see, the amount of code we have to write for both alternatives is significant. For this particular example, we recommend a different approach, and that's to stick with our custom `Coordinate` struct for storage and `Codable` conformance, and to expose the `CLLocationCoordinate2D` type to clients as a computed property. Because the private `_coordinate` property is `codable`, we get the `Codable` conformance for free; all we have to do is rename its key in the `CodingKeys` enum. And the client-facing `coordinate` property has the type our clients require, but the `Codable` system will ignore it because it's a computed property:

```

struct Placemark7: Codable {
    var name: String
    private var _coordinate: Coordinate
    var coordinate: CLLocationCoordinate2D {
        get {
            return CLLocationCoordinate2D(latitude: _coordinate.latitude,
                                         longitude: _coordinate.longitude)
        }
        set {
            _coordinate = Coordinate(latitude: newValue.latitude,
                                      longitude: newValue.longitude)
        }
    }

    private enum CodingKeys: String, CodingKey {
        case name
        case _coordinate = "coordinate"
    }
}

```

This approach works well in this case because `CLLocationCoordinate2D` is such a simple type, and translating between it and our custom type is easy.

Making Classes Codable

We saw in the previous section that it's possible (but not advisable) to retroactively conform any value type to `Codable`. However, this isn't the case for non-final classes.

As a general rule, the `Codable` system works just fine with classes, but the potential existence of subclasses adds another level of complexity. What happens if we try to conform, say, `UIColor` to `Decodable`? (We're ignoring `Encodable` for the moment because it's not relevant for this discussion; we can add it later.) We got this example from a [message by Jordan Rose](#) on the swift-evolution mailing list.

A custom `Decodable` implementation for `UIColor` might look like this:

```
extension UIColor: Decodable {
    private enum CodingKeys: CodingKey {
        case red
        case green
        case blue
        case alpha
    }

    // Error: initializer requirement 'init(from:)' can only be satisfied
    // by a `required` initializer in the definition of non-final class 'UIColor'
    // etc.
    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        let red = try container.decode(CGFloat.self, forKey: .red)
        let green = try container.decode(CGFloat.self, forKey: .green)
        let blue = try container.decode(CGFloat.self, forKey: .blue)
        let alpha = try container.decode(CGFloat.self, forKey: .alpha)
        self.init(red: red, green: green, blue: blue, alpha: alpha)
    }
}
```

This code fails to compile, and it has several errors that ultimately boil down to one unsolvable conflict: only *required initializers* can satisfy protocol requirements, and

required initializers can't be added in extensions; they must be declared directly in the class definition.

A required initializer (marked with the `required` keyword) indicates an initializer that every subclass must implement. The rule that initializers defined in protocols must be `required` ensures that they, like all protocol requirements, can be invoked dynamically on subclasses. The compiler must guarantee that code like this works:

```
func decodeDynamic(_ colorType: UIColor.Type,  
    from decoder: Decoder) throws -> UIColor {  
    return try colorType.init(from: decoder)  
}  
let color = decodeDynamic(SomeUIColorSubclass.self, from: someDecoder)
```

For this dynamic dispatch to work, the compiler must create an entry for the initializer in the class's dispatch table. This table of the class's non-final methods is created with a fixed size when the class definition is compiled; extensions can't add entries retroactively. This is why the required initializer is only allowed in the class definition.

Long story short: it's impossible to retroactively conform a non-final class to `Codable`. In the mailing list message we referenced above, Jordan discusses a number of scenarios detailing how Swift could make this work in the future — from allowing a required initializer to be final (then it wouldn't need an entry in the dispatch table), to adding runtime checks that would trap if a subclass didn't provide the *designated initializer* that the required initializer calls.

But even then, we'd still have to deal with the fact that adding `Codable` conformance to types you don't own is problematic. As in the previous section, the recommended approach is to write a wrapper struct for `UIColor` and make that codable.

One approach would be to store the `UIColor` value in a property of the wrapper struct and then write a custom `Codable` implementation, defining coding keys for the red, green, blue, and alpha components of the color. However, we could also write a struct that has stored properties for the color components and derives a `UIColor` value from the components when needed. The advantage of this approach is that we can rely completely on code synthesis for the wrapper struct:

```
@propertyWrapper  
struct CodedAsRGBA: Codable {  
    private var red: CGFloat = 0  
    private var green: CGFloat = 0
```

```
private var blue: CGFloat = 0
private var alpha: CGFloat = 0

var wrappedValue: UIColor {
    get { UIColor(red: red, green: green, blue: blue, alpha: alpha) }
    set { store(newValue) }
}

init(wrappedValue: UIColor) {
    store(wrappedValue)
}

mutating func store(_ color: UIColor) {
    let success: Bool =
        color.getRed(&red, green: &green, blue: &blue, alpha: &alpha)
    if !success {
        fatalError("Invalid color format")
    }
}
```

Note that this simple wrapper only supports UIColor instances that can be represented in RGBA. A more complete version of this wrapper would have to check for the color space and store it together with the relevant components for the color space in question.

By defining this wrapper struct formally as a property wrapper, we can use it in a manner that's transparent to the outside world:

```
struct ColoredRect: Codable {
    var rect: CGRect
    @CodedAsRGBA var color: UIColor
}
```

Encoding an array of ColoredRect values produces this JSON output:

```
let rects = [ColoredRect(rect: CGRect(x: 10, y: 20, width: 100, height: 200),
    color: .yellow)]
do {
```

```
let encoder = JSONEncoder()
let jsonData = try encoder.encode(rects)
let jsonString = String(decoding: jsonData, as: UTF8.self)
// [{"color":{"red":1,"alpha":1,"blue":0,"green":1},"rect":[[10,20],[100,200]]}]
} catch {
    print(error.localizedDescription)
}
```

Decoding Polymorphic Collections

We've seen that decoders require us to pass in the *concrete type* of the value that's being decoded. This makes intuitive sense: the decoder needs a concrete type to figure out which initializer to call, and since the encoded data typically doesn't contain type information, the type must be provided by the caller. A consequence of this focus on strong typing is that there's no polymorphism in the decode step.

Suppose we want to encode an array of views, where the actual instances are `UIView` subclasses such as `UILabel` or `UIImageView`:

```
let views: [UIView] = [label, imageView, button]
```

(Let's assume for a moment that `UIView` and all its subclasses conform to `Codable`, which they currently don't.)

If we'd encode this array and then decode it again, it wouldn't come out in identical form — the concrete types of the array elements wouldn't survive. The decoder would only produce plain `UIView` objects because all it knows is that the type of the decoded data must be `[UIView].self`.

So how can we encode such a collection of polymorphic objects? The best option is to define an enum with one case per subclass we want to support. The payloads of the enum cases store the actual objects:

```
enum View: Codable {
    case view(UIView)
    case label(UILabel)
    case imageView(UIImageView)
    // ...
}
```

We should also write two convenience functions for wrapping a `UIView` in a `View` value, and vice versa. That way, passing the source array to the encoder and getting it from the decoder only takes a single map.

Note that this isn't a dynamic solution: we'll have to manually update the `View` enum every time we want to support another subclass. This is inconvenient, but it does make sense that we're forced to explicitly name every type our code can accept from a decoder. Anything else would be a potential security risk, because an attacker could use a manipulated archive to instantiate unknown objects in our program.

Recap

Swift's ability to seamlessly convert between a program's native types and common data formats with minimal code — at least for the common case — saves us from writing and maintaining a ton of glue code. The `Codable` system becomes even more powerful if you can use Swift on both client and server: using the same types everywhere ensures all platforms produce compatible encoding formats. Overriding the default behavior is always possible, although it's sometimes inconvenient when you need to handle non-`Codable` types you haven't defined yourself.

We only discussed traditional archiving tasks in this chapter, but it's worth thinking outside the box to find other applications that can profit from a standardized way of reducing values to primitive data and vice versa. For example, you could use the `Decodable` system as a replacement for reflection to [generate SQL queries](#) from decodable values. Or, you could write [a decoder that can produce random values](#) for each of the primitive data types and use that to generate randomized test data for your unit tests.

That being said, the system shines when you use it for the task it was designed for — working with uniform data in a known format in a fully type-safe manner. Beyond the common case, there's a point when trying to customize the `Codable` system to fit your requirements becomes more work than work saved. The Swift Core Team acknowledges as much in [a March 2021 forum post](#):

The core team felt it was important to share that while `Codable` plays a critical role in the Swift ecosystem, the core team does not see `Codable` as the end-state of serialization in Swift. By design, `Codable` solves only a subset of

the serialization needs, and Swift needs to gain additional capabilities to have a more complete serialization solution.

The core team would like to initiate this conversation with the community to gather requirements and discuss future designs and their trade-offs, ranging from improvements to Codable to additional tools and APIs. Our goal is to use the information gathered in this thread to inform future proposals that will bring a more complete serialization solution to Swift.

It'll be interesting to see what comes out of this discussion.

Interoperability

15

One of Swift's strengths is the low friction when interoperating with Objective-C and C. Swift can automatically bridge Objective-C types to native Swift types, and it can even bridge with many C types. This allows us to use existing libraries and provide a nice interface on top.

In this chapter, we'll create a wrapper around the C reference implementation of [CommonMark](#). CommonMark is a formal specification for Markdown, which is a popular syntax for formatting plain text. If you've ever written a post on GitHub or Stack Overflow, you've probably used Markdown. After this practical example, we'll take a look at the tools the standard library provides for working with memory, and we'll see how they can be used to interact with C code.

Wrapping a C Library

Swift's ability to call into C code allows us to take advantage of the abundance of existing C libraries. C APIs are often clunky, and memory management is tricky, but writing a wrapper around an existing library's interface in Swift is often much easier and involves less work than building something from scratch; meanwhile, users of our wrapper will see no difference in terms of type safety or ease of use when compared to a fully native solution. All we need to start is the dynamic library and its C header files.

Our example, the CommonMark C library, is a reference implementation of the CommonMark spec that's both fast and well tested. In this tutorial, we'll take a layered approach to make CommonMark accessible from Swift. First, we'll create a thin Swift class around the opaque types the library exposes. Then, we'll wrap this class with Swift enums to provide a more idiomatic API.

Package Manager Setup

Setting up a Swift Package Manager project to import a C library isn't as complicated as it used to be, but it still involves a few steps. Here's a quick rundown of what's required.

The first step is to install the [cmark library](#) using [Homebrew](#) as the package manager on macOS. Open a terminal and type this command to install the library:

```
$ brew install cmark
```

If you're running another OS, try installing cmark via your system's package manager. At the time of writing, cmark 0.30.2 was the most recent version.

Next, set up a new SwiftPM project. Change to the directory where you store your code. Then, enter the following commands to create a subdirectory for the project and to create a SwiftPM package for an executable:

```
$ mkdir CommonMarkExample  
$ cd CommonMarkExample  
$ swift package init --type executable
```

At this point, you can type `swift run` to check that things are working. The package manager should build and run the program, printing "Hello, world!" to the console.

Now you need a way to tell Swift about the cmark C library so that you can call it from Swift. In C, you'd `#include` one or more of the library's header files to make their declarations visible to your code. Swift can't handle C header files directly; it expects dependencies to be *modules*. For a C or Objective-C library to be visible to the Swift compiler, the library must provide a *module map* in the Clang modules format. Among other things, the module map lists the header files that make up the module.

Because cmark doesn't come with a module map, your next task is to create a SwiftPM target for the cmark library and write a module map. This target won't contain any code; its only purpose is to act as the module wrapper for the cmark library.

Open `Package.swift` and edit it to look like this:

```
// swift-tools-version:5.5  
import PackageDescription  
  
let package = Package(  
    name: "CommonMarkExample",  
    dependencies: [],  
    targets: [  
        .executableTarget(  
            name: "CommonMarkExample",  
            dependencies: ["Ccmark"]),  
        .systemLibrary(  
            name: "Ccmark",  
            pkgConfig: "libcmark",  
            providers: [
```

```
.brew(["cmark"]),
.apt(["cmark"]),
],
)
)
```

(To keep the listing short, we removed the comments and the test target SwiftPM creates by default; you can keep them, of course.)

You added a *system library target* for cmark to the package manifest. In SwiftPM lingo, *system libraries* are libraries installed by systemwide package managers, such as Homebrew, or APT on Linux. A system library target is any SwiftPM target that refers to such a library. By convention, the names of pure wrapper modules like this should be prefixed with C, which is why the target is named Ccmark.

The `pkgConfig` parameter specifies the name of the [config](#) file where the package manager can find the header and library search paths for the imported library. The `providers` directive is optional. It's an installation hint the package manager can display when the target library isn't installed.

Note that you also have to include the "Ccmark" target as a dependency of your main executable target in the package manifest. The `dependencies: ["Ccmark"]` line takes care of this.

Next, create a directory for sources of the system library target; this is where the module map goes:

```
$ mkdir Sources/Ccmark
```

Before you write the module map, create a C header file named `shim.h` in the new directory. It should contain only the following line:

```
#include <cmark.h>
```

Finally, the `module.modulemap` file should look like this:

```
module Ccmark [system] {  
    header "shim.h"  
    link "cmark"  
    export *  
}
```

The shim header works around the limitation that module maps must contain absolute paths. Alternatively, you could've omitted the shim and specified the cmark header directly in the module map, as in header "/usr/local/include/cmark.h". But then the path of cmark.h would be hardcoded into the module map. With the shim, the package manager reads the correct header search path from the pkg-config file and adds it to the compiler invocation.

Now you should be able to import Ccmark and call any cmark API. Add the following snippet to main.swift to quickly check if everything works:

```
import Ccmark  
  
let markdown = "*Hello World*"  
let cString = cmark_markdown_to_html(markdown, markdown.utf8.count, 0)!  
defer { free(cString) }  
let html = String(cString:cString)  
print(html)
```

Back in the terminal, run the program:

```
$ swift run
```

If you see <p>Hello World</p> as the output, you just successfully called a C function from Swift! And now that you have a working installation, you can start writing your Swift wrapper. (If you'd like to use Xcode for editing and running your code, you can open the package directory directly in Xcode.)

You can also embed C code inside a package by [creating a C language target](#). Depending on the library, getting it to build can be a lot more work, but it does have the advantage that users of the package don't have to install a dynamic library. When you're targeting a platform like iOS — or even when writing a macOS app — you can't assume that a dynamic library such as CommonMark

is installed, and creating a C language target is a better solution. Apple does this for [its cmark fork](#), which forms the basis of the [Swift Markdown](#) package.

Wrapping the CommonMark Library

Now that everything is set up, let's begin by wrapping a single function with a nicer interface. The `cmark_markdown_to_html` function takes in Markdown-formatted text and returns the resulting HTML code in a string. The C interface looks like this:

```
/// Convert 'text' (assumed to be a UTF-8 encoded string with length
/// 'len') from CommonMark Markdown to HTML, returning a null-terminated,
/// UTF-8-encoded string. It is the caller's responsibility
/// to free the returned buffer.
char *cmark_markdown_to_html(const char *text, size_t len, int options);
```

When Swift imports this declaration, it presents the C string in the first parameter as an `UnsafePointer` to a number of `CChar` (a type alias for `Int8` or `UInt8`, depending on the target platform) values. From the documentation, we know that these are expected to be UTF-8 code units. The `len` parameter takes the length of the string:

```
// The function's interface in Swift.
func cmark_markdown_to_html(
    _ text: UnsafePointer<CChar>!, _ len: Int, _ options: Int32)
-> UnsafeMutablePointer<CChar>!
```

We want our wrapper function to work with Swift strings, of course, so you might think we need to convert the Swift string into a `CChar` pointer before passing it to `cmark_markdown_to_html`. However, bridging between native and C strings is such a common operation that Swift will do this automatically. We do have to be careful with the `len` parameter, as the function expects the length of the UTF-8-encoded string in bytes, and not as the number of characters. We get the correct value from the string's `utf8` view, and we can just pass in zero for the options:

```
func markdownToHTML(input: String) -> String {
    let outString = cmark_markdown_to_html(input, input.utf8.count, 0)!
    defer { free(outString) }
    return String(cString: outString)
}
```

Notice that we force-unwrap the string pointer the function returns. We can safely do this because we know that `cmark_markdown_to_html` always returns a valid string. By force-unwrapping inside the method, the library user can call the `markdownToHTML` method without having to worry about optionals — the result would never be `nil` anyway. This is something the compiler can't do automatically for us — C and Objective-C pointers without nullability annotations are always imported into Swift as implicitly unwrapped optionals.

The automatic bridging of native Swift strings to C strings assumes that the C function you want to call expects the string to be UTF-8 encoded. This is the correct choice in most cases, but if the C API assumes a different encoding, you can't use the automatic bridging. However, it's often quite easy to construct alternative formats. For example, if the C API expects an array of UTF-16 code points, you can use `Array(string.utf16)`. The Swift compiler will automatically bridge the Swift array to the expected C array, provided that the element types match.

Also observe that we call `free` inside `markdownToHTML` to release the memory `cmark_markdown_to_html` allocated for the output string. When interacting with C APIs, we're responsible for following the C library's memory management rules — the Swift compiler can't help us with that.

Wrapping the `cmark_node` Type

In addition to the straight HTML output, the cmark library also provides a way to parse a Markdown text into a structured tree of elements. For example, a simple text could be transformed into a list of block-level nodes such as paragraphs, quotes, lists, code blocks, headers, and so on. Some block-level elements contain other block-level elements (for example, quotes can contain multiple paragraphs), whereas others contain only inline elements (for example, a header can contain an emphasized word). No element can contain both (for example, the inline elements of a list item are always wrapped in a paragraph element).

The C library uses a single data type, `cmark_node`, to represent any node. It's opaque, meaning the authors of the library chose to hide its definition. All we see in the headers are functions that operate on or return pointers to `cmark_node`. Swift imports these pointers as `OpaquePointers`. (We'll take a closer look at the differences between the many pointer types in the standard library, such as `OpaquePointer` and `UnsafeMutablePointer`, later in this chapter.)

Let's wrap a node in a native Swift type to make it easier to work with. As we saw in the [Structs and Classes](#) chapter, we need to think about storage semantics whenever we create a custom type: Is the type a value, or does it make sense for instances to have identity? In the former case, we should favor a struct or enum, whereas the latter requires a class. Our case is interesting: on one hand, the node of a Markdown document is a value — two nodes that have the same element type and contents should be indistinguishable, hence they shouldn't have identity. On the other hand, since we don't know the internals of `cmark_node`, there's no straightforward way to make a copy of a node, so we can't guarantee value semantics. For this reason, we start with a class. Later on, we'll write another layer on top of this class to provide an interface with value semantics.

Our class stores the opaque pointer and frees the memory `cmark_node` uses on `deinit` when there are no references left to an instance of this class. We only free memory at the document level, because otherwise we might free nodes that are still in use. Freeing the document will also automatically free all the children recursively. Wrapping the opaque pointer in this way will give us automatic reference counting for free:

```
public class Node {
    let node: OpaquePointer

    init(node: OpaquePointer) {
        self.node = node
    }

    deinit {
        guard type == CMARK_NODE_DOCUMENT else { return }
        cmark_node_free(node)
    }
}
```

The next step is to wrap the `cmark_parse_document` function, which parses a Markdown text and returns the document's root node. It takes the same arguments as `cmark_markdown_to_html`: the string, its length, and an integer describing parse options. The return type of the `cmark_parse_document` function in Swift is `OpaquePointer`, which represents the node:

```
func cmark_parse_document
(_ buffer: UnsafePointer<Int8>!, _ len: Int, _ options: Int32)
-> OpaquePointer!
```

We turn the function into an initializer for our class:

```
public init(markdown: String) {
    let node = cmark_parse_document(markdown, markdown.utf8.count, 0)!
    self.node = node
}
```

Again, we force-unwrap the returned pointer because we're certain `cmark_parse_document` won't fail — any input string is valid Markdown. Like many C APIs, the `cmark` library has no nullability annotations and no clear documentation saying which pointers can be `NULL`. If you're in doubt as to whether a particular pointer is nullable or not, it's a good idea to verify your assumptions by working through the C library's source.

As mentioned above, there are a couple of interesting functions that operate on nodes. For example, there's one that returns the type of a node, such as paragraph or header:

```
cmark_node_type cmark_node_get_type(cmark_node *node);
```

In Swift, it looks like this:

```
func cmark_node_get_type(_ node: OpaquePointer!) -> cmark_node_type
```

`cmark_node_type` is a C enum that has cases for the various block-level and inline elements that are defined in Markdown, as well as one case to signify errors:

```
typedef enum {
    // Error status
    CMARK_NODE_NONE,
    ...
    // Block elements
    CMARK_NODE_DOCUMENT,
    CMARK_NODE_BLOCK_QUOTE,
    ...
    // Inline elements
    CMARK_NODE_TEXT,
    CMARK_NODE_EMPH,
    ...
} cmark_node_type;
```

Swift imports plain C enums as structs containing a single UInt32 property. Additionally, for every case in an enum, a global constant is generated:

```
struct cmark_node_type: RawRepresentable, Equatable {  
    public init(_ rawValue: UInt32)  
    public init(rawValue: UInt32)  
    public var rawValue: UInt32  
}  
  
var CMARK_NODE_NONE: cmark_node_type { get }  
var CMARK_NODE_DOCUMENT: cmark_node_type { get }  
...
```

C enums are imported as structs because enums in C are really just integers. Swift has to assume that an enum variable in C has an arbitrary integer value, which is something native Swift enums aren't designed to handle. Only enums marked with the NS_ENUM macro, used by Apple in its Objective-C frameworks, are imported as native Swift enumerations.

In Swift, the type of a node should be a property of the Node data type, so we turn the cmark_node_get_type function into a computed property of our class:

```
var type: cmark_node_type {  
    cmark_node_get_type(node)  
}
```

Now we can just write node.type to get an element's type.

There are a couple more node properties we can access. For example, if a node is a list, it can have one of two list types: bulleted or ordered. All other nodes have the list type "no list." Again, Swift represents the corresponding C enum as a struct, with a top-level variable for each case, and we can write a similar wrapper property. In this case, we also provide a setter, which will come in handy later in this chapter:

```
var listType: cmark_list_type {  
    get { return cmark_node_get_list_type(node) }  
    set { cmark_node_set_list_type(node, newValue) }  
}
```

The cmark library provides similar functions for all the other node properties (such as the header level, fenced code block information, and link URLs and titles). These properties often only make sense for specific types of nodes, and we can choose to provide an interface either with an optional (e.g. for the link URL) or with a default value (e.g. the default header level is zero). This lack of type safety illustrates a major weakness of the library's C API that we can model much better in Swift. We'll talk more about this below.

Some nodes can also have children. To iterate over them, the CommonMark library provides the functions `cmark_node_first_child` and `cmark_node_next`. We want our `Node` class to provide an array of its children. To generate this array, we start with the first child and keep adding children until either `cmark_node_first_child` or `cmark_node_next` returns `nil`, thereby signaling the end of the list. Note that the pointer returned from `cmark_node_next` automatically gets converted to an optional where `nil` represents the null pointer:

```
var children: [Node] {
    var result: [Node] = []
    var child = cmark_node_first_child(node)
    while let unwrapped = child {
        result.append(Node(node: unwrapped))
        child = cmark_node_next(child)
    }
    return result
}
```

We could also have chosen to return a lazy sequence rather than an array (for example, by using `sequence` or `AnySequence`). However, there's a problem with this idea: the node structure might change between creation and consumption of the sequence. In such a case, the iterator for finding the next node would return incorrect values, or even worse, crash. Depending on your use case, returning a lazily constructed sequence might be exactly what you want, but if your data structure can change, returning an array is a much safer choice.

With this simple wrapper class for nodes, accessing the abstract syntax tree produced by the CommonMark library from Swift becomes a lot easier. Instead of having to call functions like `cmark_node_get_list_type`, we can just write `node.listType` and get autocompletion and type safety. However, we aren't done yet. Even though the `Node` class feels much more native than the C functions, Swift allows us to express a node in an even more natural and safer way by using enums with associated values.

A Safer Interface

As we mentioned above, there are many node properties that only apply in certain contexts. For example, it doesn't make any sense to access the `headerLevel` of a list or the `listType` of a code block. As we saw in the [Enums](#) chapter, enums with associated values allow us to specify only the metadata that makes sense for each specific case. We'll create one enum for all possible inline elements, and another one for block-level items. These two enums will be the public interface to our library, thereby turning the `Node` class into an internal implementation detail.

That way, we can enforce the structure of a CommonMark document. For example, a plain text element just stores a `String`, whereas emphasis nodes contain an array of other inline elements but mustn't have any block-level children. Here's the enum for inline elements:

```
public enum Inline {  
    case text(text: String)  
    case softBreak  
    case lineBreak  
    case code(text: String)  
    case html(text: String)  
    case emphasis(children: [Inline])  
    case strong(children: [Inline])  
    case custom(literal: String)  
    case link(children: [Inline], title: String?, url: String)  
    case image(children: [Inline], title: String?, url: String)  
}
```

For block-level items, there are also specific rules for what other items they can contain. Paragraphs and headers can only contain inline elements, whereas block quotations always contain other block-level elements. Our `Block` type models these constraints:

```
public enum Block {  
    case list(items: [[Block]], type: ListType)  
    case blockQuote(items: [Block])  
    case codeBlock(text: String, language: String?)  
    case html(text: String)  
    case paragraph(text: [Inline])  
    case heading(text: [Inline], level: Int)  
    case custom(literal: String)
```

```
    case thematicBreak  
}
```

Observe that a list is defined as an array of list items, where each list item is represented by an array of Block elements. The ListType is a simple enum that states whether a list is ordered or unordered:

```
public enum ListType {  
    case unordered  
    case ordered  
}
```

Since enums are value types, this design also allows us to treat nodes as values by converting them to their enum representations.

Recall that this wasn't possible with the class wrapper around the opaque node pointer. We follow the [API Design Guidelines](#) by using initializers for type conversions. We write two pairs of initializers: one pair creates Block and Inline values from the Node class, and another pair reconstructs a Node from these enums. This allows us to write functions that create or manipulate Inline or Block values and then later reconstruct a CommonMark document composed of Node instances with the goal of using the C library to render the document into HTML or back into Markdown text.

Let's start by writing an initializer that converts a `Node` into an `Inline` element. We switch on the node's type and construct the corresponding `Inline` value. For example, for a text node, we take the node's string contents, which we access through the `literal` property in the `cmark` library. We can safely force-unwrap `literal` because we know that text nodes always have this value, whereas other node types might return `nil` from `literal`. For instance, emphasis and strong nodes only have child nodes and no literal value. To parse the latter, we map over the node's children and call our initializer recursively. Rather than duplicating that code, we create an inline function, `inlineChildren`, that only gets called when needed. We left out most of the other cases for the sake of brevity. The default case should never get reached, so we choose to trap the program if it does. This follows the convention that returning an optional or using `throws` should generally only be used for expected errors and not to signify programmer errors:

```
extension Inline {  
    init(_ node: Node) {  
        let inlineChildren = { node.children.map(Inline.init) }  
        switch node.type {  
            case CMARK_NODE_TEXT:
```

```

    self = .text(text: node.literal!)
case CMARK_NODE_STRONG:
    self = .strong(children: inlineChildren())
case CMARK_NODE_IMAGE:
    self = .image(children: inlineChildren(),
        title: node.title, url: node.urlString)
// ... (more cases)
default:
    fatalError("Unrecognized node: \(node.typeString)")
}
}
}

```

The conversion of block-level elements follows the same pattern. Note that block-level elements can have inline elements, list items, or other block-level elements as children, depending upon the node type. In the `cmark_node` syntax tree, list items get wrapped with an extra node. We remove that layer in the `listItem` property on `Node` and directly return an array of block-level elements:

```

extension Block {
    init(_ node: Node) {
        let parseInlineChildren = { node.children.map(Inline.init) }
        switch node.type {
            case CMARK_NODE_PARAGRAPH:
                self = .paragraph(text: parseInlineChildren())
            case CMARK_NODE_LIST:
                let type: ListType = node.listType == CMARK_BULLET_LIST ?
                    .unordered : .ordered
                self = .list(items: node.children.map { $0.listItem }, type: type)
            case CMARK_NODE_HEADING:
                self = .heading(text: parseInlineChildren(), level: node.headerLevel)
// ... (more cases)
            default:
                fatalError("Unrecognized node: \(node.typeString)")
        }
    }
}

```

Now, given a document-level `Node`, we can convert it into an array of `Block` elements:

```
extension Node {  
    public var elements: [Block] {  
        children.map(Block.init)  
    }  
}
```

The Block elements are values: we can freely copy or change them without having to worry about references. This is very powerful for manipulating nodes. Since values, by definition, don't care how they were created, we can also create a Markdown syntax tree in code, from scratch, without using the CommonMark library at all. The types are much clearer too; you can't accidentally do things that wouldn't make sense — such as accessing the title of a list — as the compiler won't allow it. Aside from making your code safer, this is a very robust form of documentation — by just looking at the types, it's obvious how a CommonMark document is structured. And unlike comments, the compiler will make sure that this form of documentation is never outdated.

It's now easy to write functions that operate on our new data types. For example, if we want to build a list of all the level one and two headers from a Markdown document for a table of contents, we can loop over all children and check if they're headers and have the correct level:

```
func tableOfContents(document: String) -> [Block] {  
    let blocks = Node(markdown: document).children.map(Block.init) ?? []  
    return blocks.filter {  
        switch $0 {  
            case .heading(_, let level) where level < 3: return true  
            default: return false  
        }  
    }  
}
```

But before we build more operations like this, let's tackle the inverse transformation: converting a Block back into a Node. We need this transformation because we ultimately want to use the CommonMark library to generate HTML or other text formats from the Markdown syntax tree we've built or manipulated, and the library can only deal with `cmark_node_type`.

Our plan is to add two initializers on `Node`: one that converts an `Inline` value to a node, and another that handles `Block` elements. We start by extending `Node` with a new initializer that creates a new `cmark_node` from scratch with the specified type and `children`. Recall that we wrote a `deinit`, which frees the root node of the tree (and

recursively, all its children). This deinit will make sure that the node we allocate here gets freed eventually:

```
extension Node {  
    convenience init(type: cmark_node_type, children: [Node] = []) {  
        self.init(node: cmark_node_new(type))  
        for child in children {  
            cmark_node_append_child(node, child.node)  
        }  
    }  
}
```

We'll frequently need to create text-only nodes, or nodes with a number of children, so let's add three convenience initializers to make that easier:

```
extension Node {  
    convenience init(type: cmark_node_type, literal: String) {  
        self.init(type: type)  
        self.literal = literal  
    }  
    convenience init(type: cmark_node_type, blocks: [Block]) {  
        self.init(type: type, children: blocks.map(Node.init))  
    }  
    convenience init(type: cmark_node_type, elements: [Inline]) {  
        self.init(type: type, children: elements.map(Node.init))  
    }  
}
```

Now we can use the convenience initializers we just defined to write the conversion initializers. We switch on the input and create a node with the correct type. Here's the version for inline elements:

```
extension Node {  
    convenience init(element: Inline) {  
        switch element {  
            case .text(let text):  
                self.init(type: CMARK_NODE_TEXT, literal: text)  
            case .emphasis(let children):  
                self.init(type: CMARK_NODE_EMPH, elements: children)  
            case .html(let text):  
                self.init(type: CMARK_NODE_HTML_INLINE, literal: text)  
        }  
    }  
}
```

```

    case .custom(let literal):
        self.init(type: CMARK_NODE_CUSTOM_INLINE, literal: literal)
    case let .link(children, title, url):
        self.init(type: CMARK_NODE_LINK, elements: children)
        self.title = title
        self.urlString = url
    // ... (more cases)
}
}
}

```

Creating a node from a block-level element is similar. The only slightly more complicated case is that of lists. Recall that in the above conversion from Node to Block, we removed the extra node the CommonMark library uses to represent lists, so we need to add that back in:

```

extension Node {
convenience init(block: Block) {
    switch block {
    case .paragraph(let children):
        self.init(type: CMARK_NODE_PARAGRAPH, elements: children)
    case let .list(items, type):
        let listItems = items.map { Node(type: CMARK_NODE_ITEM, blocks: $0) }
        self.init(type: CMARK_NODE_LIST, children: listItems)
        listType = type == .unordered
            ? CMARK_BULLET_LIST
            : CMARK_ORDERED_LIST
    case .blockQuote(let items):
        self.init(type: CMARK_NODE_BLOCK_QUOTE, blocks: items)
    case let .codeBlock(text, language):
        self.init(type: CMARK_NODE_CODE_BLOCK, literal: text)
        fenceInfo = language
    // ... (more cases)
}
}
}

```

Finally, to provide a nice interface for the user, we define a public initializer that takes an array of block-level elements and produces a document node, which we can then render into one of the different output formats:

```
extension Node {  
    public convenience init(blocks: [Block]) {  
        self.init(type: CMARK_NODE_DOCUMENT, blocks: blocks)  
    }  
}
```

Now we can go in both directions: we can load a document, convert it into [Block] elements, modify those elements, and turn them back into a Node. This allows us to write programs that extract information from Markdown or even change the Markdown dynamically.

By first creating a thin wrapper around the C library (the Node class), we abstracted the conversion from the underlying C API. This allowed us to focus on providing an interface that feels like idiomatic Swift. The entire project is [available on GitHub](#).

An Overview of Low-Level Types

There are many types in the standard library that provide low-level access to memory. Their sheer number can be overwhelming, as can their daunting names, like `UnsafeMutableRawBufferPointer`. The good news is that they're named consistently, so each type's purpose can be deduced from its name. Here are the most important naming parts:

- A **managed** type has automatic memory management. The compiler will take care of allocating, initializing, and freeing the memory for you.
- An **unsafe** type eschews Swift's usual safety features, such as bounds checks or init-before-use guarantees. It also doesn't provide automated memory management — you have to allocate, initialize, deinitialize, and deallocate the memory explicitly.
- A **buffer** type works on multiple (contiguously stored) elements rather than a single element, and it provides a Collection interface.
- A **pointer** type has pointer semantics (just like a C pointer).
- A **raw** type contains untyped data. It's the equivalent of `void*` in C. Types that don't contain `raw` in their name have typed data and are generic over their respective element type.
- A **mutable** type allows the mutation of the memory it points to.

If you want direct memory access but don't need to interact with C, you can use the `ManagedBuffer` class to allocate the memory. This is similar to what the standard library's collection types use under the hood to manage their memory. It consists of a single header value (for storing data such as the number of elements) and contiguous memory for the elements. It also has a `capacity` property, which isn't the same as the number of actual elements: for example, an `Array` with a count of 17 might own a buffer with a capacity of 32, meaning that 15 more elements can be added before the `Array` has to allocate more memory. There's also a variant called `ManagedBufferPointer`, but it doesn't have many applications outside the standard library and may be removed in the future.

Sometimes you need to do manual memory management. For example, you might want to pass a Swift object to a C function with the goal of retrieving it later. To work around C's lack of closures, C APIs that use callbacks (function pointers) often take an additional context argument (usually an untyped pointer, i.e. `void*`) that they pass on to the callback. When you call such a function from Swift, it'd be convenient to be able to pass a native Swift object as the context value, but C can't handle Swift objects directly. This is where the `Unmanaged` type comes in. It's a wrapper for a class instance that provides a raw pointer to itself that we can pass to C. Because objects wrapped in `Unmanaged` live outside Swift's memory management system, we have to take care to balance `retain` and `release` calls manually. We'll look at an example of this in the next section.

Pointers

In addition to the `OpaquePointer` type we've already seen, Swift has eight more pointer types that map to different classes of C pointers.

The base type, `UnsafePointer`, is similar to a `const` pointer in C. It's generic over the data type of the memory it points to, so `UnsafePointer<Int>` corresponds to `const int*`.

Notice that C differentiates between `const int*` (a *mutable pointer to immutable data*, i.e. you can't write to the pointed data using this pointer) and `int* const` (an *immutable pointer*, i.e. you can't change where this pointer points to). `UnsafePointer` in Swift is equivalent to the former variant. As always, you control the mutability of the pointer itself by declaring the variable with `var` or `let`.

Automatic Pointer Conversion for Function Arguments

You can create an `UnsafePointer` from one of the other pointer types using an initializer. Swift also supports a special syntax for calling functions that take unsafe pointers. You can pass any *mutable* variable of the correct type to such a function by prefixing it with an ampersand, thereby making it an *in-out expression*:

```
var x = 5
func fetch(p: UnsafePointer<Int>) -> Int {
    p.pointee
}
fetch(p: &x) // 5
```

This looks exactly like the `inout` parameters we covered in the [Functions](#) chapter, and it works in a similar manner — although in this case, nothing is passed back to the caller via this value because the pointer isn't mutable. The pointer that Swift creates behind the scenes and passes to the function is guaranteed to be valid only for the duration of the function call. Don't try to return the pointer from the function and access it after the function has returned — the result is undefined.

There's also a mutable variant, `UnsafeMutablePointer`. This struct works just like a regular, non-`const` C pointer; you can dereference the pointer and change the value of the memory, which then gets passed back to the caller via the in-out expression:

```
func increment(p: UnsafeMutablePointer<Int>) {
    p.pointee += 1
}
var y = 0
increment(p: &y)
y // 1
```

The Pointer Lifecycle

Rather than using an in-out expression, you can instead allocate memory directly using `UnsafeMutablePointer`. The rules for allocating memory in Swift are similar to the rules in C: after allocating the memory, you first need to initialize it before you can use it. Once you're done with the pointer, you need to deallocate the memory:

```
// Allocate and initialize memory for two Ints.
let z = UnsafeMutablePointer<Int>.allocate(capacity: 2)
```

```
z.initialize(repeating: 42, count: 2)
z.pointee // 42
// Pointer arithmetic:
(z+1).pointee = 43
// Subscripts:
z[1] // 43
// Deallocate the memory.
z.deallocate()
// Don't access pointee after deallocate.
```

If the pointer's Pointee type (the type of data it points to) is a non-trivial type that requires memory management (e.g. a class or a struct that contains a class), you must also call deinitialize before calling deallocate. The initialize and deinitialize methods perform the reference-counting operations that make ARC work. Forgetting to call deinitialize would cause a memory leak. Even worse, failing to use initialize — for example, assigning a value to uninitialized memory via the pointer's subscript — can trigger all kinds of undefined behavior or crashes.

Raw Pointers

In C APIs, it's also common to have a pointer to a sequence of bytes with no specific element type (`void*` or `const void*`). The equivalent counterparts in Swift are the `UnsafeMutableRawPointer` and `UnsafeRawPointer` types. C APIs that use `void*` or `const void*` get imported as these types. Unless you really need to operate on raw bytes, you'd usually directly convert these types into `Unsafe[Mutable]Pointer` or other typed variants, e.g. with `load(fromByteOffset:as:)`.

Optionals Represent Nullable Pointers

Unlike C, Swift uses optionals to distinguish between nullable and non-nullable pointers. Only values with an optional pointer type can represent a null pointer. Under the hood, the memory layout of an `UnsafePointer<T>` and an `Optional<UnsafePointer<T>>` is identical; the compiler is smart enough to map the `.none` case to the all-zeros bit pattern of the null pointer.

Opaque Pointers

Sometimes a C API has an opaque pointer type. For example, in the cmark library, we saw that the type `cmark_node*` gets imported as an `OpaquePointer`. Since the definition of `cmark_node` isn't exposed in the C library's header file, the Swift compiler doesn't know the type's memory layout, which is why it can't let us access the pointee's memory. You can convert opaque pointers to other pointers using an initializer.

OpaquePointer in Swift is actually *less* type-safe than the corresponding C code because opaque pointers imported from C lose their type information. In C, `cmark_node*` and `cmark_iter*` are distinct types, and the compiler would warn you if you tried to pass one to a function that expected another. Swift imports both types as `OpaquePointer` and treats them as one and the same. Ideally, `OpaquePointer` would be generic so that `OpaquePointer<cmark_node>` and `OpaquePointer<cmark_iter>` would be distinguishable in the type system.

Buffer Pointers

In Swift, we usually use the `Array` type to store a sequence of values contiguously. In C, an array is often returned as a pointer to the first element and an element count. If we want to use such a sequence as a collection, we could turn the sequence into an `Array`, but that makes a copy of the elements. This is often a good thing (because once they're in an array, the elements are memory managed by the Swift runtime). However, sometimes you don't want to make copies of each element. For those cases, there are the `Unsafe[Mutable]BufferPointer` types. You initialize them with a pointer to the start element and a count. From then on, you have a (mutable) random-access collection. The buffer pointers make it a lot easier to work with C collections.

`Array` comes with `withUnsafe[Mutable]BufferPointer` methods that provide (mutable) access to the array's storage buffer via a buffer pointer. These APIs allow you to copy elements into or from an array in bulk, or to forgo bounds checking, which can be a performance boost in loops. Swift 5 made these methods available in generic contexts in the form of `withContiguous[Mutable]StorageIfAvailable`. Be aware that by using one of these methods, you're circumventing all the usual safety checks Swift collections perform.

Finally, the `Unsafe[Mutable]RawBufferPointer` types make it easier to work with raw memory as collections (they provide the low-level equivalent to the `Data` type in Foundation).

Closures as C Callbacks

Let's look at a concrete example of a C API that uses pointers. Our goal is to write a Swift wrapper for the `qsort` sorting function in the C standard library. The type as it's imported in Swift's Darwin module (or if you're on Linux, Glibc) is given below:

```
public func qsort(  
    __base: UnsafeMutableRawPointer!, // array to be sorted  
    __nel: Int, // number of elements  
    __width: Int, // size per element  
    __compar: @escaping @convention(c) (UnsafeRawPointer?,  
        UnsafeRawPointer?) // comparator function  
    -> Int32)
```

The man page (`man qsort`) describes how to use the `qsort` function:

The `qsort()` and `heapsort()` functions sort an array of `nel` objects, the initial member of which is pointed to by `base`. The size of each object is specified by `width`.

The contents of the array `base` are sorted in ascending order according to a comparison function pointed to by `compar`, which requires two arguments pointing to the objects being compared.

And here's a wrapper method that uses `qsort` to sort an array of Swift strings:

```
extension Array where Element == String {  
    mutating func quickSort() {  
        qsort(&self, count, MemoryLayout<String>.stride) { a, b in  
            let l = a!.assumingMemoryBound(to: String.self).pointee  
            let r = b!.assumingMemoryBound(to: String.self).pointee  
  
            if r > l { return -1 }  
            else if r == l { return 0 }  
        }  
    }  
}
```

```
    else { return 1 }
}
}
}
```

Let's look at each of the arguments being passed to `qsort`:

- The first argument is a pointer to the first element of the array that should be sorted in place. The compiler can automatically convert Swift arrays to C-style pointers when you pass them into a function that takes an `UnsafePointer`. We have to use the `&` prefix because it's an `UnsafeMutableRawPointer` (a `void *base` in the C declaration). If the function didn't need to mutate its input, and if it were declared in C as `const void *base`, the ampersand wouldn't be needed. This matches the difference with `inout` arguments in Swift functions; regular arguments don't use an ampersand, but `inout` arguments require an ampersand prefix.
- Second, we have to provide the number of elements. This one is easy; we can use the array's `count` property.
- Third, to get the width of each element, we use `MemoryLayout.stride`, *not* `MemoryLayout.size`. In Swift, `MemoryLayout.size` returns the true size of a type, but when locating elements in memory, platform alignment rules may lead to gaps between adjacent elements. The stride is the size of the type, plus some padding (which may be zero) to account for this gap. For strings, size and stride are currently the same on Apple's platforms, but this won't be the case for all types — for example, the size of an `(Int32, Bool)` tuple is 5, whereas its stride is 8. When translating code from C to Swift, you probably want to write `MemoryLayout.stride` in cases where you would've used `sizeof` in C.
- The last parameter is a pointer to a C function that's used to compare two elements from the array. Swift automatically bridges a Swift function type to a C function pointer, so we can pass any function that has a matching signature. However, there's one big caveat: C function pointers are just pointers; they can't capture any values. For that reason, the compiler will only allow you to provide functions that don't capture any external state (for example, no local variables and no generics). Swift signifies this with the `@convention(c)` attribute.

The `compar` function accepts two raw pointers. Such an `UnsafeRawPointer` can be a pointer to anything. The reason we have to deal with `UnsafeRawPointer` (and not `UnsafePointer<String>`) is because C doesn't have generics. However, we know that we get passed in a `String`, so we can interpret it as a pointer to a `String`. We also know the

pointers are never nil here, so we can safely force-unwrap them. Finally, the function needs to return an Int32: a positive number if the first element is greater than the second, zero if they're equal, and a negative number if the first is less than the second.

Making It Generic

It's easy enough to create another wrapper that works for a different type of elements; we can copy and paste the code and change String to a different type and we're done. But we should really make the code generic. This is where we hit the limit of C function pointers. The code below fails to compile because the comparison function has become a closure; it now captures things from outside its scope. More specifically, it captures the comparison and equality operators, which are different for each concrete type it's called with. There's nothing we can do about this — we simply encountered an inherent limitation of C:

```
extension Array where Element: Comparable {
    mutating func quickSort() {
        // Error: a C function pointer cannot be formed
        // from a closure that captures generic parameters.
        qsort(&self, self.count, MemoryLayout<Element>.stride) { a, b in
            let l = a!.assumingMemoryBound(to: Element.self).pointee
            let r = b!.assumingMemoryBound(to: Element.self).pointee
            if r > l { return -1 }
            else if r == l { return 0 }
            else { return 1 }
        }
    }
}
```

One way to think about this limitation is by thinking like the compiler. A C function pointer is just an address in memory that points to a block of code. For functions that don't have any context, this address will be static and known at compile time. However, in the case of a generic function, an extra parameter (the generic type) is passed in. Therefore, there are no fixed addresses for specialized generic functions. This is the same for closures. Even if the compiler could rewrite a closure in such a way that it'd be possible to pass it as a function pointer, the memory management couldn't be done automatically — there's no way to know when to release the closure.

In practice, this is a problem for many C programmers as well. On macOS, there's a variant of `qsort` called `qsort_b`, which takes a block — a closure — instead of a function pointer as the last parameter. If we replace `qsort` with `qsort_b` in the code above, it'll compile and run fine.

However, `qsort_b` isn't available on most platforms since [blocks aren't part of the C standard](#). And other functions aside from `qsort` might not have a block-based variant either. Most C APIs that work with callbacks offer a different solution: They take an extra `UnsafeRawPointer` as a parameter and pass that pointer on to the callback function. The user of the API can then use this parameter to pass an arbitrary piece of data to each invocation of the callback function. `qsort` also has a variant, `qsort_r`, which does exactly this. Its type signature includes an extra parameter, `thunk`, which is an `UnsafeMutableRawPointer`. Note that this parameter has also been added to the type of the comparison function pointer because `qsort_r` passes the value to that function on every invocation:

```
public func qsort_r(  
    __base: UnsafeMutableRawPointer!,  
    __nel: Int,  
    __width: Int,  
    __thunk: UnsafeMutableRawPointer!,  
    __compar: @escaping @convention(c)  
        (UnsafeMutableRawPointer?, UnsafeRawPointer?, UnsafeRawPointer?  
         -> Int32  
)
```

If `qsort_b` isn't available on our target platform, we can reconstruct its functionality in Swift using `qsort_r`. We can pass anything we want as the `thunk` parameter, as long as we cast it to an `UnsafeRawPointer`. In our case, we want to pass the comparison closure. Recall that the `Unmanaged` type can bridge between native Swift objects and raw pointers, which is exactly what we want. Since `Unmanaged` only works with classes, we need to wrap our closure in a class. We can reuse the `Box` class from the [Properties](#) chapter for this:

```
@propertyWrapper  
class Box<A> {  
    var wrappedValue: A  
  
    init(wrappedValue: A) {
```

```
    self.wrappedValue = wrappedValue
}
}
```

With the code above, we can start writing our variant of `qsort_b`. To stick with C's naming scheme, we're calling the function `qsort_block`. Here's the implementation:

```
typealias Comparator = (UnsafeRawPointer?, UnsafeRawPointer?) -> Int32

func qsort_block(_ array: UnsafeMutableRawPointer, _ count: Int,
                _ width: Int, _ compare: @escaping Comparator)
{
    let box = Box(wrappedValue: compare) // 1
    let unmanaged = Unmanaged.passRetained(box) // 2
    defer {
        unmanaged.release() // 6
    }
    qsort_r(array, count, width, unmanaged.toOpaque()) {
        (ctx, p1, p2) -> Int32 in // 3
        let innerUnmanaged =
            Unmanaged<Box<Comparator>>.fromOpaque(ctx!) // 4
        let comparator =
            innerUnmanaged.takeUnretainedValue().wrappedValue // 4
        return comparator(p1, p2) // 5
    }
}
```

The function performs the following steps:

0. It boxes the comparator closure in a `Box` instance.
1. Then, it wraps the box in an `Unmanaged` instance. The `passRetained` call makes sure to retain the box instance so it doesn't get deallocated prematurely (remember, you're responsible for keeping objects you put into an `Unmanaged` instance alive).
2. Next, it calls `qsort_r`, passing the pointer to the `Unmanaged` object as the `thunk` parameter (`Unmanaged.toOpaque` returns a raw pointer to itself).
3. Inside `qsort_r`'s callback, it converts the raw pointer back to an `Unmanaged` object, extracts the box, and unboxes the closure. Make sure not to modify the reference count of the wrapped object. The flow is exactly the reverse of steps 1 to 3, as

`fromOpaque` returns an `Unmanaged` object, `takeUnretainedValue` extracts the `Box` instance, and `wrappedValue` unwraps the closure.

4. After, it calls the comparator function with the two elements that should be compared.
5. Finally, after `qsort_r` returns, in the `defer` block, it releases the box instance now that we no longer need it. This balances the retain operation from step 2.

Now we can modify our `qsortWrapper` function to use `qsort_block` and provide a nice generic interface to the `qsort` algorithm from the C standard library:

```
extension Array where Element: Comparable {  
    mutating func quickSort() {  
        qsort_block(&self, self.count, MemoryLayout<Element>.stride) { a, b in  
            let l = a!.assumingMemoryBound(to: Element.self).pointee  
            let r = b!.assumingMemoryBound(to: Element.self).pointee  
            if r > l { return -1 }  
            else if r == l { return 0 }  
            else { return 1 }  
        }  
    }  
}  
  
var numbers = [3, 1, 4, 2]  
numbers.quickSort()  
numbers // [1, 2, 3, 4]
```

It might seem like a lot of work to use a sorting algorithm from the C standard library. After all, Swift's built-in `sort` function is much easier to use, and it's faster in most cases. That's certainly true, but there are many other interesting C APIs out there that we can wrap with a type-safe and generic interface using the same technique.

Recap

Rewriting an existing C library from scratch in Swift is certainly fun, but it may not be the best use of your time (unless you're doing it for the sake of learning, which is totally awesome). There's a lot of well-tested C code out there, and throwing it all out would be a huge waste. Swift is great at interfacing with C code, so why not make use of these capabilities? Having said that, there's no denying that most C APIs feel very foreign in

Swift. Moreover, it's probably not a good idea to spread C constructs like pointers and manual memory management across your entire codebase.

Writing a small wrapper that handles the unsafe parts internally and exposes an idiomatic Swift interface — as we did in this chapter for the Markdown library — gives you the best of both worlds: you don't have to reinvent the wheel (i.e. write a complete Markdown parser), yet it feels 100 percent native to developers using the API.

Final Words

16

We hope you enjoyed this journey through Swift with us.

Despite its young age, Swift is already a complex language. It'd be a daunting task to cover every aspect of it in one book, let alone expect readers to remember it all. But even if you don't immediately put everything you learned to practical use, we're confident that having a better understanding of your language makes you a more accomplished programmer.

If you take one thing away from this book, we hope it's that Swift's many advanced aspects are there to help you write better, safer, and more expressive code. While you can write Swift code that feels not much different from Objective-C, Java, or C#, we hope we've convinced you that features like enums, generics, first-class functions, and structured concurrency can greatly improve your code.

The native concurrency model is perhaps Swift's last big headline feature for a while, but that's not to say the language won't continue to improve. Here are some of the areas where we expect major enhancements in the coming years:

- **Extensions to the concurrency model.** Compile-time checking will ramp up gradually to give library developers time to audit their code for concurrency, and the Swift team is already working on distributed actors, which are an extension of the actor model to multiple processes or machines.
- **Additions to the generics system.** We mention some of these additions in the book, such as where constraints for opaque types, and the lifting of restrictions on existentials.
- **Explicit control over memory management and ownership.** The goal is to give the compiler all the information it needs to avoid unnecessary copies when passing values to functions, thereby making Swift more suitable for writing low-level code with strict performance requirements.
- **More powerful introspection.** The compiler bakes a lot of metadata about types and their properties into the binaries. This information is already being used by debugging tools, but there aren't yet any public APIs to access it. The existence of this data opens the door for more powerful reflection and introspection capabilities that go way beyond what the current Mirror type can do.
- **Major improvements to the string APIs.** The Swift team is working on support for regular expressions and a result builder-based syntax for writing parsers under the moniker of declarative string processing.

If you're interested in shaping how these and other features turn out, remember that Swift is being developed in the open. Consider joining the [Swift Forums](#) and adding your perspective to the discussions.

Finally, we'd like to encourage you to take advantage of the fact that Swift is open source. When you have a question the documentation doesn't answer, the source code can often give you the answer. If you made it this far, you'll have no problem finding your way through [the standard library source files](#). Being able to check how things are implemented there was a big help for us when writing this book.