objc ↑↓

Advanced Swift

By Chris Eidhof and Airspeed Velocity

Advanced Swift

Chris Eidhof

Airspeed Velocity

objc.io

Advanced Swift

Foreword

I started OS X development in 2005. I started iOS development in 2008. When I co-wrote an advanced iOS book in 2011, I had years of production experience, decades of Objective C best practice to draw on, and colleagues and mentors with more experience whom I could ask for advice.

Swift is close to two years old and still evolving dramatically. Best practices from six months ago are already obsolete. Can anyone write an advanced Swift book yet?

Yes — but you'd want some things from the authors. You'd want them to be have been there from the beginning. You'd want them to be involved in the community. You'd wan them to dig into the *why* of Swift as much as the *how* of Swift. You'd want them to be answering hard questions. You'd want them to be *asking* hard questions.

Additionally, there's much more to Swift than the documentation. The authors need to know what's being discussed in the forums and on Twitter. They need to have dug through the standard library to really get how it works. They need to have followed it closely enough to see where Apple is going, to separate "not Swift-like" from "not Swift yet." That's what Airspeed Velocity and Chris do, and that's why you should read what they have to say.

If you've seen any of my Swift talks, you know Airspeed Velocity is my favorite Swift blog I love how it digs into the details, and that's what this book does. And like the blog, this book isn't afraid to say where Swift is still awkward. Sometimes there is no good Swift answer yet. Sometimes Swift only solves 80 percent.

Chris's blog helped inspire me to dig into Swift's functional side, and *Functional Swift* se the bar for the subject. I highly recommend it. But, as you'll learn in this book, "Swift is not Haskell." There are plenty of higher-order functions and immutable data structures here, but there are also loops and var properties when those are the right tools. This is a book that lets Swift be Swift by embracing its strengths and acknowledging its weakness but always striving to find the Swift way.

Swift is a new world with new rules and new voices. There's still plenty of room to contribute and make an impact on the language and the community. We're all still learning what "good Swift" looks like and which patterns work best. Read what Airspeed Velocity and Chris have to say, and I hope you join the conversation.

Rob Napier (@cocoaphony)

Nov 2015

Introduction

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

When we began writing this book, Swift was barely a year old. We did so before the beta o 2.0 was released — albeit tentatively, because we suspected the language would continu to evolve as it entered its second year. Few languages — perhaps no other language — have been adopted so rapidly by so many developers.

But that left people with unanswered questions. How do you write "idiomatic" Swift? Is there a right way to do certain things? The standard library gave some clues, but even that has changed over time, dropping some conventions and adopting others.

To someone coming from another language, Swift can resemble everything you like abou your language of choice. Low-level bit twiddling can look very similar to (and can be as performant as) C, but without many of the undefined behavior gotchas. The lightweight trailing closure syntax of `map` or `filter` will be familiar to Rubyists. Swift generics are similar to C++ templates, but with type constraints to ensure generic functions are correc at the time of definition rather than at the time of use. The flexibility of higher-order functions and operator overloading means you can write code that is similar in style to Haskell or F#. And the `@objc` keyword allows you to use selectors and runtime dynamisn in ways you would in Objective-C.

Given these resemblances, it's tempting to adopt the idioms of other languages. Case in point: Objective-C example projects can almost be mechanically ported to Swift. Books were rapidly published showing Swift implementations of Java or C# design patterns. Ar a new wave of monad tutorial blog posts was born.

But then comes the frustration. Why can't we use protocol extensions with associated types like interfaces in Java? Why are arrays not covariant in the way we expect? Why can't we write "functor?" Sometimes the answer is because the part of Swift in question isn't yet implemented. But more often, it's either because there is a different Swift-like way to do what you want to do, or because the Swift feature you thought was like the equivalent in some other language is not quite what you think.

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 t know about generics or overloading or the difference between static and dynamic dispatch. You may never need to call into a C library or write your own collection type. But after a while, you'll eventually need to know about these things — either to improve your code's performance, to make it more elegant or expressive, or to just get certain things done.

Learning more about these features is what this book is about. We intend to answer man
of the “How do I do this?” or “Why does Swift behave like that?” questions we’ve seen come up on various forums. Hopefully once you’ve read it, 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 materia
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, or those coming from other languages such as Java c C++, who want to bring their knowledge of Swift to the same level as that of Objective-C o some other language. It's also suitable for new programmers who have started on Swift, grown familiar with the basics, and are looking to take things to the next level.

It's not meant as an introduction to Swift; it assumes you are familiar with the syntax an structure of the language. If you want some good, compact coverage of the basics of Swif the best source is the official Apple Swift book (available on iBooks or at developer.apple.com/swift/resources/. If you're already a confident programmer, you could try reading both our book and the Apple Swift book in parallel.

This is also not a book about programming for OS X or iOS devices. Of course, since Swift is currently mainly used on Apple platforms, we have tried to include examples of practical use, but we hope this book will be useful for non-Apple-platform programmers as well.

Themes

We've organized the book under the heading of basic concepts. There are in-depth chapters on some fundamental basic concepts like optionals or strings, and some deeper dives into topics like C interoperability. But throughout the book, hopefully a few themes regarding Swift emerge:

Swift is both a high- and low-level language. Swift allows you to write code similarly t 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 a array compiles to the same assembly code as looping over a contiguous block of memory

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, c an understanding of the difference between dynamic and static method dispatch. We'll cover topics such as these in more depth later.

Swift is a multi-paradigm language. You can use it 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 c writing Java or C or Objective-C in Swift.

Swift still has access to most of the capabilities of Objective-C, including message sendin runtime type identification, and KVO. But Swift introduces many capabilities not available in Objective-C.

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 mo functional style through its use of generics, protocols, value types, and closures. It is even possible to write operators that compose functions together. The early months of Swift brought many monad blog posts into the world. But with the release of Swift 2.0 and the introduction of protocol extensions, this trend has shifted.

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 a long way from Lisp. But still, we feel like Swift shares 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.

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 obscure rather than clarify the meaning of the code.

For example, type inference removes the clutter of type declarations that are obvious fror

the context. Semicolons and parentheses that add little or no value are gone. Generics an
~~protocol extensions encourage you to avoid repeating yourself by packaging common~~
operations into reusable functions. The goal is to write code that is readable at a glance.

At first, this can be off-putting. If you have never used functions like map, filter, and reduce before, they might look harder 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 practical, until you tell it not to be. This is unlike languag such as C and C++ (where you can be unsafe easily just by forgetting to do something), o like Haskell or Java (which are sometimes safe whether or not you like it).

Eric Lippert, one of the principal designers of C#, recently 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). However Swift, being reference counted, *does* execute a class's deinit deterministically.

Swift embodies this sentiment in other ways. Undefined and unsafe behavior is avoided by default. For example, a variable cannot be used until it has been initialized, and usin 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, o the UnsafeMutablePointer type) for when you really need them. But with great power comes great undefined behavior. You can write the following:

```
let uhOh = someArray.withUnsafePointer { ptr in
    // ptr is only valid within this block, but
    // there is nothing stopping you letting it
    // escape into the wild:
    return ptr
}
// later...
uhOh[10]
```

It will compile, but who knows what it will do. But 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 they're facts. But they're just, like, our opinions, man. Feel free to disagree! Swift is still a young language and many things aren't settled. What's more is that many blog posts are flat-~~out wrong, or outdated (including several ones we wrote, especially in the early~~ days Whatever 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. Think critically, and beware of out-of-date information.

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 is familiar on first reading. 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 distinctions 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. Those are examples of **literals**, but values can also be generated at runtime. The number that 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 are creating a **variable** named that holds the value `[1,2]`. By changing `x`, e.g. by performing `x.append(3)`, we did not 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 on 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 is no concept of a variable having an as-yet-undefine
~~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: a value that "points to" a variable. Because two references can refer to the same variable, this introduces the possibility of that variable getting mutated by two different parts of the program at once.

Classes are **reference types**. You cannot hold an instance of a class (which we might occasionally call an **object** — a term fraught with troublesome overloading!) directly in a variable. Instead, you must hold a reference to it in a variable and access it via that reference.

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

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

Class references are not the only kind of reference in Swift. For example, there are also pointers, accessed through `withUnsafeMutablePointer` functions and the like. But classe are the simplest reference type to use, in part because their reference-like nature is partially hidden from you by syntactic sugar. You don't need to do any explicit "dereferencing" like you do with pointers in some other languages. (We will cover the other kind of references in more detail in the chapters on CommonMark and interoperability.)

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 does *not* mean that the object it *refers to* cannot be changed. So when referring to a variable as a constant, be careful — it is only constant i what it points to. It does not mean what it points to is constant. (Note: if those last few sentences sound like doublespeak, don't worry, as we cover this again in the chapter on structs and classes). 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.

Here we hit another complication. While structs are value types, structs can be compose
of ~~multiple other types, and those types can be references. This means that while~~
assigning one value type to another copies the value, it is a *shallow copy*. It will copy the reference, and not the value the reference points to.

We refer to types having **value semantics** to distinguish a value type that performs a *deep copy*. For example, a Swift array has value semantics if it contains structs as the elements. Yet this is still a tricky definition. If an array contains objects, the elements in the array are references to the objects. Thus, when copying an array that contains object the objects themselves do not get copied — only the references to the objects do. In the chapter on structs and classes, we describe this behavior in more detail.

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

The collection types in the Swift standard library are structs that have value semantics. Internally, they are implemented using references. In the next chapter, we'll explain hov this differs from the way the Foundation classes behave in Objective-C. The Swift structs do this efficiently via a technique called copy-on-write, which we describe the mechanic of in the chapter on structs and classes. But for now, it's important to know that this behavior does not come "for free" whenever structs have reference fields; it has to be implemented by the author of the struct. Copy-on-write is also not the only way to create value semantics, but it's the most common one.

In Swift, functions are also values. 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 function 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 do not have to be declared at the top level — you can declare a function withi 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 variable in which case those local variables are not 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 `{ }` synta called a **closure expression**. Sometimes this gets shortened to "closures," but don't let i give you the impression that only closure expressions can be closures. Functions declare with the `func` keyword are closures too.

Functions are held by reference. This means assigning a function that has state via closed-over variables to another variable does not copy that state; it shares it, similar to obj~~ect 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 chapter on functions.

Functions defined inside a class or protocol are **methods**, and they have an implicit self parameter. A function that, instead of taking multiple arguments, takes some argument and returns another function representing the partial application of the arguments to tha function, is a **curried function**. We'll see in the functions chapter how methods are actually curried functions. Sometimes we call functions that are not methods **free functions**. This is to distinguish them from methods.

Free functions, and methods called on structs, are **statically dispatched**. This means th function that will be called is known at compile time. This also means the compiler migh 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. It can also discard or simplify code that it can prov at compile time won't actually run.

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

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 tha 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 Style Guide

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

- Readability is most important. This is helped by brevity.
- Always add documentation comments to functions — *especially* generic ones.
- Types start with `UpperCaseLetters`. Functions and variables 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,`func`s 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.
- Use the trailing closure syntax, except when that closure is immediately followed by another opening brace.
- Use `guard` to exit functions early.
- Eschew force-unwraps and implicitly unwrapped optionals. They are occasionally useful, but needing them constantly is usually a sign something else is wrong.
- Don't repeat yourself. If you find you have 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 `reduce`. 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 immutability: default to `let` unless you know you need mutation. But use mutation when it makes the code clearer or more efficient. Wrap that mutation in a function to isolate unexpected side effects.
- Swift generics tend to lead to very long function signatures. Unfortunately, we have yet to settle on a good way of breaking up long function declarations into multiple lines. We'll try to be consistent in how we do this in sample code.
- Much to the dismay of half of this book's authorship, the "cuddled else" is official Swift style: `} else {`.

Collections

Arrays and Mutability

The most common collection we use in Swift is that of arrays. An array is simply a list of things. As an example, to create an array of numbers, we can write the following:

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

There are the usual operations on array, like the isEmpty and count methods. To get the first and last elements of an array, we can use first and last, which return nil if the array empty. Arrays also allow for direct access of elements at a specific index through subscripting. Subscripting is not a safe operation; before getting an element, you need to verify the index is within bounds. Otherwise, your program crashes.

If we try to modify the array defined above (by using append, for example), we get a compiler error. This is because the array is defined as a constant, using let. In many cases, this is exactly the right 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.appendContentsOf([13, 21])
```

There are a couple of benefits that come with making the distinction between var and let Variables defined with let are easier to reason about because they are immutable. When you read a declaration like let fibs = ..., you know that the value of fibs will never change — it is enforced by the compiler. This helps greatly when reading through code. Note tha in the case of classes, the value is a reference. Defining an object with let will make sure the reference never changes. However, the memory it points to (i.e. the instance variables) *can* change. We will go into more detail on the differences between structs and classes in Chapter 4.

Swift arrays have value semantics, which means that they are never shared. When creating a new variable and assigning an existing array to it, a copy is made. This is the case when creating variables, passing arrays to functions, and more. For example, in the following code snippet, x is never modified:

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

The statement var y = x makes a copy of x, so appending 4 to y will not change x — the value of x will still be [1, 2, 3].

Contrast this with the approach to mutability taken by NSArray. NSArray has no mutatin methods — to mutate an array, you need an NSMutableArray. But just because you have a non-mutating NSArray reference does *not* mean the array can't be mutated underneath you:

```
let a = NSMutableArray(array: [1,2,3])

// I don't want to be able to mutate b
let b: NSArray = a

// but it can still be mutated - via a
a.insertObject(4, atIndex: 3)
b // now contains a 4
```

The correct way to write this is to create a copy of a when we introduce b:

```
let a = NSMutableArray(array: [1,2,3])

// I don't want to be able to mutate b
let b = a.copy() as! NSArray

a.insertObject(4, atIndex: 3)
b // still is [1,2,3]
```

In the example above, it is very clear that we need to make a copy — a is mutable, after all. However, when passing around arrays between methods and functions, this is not always so easy to see, leading to much unnecessary copying.

In Swift, instead of needing two types, there is just one, and mutability is defined by declaring with var instead of let. But there is no reference sharing — when you declare a second array with let, you are guaranteed it will never change.

Making so many copies 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 an y shared internal storage up the point where y.append was called. In the chapter on structs and classes, we'll take a deeper look at value semantics, including how to implement copy-on-write for your own types.

Transforming Arrays

The release of Swift in 2014 led to a galaxy of explanations on the benefits of map, filter, and reduce. Still there are a number of points we want to make, which we will cover briefly.

Map

It's common to need to perform a transformation on every value in the array. Every programmer has written similar code hundreds of times: create a new array, loop over al elements in an existing array, perform an operation on an element, and append the resul 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)
}
```

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

```
let squared = fibs.map { fib in fib * fib }
```

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

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

map isn't hard to write — it's just a question of wrapping up the boilerplate parts of the fo loop into a generic function. Here's one possible implementation (though in Swift, it's actually an extension of SequenceType, something we'll cover in the chapter on writing generic algorithms):

```
extension Array {
    func map<U>(transform: Element->U) -> [U] {
        var result: [U] = []
        result.reserveCapacity(self.count)
        for x in self {
            result.append(transform(x))
        }
        return result
    }
}
```

Element is the generic placeholder for whatever type the array contains. And U is a new placeholder that can represent the result of the element transformation. The map function itself does not care what Element and U are; they can be anything at all. What they are and what to do to them is left to the caller.

> *Really, the signature for this function should be func map<U>(@noescape transform: Element throws -> U) rethrows -> [U] — we'll cover @noescape in the functions chapter and rethrows in the errors chapter. Neither of these are necessary; they just make it more pleasant for the caller to use.*

Parameterizing Behavior with Functions

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

map manages to separate out the boilerplate functionality — which doesn't vary from ca to call — from the functionality that always varies: 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. There are 13 separate functions that take a closure that allows the caller to customize the key step:

- **map** and **flatMap** — how to transform an element
- **filter** — should an element be included?
- **reduce** — how to fold an element into an aggregate value
- **sort** and **lexicographicCompare** — in what order should two elements come?
- **indexOf** and **contains** — does this element match?
- **minElement** and **maxElement** — which is the min/max of two elements?
- **elementsEqual** and **startsWith** — are two elements equivalent?
- **split** — is this element a separator?

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, the for loop over the source data, and the like.

Instead, the clutter is replaced with a single word that describes what is 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 orde when they're comparable, unless you specify otherwise. contains can take a value to check for, so long as the elements are equatable. These help make the code even more readable. Ascending order sort is natural, so the meaning of array.sort() is intuitive. array.indexOf("foo") is clearer than array.indexOf { $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 whole 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-insensitive sort could be performed via people.sort { $0.name.uppercaseString < $1.name.uppercaseString }.

There are other functions of similar usefulness that would also take a closure to specify their behaviors but 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)
- **allMatch** and **noneMatch** — test if all or no elements in a sequence match a criterion (can be built with contains, with some carefully placed negation)
- **count** — count the number of elements that match (similar to filter, but without constructing an array)
- **indicesOf** — return a list of indices matching a criteria (similar to indexOf, but doesn't stop on the first one)
- **takeWhile** — filter elements while a predicate returns true, then drop the rest (similar to filter, but with an early exit, and useful for infinite or lazily-computed sequences)
- **dropWhile** — drop elements until the predicate ceases to be true, and then return the rest (similar to takeWhile, but this returns the inverse)

Many of these we define elsewhere in the book.

You might find yourself writing something that fits a pattern more than a couple of time — something like this:

```
let someArray: [SomeObject] = []
```

```
var object: SomeObject?
for oneObject in someArray where oneObject.passesTest() {
    object = oneObject
    break
}
```

If that's the case, then consider writing a short extension to SequenceType. The method findElement wraps this logic. We use a closure to abstract over the part of our for loop tha varies:

```
extension SequenceType {
    func findElement (match: Generator.Element->Bool) -> Generator.Element? {
        for element in self where match(element) {
            return element
        }
        return nil
    }
}
```

This then allows you to replace your for loop with the following:

```
let object = someArray.findElement { $0.passesTest() }
```

This has all the same benefits we described for map. The example with findElement is more readable than the example with the for loop; even though the for loop is simple, yo still have to run the loop through in your head, which is a small mental tax. Using findElement introduces less chance of error, and it allows you to declare object with let instead of var.

It also works nicely with guard — in all likelihood, you're going to terminate a flow early the element isn't found:

```
guard let object = someSequence.findElement({ $0.passesTest() })
    else { return }
// use object in body of function
```

We'll write 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 th 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 ~~transformation of the array. If you ever see something like the above, then it is a clear~~ cas
for using a for loop instead of a function like map. There is a function called forEach that would be more appropriate in this case, but it has its own issues. We will look at forEach in a bit.

This is different from deliberately giving the closure *local* state, which is quite a useful technique, and it's what makes closures — functions that can capture and mutate variables outside their scope — so powerful a tool when combined with higher-order functions. For example, the accumulate function described above could be implementec with map and a stateful closure, like this:

```
extension Array {
    func accumulate<U>(initial: U, combine: (U, Element) -> U) -> [U] {
        var running = initial
        return self.map { next in
            running = combine(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 value as it progresses:

```
[1,2,3,4].accumulate(0, combine: +)
```

```
[1, 3, 6, 10]
```

Filter

Another very common operation is to take an array and create a new array that only includes original elements that match a certain condition. The pattern of looping over ar array and filtering out the elements that match a condition is captured in the filter method on arrays:

```
fibs.filter { num in num % 2 == 0 }
```

As a final way of writing this with less code, we can use Swift's built-in notation for shorthand argument names. Instead of naming the num argument, we can write the cod above like this:

```
fibs.filter { $0 % 2 == 0 }
```

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 have done before. It' really a matter of personal taste — choose whichever is more readable at a glance. A gooc

rule of thumb is this: if the closure fits neatly on one line, shorthand argument names ar
a good fit.

By combining map and filter, we can easily write a lot of operations on arrays without having to introduce a single intermediate array. The resulting code will become shorter and easier to read. For example, to find all squares under 100 that are even, we could ma
the range 0..<10 in order to square it, 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 much the same as map:

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

For more on the where clause used in the for loop, see the optionals chapter.

One quick performance tip: 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 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 matches the first element. Generally, only ever use filter if you want all the results.

Often you want to do something that can be done by contains, but it looks pretty ugly. Fo
example, you can check if every element of a sequence matches a predicate using !sequence.contains { !condition }, but it’s much more readable to wrap this in a new function that has a more descriptive name:

```
extension SequenceType {
    public func allMatch(predicate: Generator.Element -> Bool) -> Bool {
        // every element matches a predicate if no element doesn't match it:
        return !self.contains { !predicate($0) }
    }
```