# **What's new in Swift 4.0**

**Hands-on code examples to help you learn what's new in Swift 4: new encoding and decoding, smarter keypaths, multi-line strings, and more!**

[Paul Hudson](https://www.hackingwithswift.com/about)    June 5th 2017     [@twostraws](https://twitter.com/twostraws)

Swift 4.0 is a major new release for everyone's favorite app development language, and introduces a variety of features that let us write simpler, safer code. You'll be pleased to know it's nothing as dramatic as the epic changes introduced with Swift 3.0, and indeed most changes are fully backwards-compatible with your existing Swift code. So, while you might need to make a handful of changes it shouldn't take long.

If you liked this article, you might also enjoy these:

* [What's new in iOS 11?](https://www.hackingwithswift.com/whats-new-in-ios-11)
* [What's new in Swift 3.1?](https://www.hackingwithswift.com/swift3-1)
* [What's new in Swift 3?](https://www.hackingwithswift.com/swift3)
* [Buy my Server-Side Swift book](https://www.hackingwithswift.com/store/server-side-swift)
* [Buy my Hacking with macOS book](https://www.hackingwithswift.com/store/hacking-with-macos)
* [Read my free Swift tutorial series](https://www.hackingwithswift.com/read)

## **Swifty encoding and decoding**

We know value types are great, but we also know they interact terribly with Objective-C APIs such as NSCoding – you either need to write a shim layer or give in and use classes, both of which are unpleasant. Worse, even if you give in and switch to classes, you still need to write your encoding and decoding methods by hand, which is painful and error-prone.

Swift 4 introduces a new Codable protocol that lets you serialize and deserialize custom data types without writing any special code – and without having to worry about losing your value types. Even better, you can choose how you want the data to be serialized: you can use classic property list format or even JSON.

**Yes, you read all that correctly: Swift 4 lets you serialize your custom data types to JSON without writing any special code.**

Let's take a look at how beautiful this is. First, here's a custom data type and some instances of it:

struct Language: Codable {

 var name: String

 var version: Int

}

let swift = Language(name: "Swift", version: 4)

let php = Language(name: "PHP", version: 7)

let perl = Language(name: "Perl", version: 6)

You can see I've marked the Language struct as conforming to the Codable protocol. With that one tiny addition, we can convert it to a Data representation of JSON like this:

let encoder = JSONEncoder()

if let encoded = try? encoder.encode(swift) {

 // save `encoded` somewhere

}

Swift will automatically encode all properties inside your data type – you don't need to do anything.

Now, if you're like me and have a long history of using NSCoding, you're probably somewhat doubtful: is that really all it takes, and how can we be sure it's working? Well, let's add some more code to try converting the Data object into a string so we can print it out, then decode it back into a new Language instance that we can read from:

if let encoded = try? encoder.encode(swift) {

 if let json = String(data: encoded, encoding: .utf8) {

 print(json)

 }

 let decoder = JSONDecoder()

 if let decoded = try? decoder.decode(Language.self, from: encoded) {

 print(decoded.name)

 }

}

Notice how decoding doesn't require a typecast – you provide the data type name as its first parameter, so Swift infers the return type from there.

Both JSONEncoder and its property list counterpart PropertyListEncoder have lots of options for customizing how they work: do you want compact JSON or pretty-printed JSON? Do you want to use ISO8601 dates or Unix epoch dates? Do you want to use binary property lists or XML? For more information on these and other options, see [the Swift Evolution proposal for this new feature](https://github.com/apple/swift-evolution/blob/master/proposals/0167-swift-encoders.md).

## **Multi-line string literals**

Writing multi-line strings in Swift has always meant adding \n inside your strings to add line breaks wherever you want them. This doesn't look good in code, but at least it displays correctly for users. Fortunately, Swift 4 introduces new multi-line string literal syntax that lets you add line breaks freely and use quote marks without escaping, while still benefiting from functionality like string interpolation.

To start a string literal, you need to write three double quotation marks: """ then press return. You can then go ahead and write a string as long as you want, including variables and line breaks, before ending your string by pressing return then writing three more double quotation marks.

I've been specific about pressing return because string literals have two important rules: when you open a string using """ the content of your string must begin on a new line, and when you end a multi-line string using """ that must also begin on a new line.

Here it is in action:

let longString = """

When you write a string that spans multiple

lines make sure you start its content on a

line all of its own, and end it with three

quotes also on a line of their own.

Multi-line strings also let you write "quote marks"

freely inside your strings, which is great!

"""

That creates a new string with several line breaks right there in the definition – much easier to read and write.

For more information see [the Swift Evolution proposal for this new feature](https://github.com/apple/swift-evolution/blob/master/proposals/0168-multi-line-string-literals.md).

## **Improved keypaths for key-value coding**

One of the most loved features of Objective-C is its ability to reference a property dynamically rather than directly – that is, to be able to say "given object X, here is the property I'd like to read" without actually reading it. These references, called keypaths, are distinct from direct property accesses because they don't actually read or write the value, they just stash it away for use later on.

If you've never used keypaths before, let me show you an analogy of how they work using regular Swift methods. We're going to define a struct called Starship and a struct called Crew, then create one instance of each:

// an example struct

struct Crew {

 var name: String

 var rank: String

}

// another example struct, this time with a method

struct Starship {

 var name: String

 var maxWarp: Double

 var captain: Crew

 func goToMaximumWarp() {

 print("\(name) is now travelling at warp \(maxWarp)")

 }

}

// create instances of those two structs

let janeway = Crew(name: "Kathryn Janeway", rank: "Captain")

let voyager = Starship(name: "Voyager", maxWarp: 9.975, captain: janeway)

// grab a reference to the `goToMaximumWarp()` method

let enterWarp = voyager.goToMaximumWarp

// call that reference

enterWarp()

Because functions are first-class types in Swift, the last two lines are able to create a reference to the goToMaximumWarp() method called enterWarp, then call that later on whenever we want to. The problem is, you can't do the same thing for properties – you can't say "create a reference to the captain's name property that I can check when the inevitable mutiny happens," because Swift will just read the property directly and you'll just get its original value.

This is fixed with keypaths: they are uninvoked references to properties just like our enterWarp() code. If you invoke the reference now you get the current value, but if you invoke the reference later you get the latest value. You can dig through any number of properties, and Swift uses its type inference to ensure you get the correct type back.

The Swift Evolution community spent quite a while discussing the correct syntax for keypaths because it needed to be something visually different from other Swift code, and the syntax they ended up with uses backslashes: \Starship.name, \Starship.maxWarp, and \Starship.captain.name. You can assign those two to a variable then use them whenever you want, on any Starship instance. For example:

let nameKeyPath = \Starship.name

let maxWarpKeyPath = \Starship.maxWarp

let captainName = \Starship.captain.name

let starshipName = voyager[keyPath: nameKeyPath]

let starshipMaxWarp = voyager[keyPath: maxWarpKeyPath]

let starshipCaptain = voyager[keyPath: captainName]

That will make starshipName a string and starshipMaxWarp a double, because Swift is able to infer the types correctly. The third example there even goes into the property of a property, and Swift still figures it out correctly.

Future plans for this include being able to access array indexes and to create keypaths from strings at runtime – for more information see [the Swift Evolution proposal for this new feature](https://github.com/apple/swift-evolution/blob/master/proposals/0161-key-paths.md).

## **Intermission**

If you're enjoying this article, you might like my free Natural Swift video. It gives you 75 minutes of hands-on coding that teaches functional programming, protocol-oriented programming, and value types, and **you can download it for free with no obligation or catches** – [just click here](https://gumroad.com/l/natural-swift).

And now back to your regularly scheduled broadcast…

## **Improved dictionary functionality**

One of the most intriguing proposals for Swift 4 was to add some new functionality to dictionaries to make them more powerful, and also to make them behave more like you would expect in certain situations.

Let's start with a simple example: filtering dictionaries in Swift 3 does not return a new dictionary. Instead, it returns an array of tuples with key/value labels. For example:

let cities = ["Shanghai": 24\_256\_800, "Karachi": 23\_500\_000, "Beijing": 21\_516\_000, "Seoul": 9\_995\_000];

let massiveCities = cities.filter { $0.value > 10\_000\_000 }

After that code runs you can't read massiveCities["Shanghai"] because it is no longer a dictionary. Instead, you need to use massiveCities[0].value, which isn't great.

As of Swift 4 this behaves more like you would expect: you get back a new dictionary. Obviously this will break any existing code that relies on the tuple-array return type.

Similarly, the map() method on dictionaries never quite worked the way many people hoped: you got a key-value tuple passed in, and could return a single value to be added to an array. For example:

let populations = cities.map { $0.value \* 2 }

That hasn't changed in Swift 4, but there is a new method called mapValues() that is going to be much more useful because it lets you transform the values and place them back into a dictionary using the original keys.

For example, this code will round and stringify all city populations, then put them back into a new dictionary with the same keys of Shanghai, Karachi, and Seoul:

let roundedCities = cities.mapValues { "\($0 / 1\_000\_000) million people" }

(In case you were wondering, it's not safe to map dictionary keys because you might create duplicates by accident.)

Easily my favorite new dictionary addition is a grouping initializer, which converts a sequence into a dictionary of sequences that are grouped by whatever you want. Continuing our cities example, we could use cities.keys to get back an array of city names, then group them by their first letter, like this:

let groupedCities = Dictionary(grouping: cities.keys) { $0.characters.first! }

print(groupedCities)

That will output the following:

["B": ["Beijing"], "S": ["Shanghai", "Seoul"], "K": ["Karachi"]]

Alternatively, we could group the cities based on the length of their names like this:

let groupedCities = Dictionary(grouping: cities.keys) { $0.count }

print(groupedCities)

That will output the following:

[5: ["Seoul"], 7: ["Karachi", "Beijing"], 8: ["Shanghai"]]

Finally, it's now possible to access a dictionary key and provide a default value to use if the key is missing:

let person = ["name": "Taylor", "city": "Nashville"]

let name = person["name", default: "Anonymous"]

Now, any experienced developer will probably argue that's better written using nil coalescing, and I agree. You could write this line instead using the current version of Swift:

let name = person["name"] ?? "Anonymous"

However, that doesn't work when you're modifying the dictionary value rather than just reading it. You can't modify a dictionary value in place because accessing its key returns an optional – the key might not exist, after all. With Swift 4's default dictionary values you can write much more succinct code, such as this:

var favoriteTVShows = ["Red Dwarf", "Blackadder", "Fawlty Towers", "Red Dwarf"]

var favoriteCounts = [String: Int]()

for show in favoriteTVShows {

 favoriteCounts[show, default: 0] += 1

}

That loops over every string in favoriteTVShows, and uses a dictionary called favoriteCountsto keep track of how often each item appears. We can modify the dictionary in one line of code because we know it will always have a value: either the default value of 0, or some higher number based on previous counting.

For more information see [the Swift Evolution proposal for these new features](https://github.com/apple/swift-evolution/blob/master/proposals/0165-dict.md).

## **Strings are collections again!**

This is a small change, but one guaranteed to make a lot of people happy: strings are collections again. This means you can reverse them, loop over them character-by-character, map() and flatMap() them, and more. For example:

let quote = "It is a truth universally acknowledged that new Swift versions bring new features."

let reversed = quote.reversed()

for letter in quote {

 print(letter)

}

This change was introduced as part of a broad set of amendments called the [String Manifesto](https://github.com/apple/swift/blob/master/docs/StringManifesto.md).

## **One-sided ranges**

Last but not least, Swift 4 introduces Python-like one-sided collection slicing, where the missing side is automatically inferred to be the start or end of the collection. This has no effect on existing code because it's a new use for the existing operator, so you don't need to worry about potential breakage.

Here's an example:

let characters = ["Dr Horrible", "Captain Hammer", "Penny", "Bad Horse", "Moist"]

let bigParts = characters[..<3]

let smallParts = characters[3...]

print(bigParts)

print(smallParts)

That code will print out ["Dr Horrible", "Captain Hammer", "Penny"] then ["Bad Horse", "Moist"].

For more information see [the Swift Evolution proposal for this new feature](https://github.com/apple/swift-evolution/blob/master/proposals/0172-one-sided-ranges.md).

## **There's more still to come…**

The first release of Xcode that ships with Swift 4 is likely to arrive in June, presumably along with iOS 11, tvOS 11, watchOS 4, and macOS Somewhere Else In California. What we've seen so far is already promising, particularly because it's clear the team is working hard to make Swift 4 as additive as possible. Primarily adding new features rather than breaking or modifying existing ones should make it easier to upgrade to, and hopefully signals the start of a new stability for the language.

Although the Swift Evolution can be chaotic sometimes (access levels, anyone?), Swift 4 validates Apple's community approach once again. I've linked to several Swift Evolution proposals above, each of which were discussed extensively by the community to help reach consensus – this isn't just Apple engineers forcing through changes because they can, but instead is a sensible, considered approach to refining what is already a smart and elegant language.

One feature that was postponed was [ABI compatibility](https://github.com/apple/swift/blob/master/docs/ABIStabilityManifesto.md), which would allow developers to distribute compiled libraries – one of the few key missing features that remain in Swift today. Hopefully we'll get it before Swift 5…

Swift 4.1 is the first minor release of Swift 4, bringing with it some useful improvements such as automatically synthesized equatable and hashable, conditional conformances, a smarter way to detect the simulator environment, and more.

Make sure you have [Xcode 9.3 or later](https://developer.apple.com/download/%22%20%5Ct%20%22_blank) then create a new playground. Let’s take a look at the top new features for this release…

* **Update:** I created an [Xcode Playground showing what's new in Swift 4.1 with examples you can edit](https://www.hackingwithswift.com/articles/70/learn-whats-new-in-swift-4-1-with-a-playground).

Synthesized Equatable and Hashable

The Equatable protocol allows Swift to compare one instance of a type against another. When we say 5 == 5, Swift understands what that means because Int conforms to Equatable, which means it implements a function describing what == means for two instances of Int.

Implementing Equatable in our own value types allows them to work like Swift’s strings, arrays, numbers, and more, and it’s usually a good idea to make your structs conform to Equatable just so they fit the concept of value types better.

However, implementing Equatable by hand can be annoying. Consider this code:

struct Person {

 var firstName: String

 var lastName: String

 var age: Int

 var city: String

}

If you have two instances of Person and want to make sure they are identical, you need to compare all four of its properties, like this:

struct Person: Equatable {

 var firstName: String

 var lastName: String

 var age: Int

 var city: String

 static func ==(lhs: Person, rhs: Person) -> Bool {

 return lhs.firstName == rhs.firstName && lhs.lastName == rhs.lastName && lhs.age == rhs.age && lhs.city == rhs.city

 }

}

Even *reading* that is tiring, never mind *writing* it.

Fortunately, Swift 4.1 can synthesize conformance for Equatable for us – it can generate an == method automatically, which will compare all properties in one value with all properties in another, just like above. So, all you have to do now is add Equatable as a protocol for your type, and Swift will do the rest:

struct Person: Equatable {

 var firstName: String

 var lastName: String

 var age: Int

 var city: String

}

Of course, if you *want* you can implement == yourself. For example, if your type has an id field that identifies it uniquely, you would write == to compare that single value rather than letting Swift do all the extra work.

Swift 4.1 also introduces synthesized support for the Hashable protocol, which means it will generate a hashValue property for conforming types automatically. Hashable was always annoying to implement because you need to return a unique (or at least mostly unique) hash for every object. It’s important, though, because it lets you use your objects as dictionary keys and store them in sets.

Previously we’d need to write code like this:

var hashValue: Int {

 return firstName.hashValue ^ lastName.hashValue &\* 16777619

}

For the most part that’s no longer needed in Swift 4.1, although as with Equatable you might still want to write your own method if there’s something specific you need.

**Note:** You still need to opt in to these protocols by adding a conformance to your type, and using the synthesized code does require that all properties in your type conform to Equatable or Hashable respectively.

For more information, see [Swift Evolution proposal SE-0185](https://github.com/apple/swift-evolution/blob/master/proposals/0185-synthesize-equatable-hashable.md).

Key decoding strategy for Codable

* I wrote a full article on this useful new feature: [Swift 4.1 improves Codable with keyDecodingStrategy](https://www.hackingwithswift.com/articles/52/swift-4-1-improves-codable-with-keydecodingstrategy)

In Swift 4.0 a common problem was trying to use Codable with JSON that utilized snake\_case for its key names rather than the camelCase we normally use in Swift. Codable was unable to understand how the two different name types were mapped, so you had to create a custom CodingKeys enum helping it out.

This is where Swift 4.1's new keyDecodingStrategy property comes in: it’s set to .useDefaultKeys by default, which does a direct mapping of JSON names to Swift properties. However, if you change it to .convertFromSnakeCase then Codable handles the name conversion for us.

For example:

let decoder = JSONDecoder()

decoder.keyDecodingStrategy = .convertFromSnakeCase

do {

 let macs = try decoder.decode([Mac].self, from: jsonData)

 print(macs)

} catch {

 print(error.localizedDescription)

}

When you want to go back the other way – to convert a Codable struct with camelCase properties back to JSON with snake\_case keys, set the keyEncodingStrategy to .convertToSnakeCase like this:

let encoder = JSONEncoder()

encoder.keyEncodingStrategy = .convertToSnakeCase

let encoded = try encoder.encode(someObject)

Conditional conformances

Swift 4.1 implements [SE-0143](https://github.com/apple/swift-evolution/blob/master/proposals/0143-conditional-conformances.md), which introduced proposed conditional conformances into the language. This allows types to conform to a protocol only when certain conditions are met.

To demonstrate conditional conformances, let's create a Purchaseable protocol that we can use to buy things:

protocol Purchaseable {

 func buy()

}

We can now define a Book struct that conforms to the protocol, and prints a message when a book is bought:

struct Book: Purchaseable {

 func buy() {

 print("You bought a book")

 }

}

So far this is easy enough, but let's take it one step further: what if the user has a basket full of books, and wants to buy them all? We could loop over all books in the array by hand, calling buy() on each one. But a better approach is to write an extension on Arrayto make it conform to Purchaseable, then give it a buy() method that in turn calls buy() on each of its elements.

This is where conditional conformances come in: if we tried to extend all arrays, we'd be adding functionality where it wouldn't make sense – we'd be adding buy() to arrays of strings, for example, even though those strings don't have a buy() method we can call.

Swift 4.1 lets us make arrays conform to Purchaseable only if their elements also conform to Purchaseable, like this:

extension Array: Purchaseable where Element: Purchaseable {

 func buy() {

 for item in self {

 item.buy()

 }

 }

}

As you can see, conditional conformances let us constrain the way our extensions are applied more precisely than was possible before.

Conditional conformances also make large parts of Swift code easier and safer, even if you don't do any extra work yourself. For example, this code creates two arrays of optional strings and checks whether they are equal:

var left: [String?] = ["Andrew", "Lizzie", "Sophie"]

var right: [String?] = ["Charlotte", "Paul", "John"]

left == right

That might seem trivial, but that code wouldn't even compile in Swift 4.0 – both Stringand [String] were equatable, but [String?] was not.

The introduction of conditional conformance in Swift 4.1 means that it’s now possible to add protocol conformance to a type as long as it satisfies a condition. In this case, if the elements of the array are equatable, that means the whole thing is equatable. So, the above code now compiles in Swift 4.1

Conditional conformance has been extended to the Codable protocol in a way that will definitely make things safer. For example:

import Foundation

struct Person {

 var name = "Taylor"

}

var people = [Person()]

var encoder = JSONEncoder()

// try encoder.encode(people)

If you uncomment the encoder.encode(people) line, Swift will refuse to build your code because you're trying to encode a struct that doesn't conform to Codable. However, that code compiled cleanly with Swift 4.0, then threw a fatal error at runtime because Persondoesn’t conform to Codable.

Obviously no one wants a fatal error at runtime, because it means your app crashes. Fortunately, Swift 4.1 cleans this up using conditional conformances: Optional, Array, Dictionary, and Set now only conform to Codable if their contents also conform to Codable, so the above code will refuse to compile.

Recursive constraints on associated types

Swift 4.1 implements [SE-0157](https://github.com/apple/swift-evolution/blob/master/proposals/0157-recursive-protocol-constraints.md), which lifts restrictions on the way we use associated types inside protocols. As a result, we can now create recursive constraints for our associated types: associated types that are constrained by the protocol they are defined in.

To demonstrate this, let's consider a simple team hierarchy in a tech company. In this company, every employee has a manager – someone more senior to them that they report to. Each manager must also be an employee of the company, because it would be weird if they weren't.

We can express this relationship in a simple Employee protocol:

protocol Employee {

 associatedtype Manager: Employee

 var manager: Manager? { get set }

}

**Note:** I've used an optional Manager? because ultimately one person (presumably the CEO) has no manager.

Even though that's a fairly self-evident relationship, it wasn't possible to compile that code in Swift 4.0 because we're using the Employee protocol inside itself. However, this is fixed in Swift 4.1 because of the new ability to use recursive constraints on associated types.

Thanks to this new feature, we can model a simple tech company that has three kinds of team members: junior developers, senior developers, and board members. The reporting structure is also simple: junior developers are managed by senior developers, senior developers are managed by board members, and board members may be managed by another board member – e.g. the CTO reporting to the CEO.

That looks exactly as you would imagine thanks to Swift 4.1:

class BoardMember: Employee {

 var manager: BoardMember?

}

class SeniorDeveloper: Employee {

 var manager: BoardMember?

}

class JuniorDeveloper: Employee {

 var manager: SeniorDeveloper?

}

**Note:** I've used classes here rather than structs because BoardMember itself contains a BoardMember property and that would result in an infinitely sized struct. If one of these has to be a class I personally would prefer to make all three classes just for consistency, but if you preferred you could leave BoardMember as a class and make both SeniorDeveloper and JuniorDeveloper into structs.

Build configuration import testing

Swift 4.1 implements [SE-0075](https://github.com/apple/swift-evolution/blob/master/proposals/0075-import-test.md), which introduces a new canImport condition that lets us check whether a specific module can be imported when our code is compiled.

This is particularly important for cross-platform code: if you had a Swift file that implemented one behavior on macOS and another on iOS, or if you needed specific functionality for Linux. For example:

#if canImport(SpriteKit)

 // this will be true for iOS, macOS, tvOS, and watchOS

#else

 // this will be true for other platforms, such as Linux

#endif

Previously you would have had to use inclusive or exclusive tests by operating system, like this:

#if !os(Linux)

 // Matches macOS, iOS, watchOS, tvOS, and any other future platforms

#endif

#if os(macOS) || os(iOS) || os(tvOS) || os(watchOS)

 // Matches only Apple platforms, but needs to be kept up to date as new platforms are added

#endif

The new canImport condition lets us focus on the functionality we care about rather than what platform we're compiling for, thus avoiding a variety of problems.

Target environment testing

Swift 4.1 implements [SE-0190](https://github.com/apple/swift-evolution/blob/master/proposals/0190-target-environment-platform-condition.md), which introduces a new targetEnvironment condition that lets us differentiate between builds that are for physical devices and those that are for a simulated environment.

At this time targetEnvironment has only one value, simulator, which will be true if your build is targeting a simulated device such as the iOS Simulator. For example:

#if targetEnvironment(simulator)

 // code for the simulator here

#else

 // code for real devices here

#endif

This is useful when writing code to deal with functionality the simulator doesn't support, such as capturing photos from a camera or reading the accelerometer.

As an example, let's look at processing a photo from the camera. If we're running on a real device we'll create and configure a UIImagePickerController() to take photos using the camera, but if we're in the simulator we'll just load a sample image from our app bundle:

import UIKit

class TestViewController: UIViewController, UIImagePickerControllerDelegate {

 // a method that does some sort of image processing

 func processPhoto(\_ img: UIImage) {

 // process photo here

 }

 // a method that loads a photo either using the camera or using a sample

 func takePhoto() {

 #if targetEnvironment(simulator)

 // we're building for the simulator; use the sample photo

 if let img = UIImage(named: "sample") {

 processPhoto(img)

 } else {

 fatalError("Sample image failed to load")

 }

 #else

 // we're building for a real device; take an actual photo

 let picker = UIImagePickerController()

 picker.sourceType = .camera

 vc.allowsEditing = true

 picker.delegate = self

 present(picker, animated: true)

 #endif

 }

 // this is called if the photo was taken successfully

 func imagePickerController(\_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String : Any]) {

 // hide the camera

 picker.dismiss(animated: true)

 // attempt to retrieve the photo they took

 guard let image = info[UIImagePickerControllerEditedImage] as? UIImage else {

 // that failed; bail out

 return

 }

 // we have an image, so we can process it

 processPhoto(image)

 }

}

flatMap is now (partly) compactMap

The flatMap() method was useful for a variety of things in Swift 4.0, but one was particularly useful: the ability to transform each object in a collection, then remove any items that were nil.

[Swift Evolution proposal SE-0187](https://github.com/apple/swift-evolution/blob/master/proposals/0187-introduce-filtermap.md) suggested changing this, and as of Swift 4.1 this flatMap() variant has been renamed to compactMap() to make its meaning clearer.

For example:

let array = ["1", "2", "Fish"]

let numbers = array.compactMap { Int($0) }

That will create an Int array containing the numbers 1 and 2, because "Fish" will fail conversion to Int, return nil, and be ignored.

Looking forward to Swift 5.0

The introduction of conditional conformance has enabled the Swift team to take out a fair amount of code while also promoting stability, and automatic Equatable and Hashablesupport will definitely make our lives easier.

Swift 4.2 introduces a variety of further enhancements, including derived collections of enum cases, warning and error diagnostic directives, and dynamic member look up – [click here to learn what's new in Swift 4.2](https://www.hackingwithswift.com/articles/77/whats-new-in-swift-4-2).

There are still other big proposals on the way, including [SE-0192: Non-Exhaustive Enums](https://github.com/apple/swift-evolution/blob/master/proposals/0192-non-exhaustive-enums.md). But as important as those features are, this is the year Apple will, we hope, deliver ABI stability for Swift, and that’s going to be *huge*. Fingers crossed!

Swift 4.2 is the second minor release of Swift 4, and brings with it another raft of awesome improvements – this is turning out to be an incredible year for Swift, and yet more validation that the community-driven Swift Evolution process is helping make a great language even better.

This time we’re getting features such as enum case arrays, warning and error compiler directives, dynamic member look up, and more, all of which come on top of the new features introduced in Swift 4.1 – check out my article [what's new in Swift 4.1](https://www.hackingwithswift.com/articles/50/whats-new-in-swift-4-1) if you missed those changes. You might also want to look ahead to see [what's new in Swift 5.0](https://www.hackingwithswift.com/articles/126/whats-new-in-swift-5-0), because that introduces raw strings, count(where:), isMultiple(of:), and more.

* **Try it yourself:** I created an [Xcode Playground showing what's new in Swift 4.2 with examples you can edit](https://github.com/twostraws/whats-new-in-swift-4-2%22%20%5Ct%20%22_blank).

Derived collections of enum cases

[SE-0194](https://github.com/apple/swift-evolution/blob/master/proposals/0194-derived-collection-of-enum-cases.md) introduces a new CaseIterable protocol that automatically generates an array property of all cases in an enum.

Prior to Swift 4.2 this either took hacks, hand-coding, or Sourcery code generation to accomplish, but now all you need to do is make your enum conform to the CaseIterableprotocol. At compile time, Swift will automatically generate an allCases property that is an array of all your enum’s cases, in the order you defined them.

For example, this creates an enum of pasta shapes and asks Swift to automatically generate an allCases array for it:

enum Pasta: CaseIterable {

 case cannelloni, fusilli, linguine, tagliatelle

}

You can then go ahead and use that property as a regular array – it will be a [Pasta] given the code above, so we could print it like this:

for shape in Pasta.allCases {

 print("I like eating \(shape).")

}

This automatic synthesis of allCases will only take place for enums that do not use associated values. Adding those automatically wouldn’t make sense, however if you want you can add it yourself:

enum Car: CaseIterable {

 static var allCases: [Car] {

 return [.ford, .toyota, .jaguar, .bmw, .porsche(convertible: false), .porsche(convertible: true)]

 }

 case ford, toyota, jaguar, bmw

 case porsche(convertible: Bool)

}

At this time, Swift is unable to synthesize the allCases property if any of your enum cases are marked unavailable. So, if you need allCases then you’ll need to add it yourself, like this:

enum Direction: CaseIterable {

 static var allCases: [Direction] {

 return [.north, .south, .east, .west]

 }

 case north, south, east, west

 @available(\*, unavailable)

 case all

}

**Important:** You need to add CaseIterable to the original declaration of your enum rather than an extension in order for the allCases array to be synthesized. This means you can’t use extensions to retroactively make existing enums conform to the protocol.

Warning and error diagnostic directives

[SE-0196](https://github.com/apple/swift-evolution/blob/master/proposals/0196-diagnostic-directives.md) introduces new compiler directives that help us mark issues in our code. These will be familiar to any developers who had used Objective-C previously, but as of Swift 4.2 we can enjoy them in Swift too.

The two new directives are #warning and #error: the former will force Xcode to issue a warning when building your code, and the latter will issue a compile error so your code won’t build at all. Both of these are useful for different reasons:

* #warning is mainly useful as a reminder to yourself or others that some work is incomplete. Xcode templates often use #warning to mark method stubs that you should replace with your own code.
* #error is mainly useful if you ship a library that requires other developers to provide some data. For example, an authentication key for a web API – you want users to include their own key, so using #error will force them to change that code before continuing.

Both of these work in the same way: #warning("Some message") and #error("Some message"). For example:

func encrypt(\_ string: String, with password: String) -> String {

 #warning("This is terrible method of encryption")

 return password + String(string.reversed()) + password

}

struct Configuration {

 var apiKey: String {

 #error("Please enter your API key below then delete this line.")

 return "Enter your key here"

 }

}

Both #warning and #error work alongside the existing #if compiler directive, and will only be triggered if the condition being evaluated is true. For example:

#if os(macOS)

#error("MyLibrary is not supported on macOS.")

#endif

Dynamic member look up

[SE-0195](https://github.com/apple/swift-evolution/blob/master/proposals/0195-dynamic-member-lookup.md) introduces a way to bring Swift closer to scripting languages such as Python, but in a type-safe way – you don’t lose any of Swift’s safety, but you do gain the ability to write the kind of code you’re more likely to see in PHP and Python.

At the core of this feature is a new attribute called @dynamicMemberLookup, which instructs Swift to call a subscript method when accessing properties. This subscript method, subscript(dynamicMember:), is *required*: you’ll get passed the string name of the property that was requested, and can return any value you like.

Let’s look at a trivial example so you can understand the basics. We could create a Personstruct that reads its values from a dictionary like this:

@dynamicMemberLookup

struct Person {

 subscript(dynamicMember member: String) -> String {

 let properties = ["name": "Taylor Swift", "city": "Nashville"]

 return properties[member, default: ""]

 }

}

The @dynamicMemberLookup attribute requires the type to implement a subscript(dynamicMember:) method to handle the actual work of dynamic member lookup. As you can see, I’ve written one that accepts the member name as string and returns a string, and internally it just looks up the member name in a dictionary and returns its value.

That struct allows us to write code like this:

let person = Person()

print(person.name)

print(person.city)

print(person.favoriteIceCream)

That will compile cleanly and run, even though name, city, and favoriteIceCream do not exist as properties on the Person type. Instead, they are all looked up at runtime: that code will print “Taylor Swift” and “Nashville” for the first two calls to print(), then an empty string for the final one because our dictionary doesn’t store anything for favoriteIceCream.

My subscript(dynamicMember:) method *must* return a string, which is where Swift’s type safety comes in: even though you’re dealing with dynamic data, Swift will still ensure you get back what you expected. And if you want multiple different types, just implement different subscript(dynamicMember:) methods, like this:

@dynamicMemberLookup

struct Employee {

 subscript(dynamicMember member: String) -> String {

 let properties = ["name": "Taylor Swift", "city": "Nashville"]

 return properties[member, default: ""]

 }

 subscript(dynamicMember member: String) -> Int {

 let properties = ["age": 26, "height": 178]

 return properties[member, default: 0]

 }

}

Now that any property can be accessed in more than one way, Swift requires you to be clear which one should be run. That might be implicit, for example if you send the return value into a function that accepts only strings, or it might be explicit, like this:

let employee = Employee()

let age: Int = employee.age

Either way, Swift must know for sure which subscript will be called.

You can even overload subscript to return closures:

@dynamicMemberLookup

struct User {

 subscript(dynamicMember member: String) -> (\_ input: String) -> Void {

 return {

 print("Hello! I live at the address \($0).")

 }

 }

}

let user = User()

user.printAddress("555 Taylor Swift Avenue")

When that’s run, user.printAddress returns a closure that prints out a string, and the ("555 Taylor Swift Avenue") part immediately calls it with that input.

If you use dynamic member subscripting in a type that has also some regular properties and methods, those properties and methods will always be used in place of the dynamic member. For example, we could define a Singer struct with a built-in name property alongside a dynamic member subscript:

struct Singer {

 public var name = "Justin Bieber"

 subscript(dynamicMember member: String) -> String {

 return "Taylor Swift"

 }

}

let singer = Singer()

print(singer.name)

That code will print “Justin Bieber”, because the name property will be used rather than the dynamic member subscript.

@dynamicMemberLookup plays a full part in Swift’s type system, which means you can assign them to protocols, structs, enums, and classes – even classes that are marked @objc.

In practice, this means two things. First, you can create a class using @dynamicMemberLookup, and any classes that inherit from it are also automatically @dynamicMemberLookup. So, this will print “I’m a sandwich” because HotDog inherits from Sandwich:

@dynamicMemberLookup

class Sandwich {

 subscript(dynamicMember member: String) -> String {

 return "I'm a sandwich!"

 }

}

class HotDog: Sandwich { }

let chiliDog = HotDog()

print(chiliDog.description)

**Note:** If you don’t think hot dogs are sandwiches, why not [follow me on Twitter](https://twitter.com/twostraws) so you can tell me how wrong I am?

Second, you can retroactively make other types use @dynamicMemberLookup by defining it on a protocol, adding a default implementation of subscript(dynamicMember:) using a protocol extension, then making other types conform to your protocol however you want.

For example, this creates a new Subscripting protocol, provides a default subscript(dynamicMember:) implementation that returns a message, then extends Swift’s String to use that protocol:

@dynamicMemberLookup

protocol Subscripting { }

extension Subscripting {

 subscript(dynamicMember member: String) -> String {

 return "This is coming from the subscript"

 }

}

extension String: Subscripting { }

let str = "Hello, Swift"

print(str.username)

In his Swift Evolution proposal, Chris Lattner also gives an example JSON enum that uses dynamic member lookup to create more natural syntax for navigating through JSON:

@dynamicMemberLookup

enum JSON {

 case intValue(Int)

 case stringValue(String)

 case arrayValue(Array<JSON>)

 case dictionaryValue(Dictionary<String, JSON>)

 var stringValue: String? {

 if case .stringValue(let str) = self {

 return str

 }

 return nil

 }

 subscript(index: Int) -> JSON? {

 if case .arrayValue(let arr) = self {

 return index < arr.count ? arr[index] : nil

 }

 return nil

 }

 subscript(key: String) -> JSON? {

 if case .dictionaryValue(let dict) = self {

 return dict[key]

 }

 return nil

 }

 subscript(dynamicMember member: String) -> JSON? {

 if case .dictionaryValue(let dict) = self {

 return dict[member]

 }

 return nil

 }

}

Without dynamic member look up you would need to navigate an instance of that JSONenum like this:

let json = JSON.stringValue("Example")

json[0]?["name"]?["first"]?.stringValue

But *with* dynamic member look up you can use this instead:

json[0]?.name?.first?.stringValue

I think this example is particularly important because it gets to the nub of what @dynamicMemberLookup does: it’s syntactic sugar that turns a custom subscript into simple dot syntax.

**Note:** Using dynamic member lookup means that code completion loses much if not all of its usefulness, because there’s nothing to complete. This isn’t too much of a surprise, though, and it’s something that Python IDEs have had to deal with for some time. Chris Lattner (the author of SE-0195) discussed future possibilities for code completion in the proposal itself – it’s [worth reading](https://github.com/apple/swift-evolution/blob/master/proposals/0195-dynamic-member-lookup.md#future-directions-python-code-completion).

 Enhanced conditional conformances

Conditional conformances [were introduced in Swift 4.1](https://www.hackingwithswift.com/articles/50/whats-new-in-swift-4-1), allowing types to conform to a protocol only when certain conditions are met.

For example, if we had a Purchaseable protocol:

protocol Purchaseable {

 func buy()

}

And a simple type that conforms to that protocol:

struct Book: Purchaseable {

 func buy() {

 print("You bought a book")

 }

}

Then we could make Array conform to Purchaseable if all the elements inside the array were also Purchasable:

extension Array: Purchaseable where Element: Purchaseable {

 func buy() {

 for item in self {

 item.buy()

 }

 }

}

This worked great at compile time, but there was a problem: if you needed to query a conditional conformance at runtime, your code would crash because it wasn’t supported in Swift 4.1

Well, in Swift 4.2 that’s now fixed, so if you receive data of one type and want to check if it can be converted to a conditionally conformed protocol, it works great.

For example:

let items: Any = [Book(), Book(), Book()]

if let books = items as? Purchaseable {

 books.buy()

}

In addition, support for automatic synthesis of Hashable conformance has improved greatly in Swift 4.2. Several built-in types from the Swift standard library – including optionals, arrays, dictionaries, and ranges – now automatically conform to the Hashableprotocol when their elements conform to Hashable.

For example:

struct User: Hashable {

 var name: String

 var pets: [String]

}

Swift 4.2 can automatically synthesize Hashable conformance for that struct, but Swift 4.1 could not.

Random number generation and shuffling

[SE-0202](https://github.com/apple/swift-evolution/blob/master/proposals/0202-random-unification.md) introduces a new random API that’s native to Swift. This means you can for the most part stop using arc4random\_uniform() and GameplayKit to get randomness, and instead rely on a cryptographically secure randomizer that’s baked right into the core of the language.

You can generate random numbers by calling the random() method on whatever numeric type you want, providing the range you want to work with. For example, this generates a random number in the range 1 through 4, inclusive on both sides:

let randomInt = Int.random(in: 1..<5)

Similar methods exist for Float, Double, and CGFloat:

let randomFloat = Float.random(in: 1..<10)

let randomDouble = Double.random(in: 1...100)

let randomCGFloat = CGFloat.random(in: 1...1000)

There’s also one for booleans, generating either true or false randomly:

let randomBool = Bool.random()

Checking a random boolean is effectively the same as checking Int.random(in: 0...1) == 1, but it expresses your intent more clearly.

SE-0202 also includes support for shuffling arrays using new shuffle() and shuffled()methods depending on whether you want in-place shuffling or not. For example:

var albums = ["Red", "1989", "Reputation"]

// shuffle in place

albums.shuffle()

// get a shuffled array back

let shuffled = albums.shuffled()

It also adds a new randomElement() method to arrays, which returns one random element from the array if it isn’t empty, or nil otherwise:

if let random = albums.randomElement() {

 print("The random album is \(random).")

}

Simpler, more secure hashing

Swift 4.2 implements [SE-0206](https://github.com/apple/swift-evolution/blob/master/proposals/0206-hashable-enhancements.md), which simplifies the way we make custom types conform to the Hashable protocol.

From Swift 4.1 onwards conformance to Hashable can be synthesized by the compiler. However, if you want your own hashing implementation – for example, if your type has many properties but you know that one of them was enough to identify it uniquely – you still need to write your own code using whatever algorithm you thought was best.

Swift 4.2 introduces a new Hasher struct that provides a randomly seeded, universal hash function to make this process easier:

struct iPad: Hashable {

 var serialNumber: String

 var capacity: Int

 func hash(into hasher: inout Hasher) {

 hasher.combine(serialNumber)

 }

}

You can add more properties to your hash by calling combine() repeatedly, and the order in which you add properties affects the finished hash value.

You can also use Hasher as a standalone hash generator: just provide it with whatever values you want to hash, then call finalize() to generate the final value. For example:

let first = iPad(serialNumber: "12345", capacity: 256)

let second = iPad(serialNumber: "54321", capacity: 512)

var hasher = Hasher()

hasher.combine(first)

hasher.combine(second)

let hash = hasher.finalize()

Hasher uses a random seed every time it hashes an object, which means the hash value for any object is effectively guaranteed to be different between runs of your app.

This in turn means that elements you add to a set or a dictionary are highly likely to have a different order each time you run your app.

Checking sequence elements match a condition

[SE-0207](https://github.com/apple/swift-evolution/blob/master/proposals/0207-containsOnly.md) provides a new allSatisfy() method that checks whether all items in a sequence pass a condition.

For example, if we had an array of exam results like this:

let scores = [85, 88, 95, 92]

We could decide whether a student passed their course by checking whether all their exam results were 85 or higher:

let passed = scores.allSatisfy { $0 >= 85 }

In-place collection element removal

[SE-0197](https://github.com/apple/swift-evolution/blob/master/proposals/0197-remove-where.md) introduces a new removeAll(where:) method that performs a high-performance, in-place filter for collections. You give it a closure condition to run, and it will strip out all objects that match the condition.

For example, if you have a collection of names and want to remove people called “Terry”, you’d use this:

var pythons = ["John", "Michael", "Graham", "Terry", "Eric", "Terry"]

pythons.removeAll { $0.hasPrefix("Terry") }

print(pythons)

Now, you might very well think that you could accomplish that by using filter() like this:

pythons = pythons.filter { !$0.hasPrefix("Terry") }

However, that doesn’t use memory very efficiently, it specifies what you *don’t* want rather than what you *want*, and more advanced in-place solutions come with a range of complexities that are off-putting to novices. Ben Cohen, the author of SE-0197, gave a talk at [dotSwift 2018](https://www.dotconferences.com/2018/01/ben-cohen-extending-the-standard-library%22%20%5Ct%20%22_blank) where he discussed the implementation of this proposal in more detail – if you’re keen to learn why it’s so efficient, you should start there!

Boolean toggling

[SE-0199](https://github.com/apple/swift-evolution/blob/master/proposals/0199-bool-toggle.md) introduces a new toggle() method to booleans that flip them between true and false. This caused a lot of discussion in the Swift community, partly because some thought it too trivial for inclusion, but partly also because the Swift Forums discussion [veered out of control at times](http://ericasadun.com/2018/03/09/swift-evolution-and-civility/).

The entire code to implement proposal is only a handful of lines of Swift:

extension Bool {

 mutating func toggle() {

 self = !self

 }

}

However, the end result makes for much more natural Swift code:

var loggedIn = false

loggedIn.toggle()

As noted in the proposal, this is particularly useful in more complex data structures: myVar.prop1.prop2.enabled.toggle() avoids the potential typing errors that could be caused using manual negation.

The proposal makes Swift easier and safer to write, and is purely additive, so I think most folks will switch to using it quickly enough.

A standard Result type

* [**Watch the video**](https://www.youtube.com/watch?v=RBZFCp3kSLM)

[SE-0235](https://github.com/apple/swift-evolution/blob/master/proposals/0235-add-result.md) introduces a Result type into the standard library, giving us a simpler, clearer way of handling errors in complex code such as asynchronous APIs.

Swift’s Result type is implemented as an enum that has two cases: success and failure. Both are implemented using generics so they can have an associated value of your choosing, but failure must be something that conforms to Swift’s Error type.

To demonstrate Result, we could write a function that connects to a server to figure out how many unread messages are waiting for the user. In this example code we’re going to have just one possible error, which is that the requested URL string isn’t a valid URL:

enum NetworkError: Error {

 case badURL

}

The fetching function will accept a URL string as its first parameter, and a completion handler as its second parameter. That completion handler will itself accept a Result, where the success case will store an integer, and the failure case will be some sort of NetworkError. We’re not actually going to connect to a server here, but using a completion handler at least lets us simulate asynchronous code.

Here’s the code:

import Foundation

func fetchUnreadCount1(from urlString: String, completionHandler: @escaping (Result<Int, NetworkError>) -> Void) {

 guard let url = URL(string: urlString) else {

 completionHandler(.failure(.badURL))

 return

 }

 // complicated networking code here

 print("Fetching \(url.absoluteString)...")

 completionHandler(.success(5))

}

To use that code we need to check the value inside our Result to see whether our call succeeded or failed, like this:

fetchUnreadCount1(from: "https://www.hackingwithswift.com") { result in

 switch result {

 case .success(let count):

 print("\(count) unread messages.")

 case .failure(let error):

 print(error.localizedDescription)

 }

}

There are three more things you ought to know before you start using Result in your own code.

First, Result has a get() method that either returns the successful value if it exists, or throws its error otherwise. This allows you to convert Result into a regular throwing call, like this:

fetchUnreadCount1(from: "https://www.hackingwithswift.com") { result in

 if let count = try? result.get() {

 print("\(count) unread messages.")

 }

}

Second, Result has an initializer that accepts a throwing closure: if the closure returns a value successfully that gets used for the success case, otherwise the thrown error is placed into the failure case.

For example:

let result = Result { try String(contentsOfFile: someFile) }

Third, rather than using a specific error enum that you’ve created, you can also use the general Error protocol. In fact, the Swift Evolution proposal says “it's expected that most uses of Result will use Swift.Error as the Error type argument.”

So, rather than using Result<Int, NetworkError> you could use Result<Int, Error>. Although this means you lose the safety of typed throws, you gain the ability to throw a variety of different error enums – which you prefer really depends on your coding style.

Raw strings

* [**Watch the video**](https://www.youtube.com/watch?v=e6tuUzmxyOU)

[SE-0200](https://github.com/apple/swift-evolution/blob/master/proposals/0200-raw-string-escaping.md) added the ability to create raw strings, where backslashes and quote marks are interpreted as those literal symbols rather than escapes characters or string terminators. This makes a number of use cases more easy, but regular expressions in particular will benefit.

To use raw strings, place one or more # symbols before your strings, like this:

let rain = #"The "rain" in "Spain" falls mainly on the Spaniards."#

The # symbols at the start and end of the string become part of the string delimiter, so Swift understands that the standalone quote marks around “rain” and “Spain” should be treated as literal quote marks rather than ending the string.

Raw strings allow you to use backslashes too:

let keypaths = #"Swift keypaths such as \Person.name hold uninvoked references to properties."#

That treats the backslash as being a literal character in the string, rather than an escape character. This in turn means that string interpolation works differently:

let answer = 42

let dontpanic = #"The answer to life, the universe, and everything is \#(answer)."#

Notice how I’ve used \#(answer) to use string interpolation – a regular \(answer) will be interpreted as characters in the string, so when you want string interpolation to happen in a raw string you must add the extra #.

One of the interesting features of Swift’s raw strings is the use of hash symbols at the start and end, because you can use more than one in the unlikely event you’ll need to. It’s hard to provide a good example here because it really ought to be extremely rare, but consider this string: **My dog said "woof"#gooddog**. Because there’s no space before the hash, Swift will see "# and immediately interpret it as the string terminator. In this situation we need to change our delimiter from #" to ##", like this:

let str = ##"My dog said "woof"#gooddog"##

Notice how the number of hashes at the end must match the number at the start.

Raw strings are fully compatible with Swift’s multi-line string system – just use #""" to start, then """# to end, like this:

let multiline = #"""

The answer to life,

the universe,

and everything is \#(answer).

"""#

Being able to do without lots of backslashes will prove particularly useful in regular expressions. For example, writing a simple regex to find keypaths such as \Person.nameused to look like this:

let regex1 = "\\\\[A-Z]+[A-Za-z]+\\.[a-z]+"

Thanks to raw strings we can write the same thing with half the number of backslashes:

let regex2 = #"\\[A-Z]+[A-Za-z]+\.[a-z]+"#

We still need *some*, because regular expressions use them too.

Customizing string interpolation

[SE-0228](https://github.com/apple/swift-evolution/blob/master/proposals/0228-fix-expressiblebystringinterpolation.md) dramatically revamped Swift’s string interpolation system so that it’s more efficient and more flexible, and it’s creating a whole new range of features that were previously impossible.

In its most basic form, the new string interpolation system lets us control how objects appear in strings. Swift has default behavior for structs that is helpful for debugging, because it prints the struct name followed by all its properties. But if you were working with classes (that don’t have this behavior), or wanted to format that output so it could be user-facing, then you could use the new string interpolation system.

For example, if we had a struct like this:

struct User {

 var name: String

 var age: Int

}

If we wanted to add a special string interpolation for that so that we printed users neatly, we would add an extension to String.StringInterpolation with a new appendInterpolation()method. Swift already has several of these built in, and users the interpolation *type* – in this case User to figure out which method to call.

In this case, we’re going to add an implementation that puts the user’s name and age into a single string, then calls one of the built-in appendInterpolation() methods to add that to our string, like this:

extension String.StringInterpolation {

 mutating func appendInterpolation(\_ value: User) {

 appendInterpolation("My name is \(value.name) and I'm \(value.age)")

 }

}

Now we can create a user and print out their data:

let user = User(name: "Guybrush Threepwood", age: 33)

print("User details: \(user)")

That will print **User details: My name is Guybrush Threepwood and I'm 33**, whereas with the custom string interpolation it would have printed **User details: User(name: "Guybrush Threepwood", age: 33)**. Of course, that functionality is no different from just implementing the CustomStringConvertible protocol, so let’s move on to more advanced usages.

Your custom interpolation method can take as many parameters as you need, labeled or unlabeled. For example, we could add an interpolation to print numbers using various styles, like this:

extension String.StringInterpolation {

 mutating func appendInterpolation(\_ number: Int, style: NumberFormatter.Style) {

 let formatter = NumberFormatter()

 formatter.numberStyle = style

 if let result = formatter.string(from: number as NSNumber) {

 appendLiteral(result)

 }

 }

}

The NumberFormatter class has a number of styles, including currency ($72.83), ordinal (1st, 12th), and spell out (five, forty-three). So, we could create a random number and have it spelled out into a string like this:

let number = Int.random(in: 0...100)

let lucky = "The lucky number this week is \(number, style: .spellOut)."

print(lucky)

You can call appendLiteral() as many times as you need, or even not at all if necessary. For example, we could add a string interpolation to repeat a string multiple times, like this:

extension String.StringInterpolation {

 mutating func appendInterpolation(repeat str: String, \_ count: Int) {

 for \_ in 0 ..< count {

 appendLiteral(str)

 }

 }

}

print("Baby shark \(repeat: "doo ", 6)")

And, as these are just regular methods, you can use Swift’s full range of functionality. For example, we might add an interpolation that joins an array of strings together, but if that array is empty execute a closure that returns a string instead:

extension String.StringInterpolation {

 mutating func appendInterpolation(\_ values: [String], empty defaultValue: @autoclosure () -> String) {

 if values.count == 0 {

 appendLiteral(defaultValue())

 } else {

 appendLiteral(values.joined(separator: ", "))

 }

 }

}

let names = ["Harry", "Ron", "Hermione"]

print("List of students: \(names, empty: "No one").")

Using @autoclosure means that we can use simple values or call complex functions for the default value, but none of that work will be done unless values.count is zero.

With a combination of the ExpressibleByStringLiteral and ExpressibleByStringInterpolationprotocols it’s now possible to create whole types using string interpolation, and if we add CustomStringConvertible we can even make those types print as strings however we want.

To make this work, we need to fulfill some specific criteria:

* Whatever type we create should conform to ExpressibleByStringLiteral, ExpressibleByStringInterpolation, and CustomStringConvertible. The latter is only needed if you want to customize the way the type is printed.
* *Inside* your type needs to be a nested struct called StringInterpolation that conforms to StringInterpolationProtocol.
* The nested struct needs to have an initializer that accepts two integers telling us roughly how much data it can expect.
* It also needs to implement an appendLiteral() method, as well as one or more appendInterpolation() methods.
* Your main type needs to have two initializers that allow it to be created from string literals and string interpolations.

We can put all that together into an example type that can construct HTML from various common elements. The “scratchpad” inside the nested StringInterpolation struct will be a string: each time a new literal or interpolation is added, we’ll append it to the string. To help you see exactly what’s going on, I’ve added some print() calls inside the various append methods.

Here’s the code.

struct HTMLComponent: ExpressibleByStringLiteral, ExpressibleByStringInterpolation, CustomStringConvertible {

 struct StringInterpolation: StringInterpolationProtocol {

 // start with an empty string

 var output = ""

 // allocate enough space to hold twice the amount of literal text

 init(literalCapacity: Int, interpolationCount: Int) {

 output.reserveCapacity(literalCapacity \* 2)

 }

 // a hard-coded piece of text – just add it

 mutating func appendLiteral(\_ literal: String) {

 print("Appending \(literal)")

 output.append(literal)

 }

 // a Twitter username – add it as a link

 mutating func appendInterpolation(twitter: String) {

 print("Appending \(twitter)")

 output.append("<a href=\"https://twitter/\(twitter)\">@\(twitter)</a>")

 }

 // an email address – add it using mailto

 mutating func appendInterpolation(email: String) {

 print("Appending \(email)")

 output.append("<a href=\"mailto:\(email)\">\(email)</a>")

 }

 }

 // the finished text for this whole component

 let description: String

 // create an instance from a literal string

 init(stringLiteral value: String) {

 description = value

 }

 // create an instance from an interpolated string

 init(stringInterpolation: StringInterpolation) {

 description = stringInterpolation.output

 }

}

We can now create and use an instance of HTMLComponent using string interpolation like this:

let text: HTMLComponent = "You should follow me on Twitter \(twitter: "twostraws"), or you can email me at \(email: "paul@hackingwithswift.com")."

 print(text)

Thanks to the print() calls that were scattered inside, you’ll see exactly how the string interpolation functionality works: you’ll see “Appending You should follow me on Twitter”, “Appending twostraws”, “Appending , or you can email me at “, “Appending paul@hackingwithswift.com”, and finally “Appending .” – each part triggers a method call, and is added to our string.

Dynamically callable types

[SE-0216](https://github.com/apple/swift-evolution/blob/master/proposals/0216-dynamic-callable.md) adds a new @dynamicCallable attribute to Swift, which brings with it the ability to mark a type as being directly callable. It’s syntactic sugar rather than any sort of compiler magic, effectively transforming this code:

let result = random(numberOfZeroes: 3)

Into this:

let result = random.dynamicallyCall(withKeywordArguments: ["numberOfZeroes": 3])

Previously I wrote about a [feature in Swift 4.2 called @dynamicMemberLookup](https://www.hackingwithswift.com/articles/55/how-to-use-dynamic-member-lookup-in-swift). @dynamicCallable is the natural extension of @dynamicMemberLookup, and serves the same purpose: to make it easier for Swift code to work alongside dynamic languages such as Python and JavaScript.

To add this functionality to your own types, you need to add the @dynamicCallable attribute plus one or both of these methods:

func dynamicallyCall(withArguments args: [Int]) -> Double

func dynamicallyCall(withKeywordArguments args: KeyValuePairs<String, Int>) -> Double

The first of those is used when you call the type without parameter labels (e.g. a(b, c)), and the second is used when you *do* provide labels (e.g. a(b: cat, c: dog)).

@dynamicCallable is really flexible about which data types its methods accept and return, allowing you to benefit from all of Swift’s type safety while still having some wriggle room for advanced usage. So, for the first method (no parameter labels) you can use anything that conforms to ExpressibleByArrayLiteral such as arrays, array slices, and sets, and for the second method (with parameter labels) you can use anything that conforms to ExpressibleByDictionaryLiteral such as dictionaries and key value pairs.

* **Note:** If you haven’t used KeyValuePairs before, now would be a good time to learn what they are because they are extremely useful with @dynamicCallable. Find out more here: [What are KeyValuePairs?](https://www.hackingwithswift.com/example-code/language/what-are-keyvaluepairs)

As well as accepting a variety of inputs, you can also provide multiple overloads for a variety of outputs – one might return a string, one an integer, and so on. As long as Swift is able to resolve which one is used, you can mix and match all you want.

Let’s look at an example. First, here’s a RandomNumberGenerator struct that generates numbers between 0 and a certain maximum, depending on what input was passed in:

struct RandomNumberGenerator {

 func generate(numberOfZeroes: Int) -> Double {

 let maximum = pow(10, Double(numberOfZeroes))

 return Double.random(in: 0...maximum)

 }

}

To switch that over to @dynamicCallable we’d write something like this instead:

@dynamicCallable

struct RandomNumberGenerator {

 func dynamicallyCall(withKeywordArguments args: KeyValuePairs<String, Int>) -> Double {

 let numberOfZeroes = Double(args.first?.value ?? 0)

 let maximum = pow(10, numberOfZeroes)

 return Double.random(in: 0...maximum)

 }

}

That method can be called with any number of parameters, or perhaps zero, so we read the first value carefully and use nil coalescing to make sure there’s a sensible default.

We can now create an instance of RandomNumberGenerator and call it like a function:

let random = RandomNumberGenerator()

let result = random(numberOfZeroes: 0)

If you had used dynamicallyCall(withArguments:) instead – or at the same time, because you can have them both a single type – then you’d write this:

@dynamicCallable

struct RandomNumberGenerator {

 func dynamicallyCall(withArguments args: [Int]) -> Double {

 let numberOfZeroes = Double(args[0])

 let maximum = pow(10, numberOfZeroes)

 return Double.random(in: 0...maximum)

 }

}

let random = RandomNumberGenerator()

let result = random(0)

There are some important rules to be aware of when using @dynamicCallable:

* You can apply it to structs, enums, classes, and protocols.
* If you implement withKeywordArguments: and don’t implement withArguments:, your type can still be called without parameter labels – you’ll just get empty strings for the keys.
* If your implementations of withKeywordArguments: or withArguments: are marked as throwing, calling the type will also be throwing.
* You can’t add @dynamicCallable to an extension, only the primary definition of a type.
* You can still add other methods and properties to your type, and use them as normal.

Perhaps more importantly, there is no support for method resolution, which means we must call the type directly (e.g. random(numberOfZeroes: 5)) rather than calling specific methods on the type (e.g. random.generate(numberOfZeroes: 5)). There is already some discussion on adding the latter using a method signature such as this:

func dynamicallyCallMethod(named: String, withKeywordArguments: KeyValuePairs<String, Int>)

If that became possible in future Swift versions it might open up some very interesting possibilities for test mocking.

In the meantime, @dynamicCallable is not likely to be widely popular, but it *is* hugely important for a small number of people who want interactivity with Python, JavaScript, and other languages.

Handling future enum cases

[SE-0192](https://github.com/apple/swift-evolution/blob/master/proposals/0192-non-exhaustive-enums.md) adds the ability to distinguish between enums that are fixed and enums that might change in the future.

One of Swift’s security features is that it requires all switch statements to be exhaustive – that they must cover all cases. While this works well from a safety perspective, it causes compatibility issues when new cases are added in the future: a system framework might send something different that you hadn’t catered for, or code you rely on might add a new case and cause your compile to break because your switch is no longer exhaustive.

With the @unknown attribute we can now distinguish between two subtly different scenarios: “this default case should be run for all other cases because I don’t want to handle them individually,” and “I want to handle all cases individually, but if anything comes up in the future use this rather than causing an error.”

Here’s an example enum:

enum PasswordError: Error {

 case short

 case obvious

 case simple

}

We could write code to handle each of those cases using a switch block:

func showOld(error: PasswordError) {

 switch error {

 case .short:

 print("Your password was too short.")

 case .obvious:

 print("Your password was too obvious.")

 default:

 print("Your password was too simple.")

 }

}

That uses two explicit cases for short and obvious passwords, but bundles the third case into a default block.

Now, if in the future we added a new case to the enum called old, for passwords that had been used previously, our default case would automatically be called even though its message doesn’t really make sense – the password might not be too simple.

Swift can’t warn us about this code because it’s technically correct (the best kind of correct), so this mistake would easily be missed. Fortunately, the new @unknown attribute fixes it perfectly – it can be used only on the default case, and is designed to be run when new cases come along in the future.

For example:

func showNew(error: PasswordError) {

 switch error {

 case .short:

 print("Your password was too short.")

 case .obvious:

 print("Your password was too obvious.")

 @unknown default:

 print("Your password wasn't suitable.")

 }

}

That code will now issue warnings because the switch block is no longer exhaustive – Swift wants us to handle each case explicitly. Helpfully this is only a *warning*, which is what makes this attribute so useful: if a framework adds a new case in the future you’ll be warned about it, but it won’t break your source code.

Flattening nested optionals resulting from try?

[SE-0230](https://github.com/apple/swift-evolution/blob/master/proposals/0230-flatten-optional-try.md) modifies the way try? works so that nested optionals are flattened to become regular optionals. This makes it work the same way as optional chaining and conditional typecasts, both of which flatten optionals in earlier Swift versions.

Here’s a practical example that demonstrates the change:

struct User {

 var id: Int

 init?(id: Int) {

 if id < 1 {

 return nil

 }

 self.id = id

 }

 func getMessages() throws -> String {

 // complicated code here

 return "No messages"

 }

}

let user = User(id: 1)

let messages = try? user?.getMessages()

The User struct has a failable initializer, because we want to make sure folks create users with a valid ID. The getMessages() method would in theory contain some sort of complicated code to get a list of all the messages for the user, so it’s marked as throws; I’ve made it return a fixed string so the code compiles.

The key line is the last one: because the user is optional it uses optional chaining, and because getMessages() can throw it uses try? to convert the throwing method into an optional, so we end up with a nested optional. In Swift 4.2 and earlier this would make messages a String?? – an optional optional string – but in Swift 5.0 and later try? won’t wrap values in an optional if they are already optional, so messages will just be a String?.

This new behavior matches the existing behavior of optional chaining and conditional typecasting. That is, you could use optional chaining a dozen times in a single line of code if you wanted, but you wouldn’t end up with 12 nested optionals. Similarly, if you used optional chaining with as?, you would still end up with only one level of optionality, because that’s usually what you want.

Checking for integer multiples

* [**Watch the video**](https://www.youtube.com/watch?v=iCRwqxON8Os)

[SE-0225](https://github.com/apple/swift-evolution/blob/master/proposals/0225-binaryinteger-iseven-isodd-ismultiple.md) adds an isMultiple(of:) method to integers, allowing us to check whether one number is a multiple of another in a much clearer way than using the division remainder operation, %.

For example:

let rowNumber = 4

if rowNumber.isMultiple(of: 2) {

 print("Even")

} else {

 print("Odd")

}

Yes, we could write the same check using if rowNumber % 2 == 0 but you have to admit that’s less clear – having isMultiple(of:) as a method means it can be listed in code completion options in Xcode, which aids discoverability.

Transforming and unwrapping dictionary values with compactMapValues()

* [**Watch the video**](https://www.youtube.com/watch?v=Le32Tbkv2v0)

[SE-0218](https://github.com/apple/swift-evolution/blob/master/proposals/0218-introduce-compact-map-values.md) adds a new compactMapValues() method to dictionaries, bringing together the compactMap() functionality from arrays (“transform my values, unwrap the results, then discard anything that’s nil”) with the mapValues() method from dictionaries (“leave my keys intact but transform my values”).

As an example, here’s a dictionary of people in a race, along with the times they took to finish in seconds. One person did not finish, marked as “DNF”:

let times = [

 "Hudson": "38",

 "Clarke": "42",

 "Robinson": "35",

 "Hartis": "DNF"

]

We can use compactMapValues() to create a new dictionary with names and times as an integer, with the one DNF person removed:

let finishers1 = times.compactMapValues { Int($0) }

Alternatively, you could just pass the Int initializer directly to compactMapValues(), like this:

let finishers2 = times.compactMapValues(Int.init)

You can also use compactMapValues() to unwrap optionals and discard nil values without performing any sort of transformation, like this:

let people = [

 "Paul": 38,

 "Sophie": 8,

 "Charlotte": 5,

 "William": nil

]

let knownAges = people.compactMapValues { $0 }

Withdrawn: Counting matching items in a sequence

* [**Watch the video**](https://www.youtube.com/watch?v=syPKtPb0y-Y)

**This Swift 5.0 feature was withdrawn in beta testing because it was causing performance issues for the type checker. Hopefully it will come back in time for Swift 5.1, perhaps with a new name to avoid problems.**

[SE-0220](https://github.com/apple/swift-evolution/blob/master/proposals/0220-count-where.md) introduces a new count(where:) method that performs the equivalent of a filter() and count in a single pass. This saves the creation of a new array that gets immediately discarded, and provides a clear and concise solution to a common problem.

This example creates an array of test results, and counts how many are greater or equal to 85:

let scores = [100, 80, 85]

let passCount = scores.count { $0 >= 85 }

And this counts how many names in an array start with “Terry”:

let pythons = ["Eric Idle", "Graham Chapman", "John Cleese", "Michael Palin", "Terry Gilliam", "Terry Jones"]

let terryCount = pythons.count { $0.hasPrefix("Terry") }

This method is available to all types that conform to Sequence, so you can use it on sets and dictionaries too.

Where next?

Swift 5.0 is the latest release of Swift, but previous releases have been packed with great features too. You can read my articles on those below:

* [What's new in Swift 4.2?](https://www.hackingwithswift.com/articles/77/whats-new-in-swift-4-2)
* [What's new in Swift 4.1?](https://www.hackingwithswift.com/articles/50/whats-new-in-swift-4-1)
* [What's new in Swift 4.0?](https://www.hackingwithswift.com/swift4)

But there's more to come – Apple already announced the [Swift 5.1 release process](https://swift.org/blog/5-1-release-process/) on Swift.org, which will include module stability alongside some other improvements. At the time of writing there are very few hard dates attached to 5.1, but it's looking like we'll see it ship in beta around WWDC.

If you'd like to learn more about what's changing in Swift 5.1, see my article: [what's new in Swift 5.1](https://www.hackingwithswift.com/articles/182/whats-new-in-swift-5-1).