O'REILLY®

What's New in Swift 3

Paris Buttfield-Addison,
Jon Manning
& Tim Nugent

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

What's New in Swift 3

Paris Buttfield-Addison, Jon Manning, and Tim Nugent

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

What's New in Swift 3
by Paris Buttfield-Addison, Jon Manning, and Tim Nugent

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Rachel Roumeliotis
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2016: First Edition

Revision History for the First Edition
2016-10-18: First Release

978-1-491-96667-9

[LSI]

Table of Contents

CHAPTER 1
Introduction

Swift was introduced to the world in 2014 and has rapidly evolved since then, eventually being released as an open source project in late 2015. Swift has been one of the fastest-growing programming languages in history, by a variety of metrics, and is worth serious consideration regardless of your level of programming experience or the size and age of your project's code.

Swift was designed to be a complete replacement for Objective-C, the language used for all iOS and Mac OS development prior to Swift's release. Swift is ideal for new projects; additionally, because you can easily use Swift and Objective-C in the same project, you can incrementally convert your existing Objective-C code to Swift.

Swift has truly been released into the open: the conceptualization, discussion, and development of new language features, direction decisions, and changes to the features all take place on open mailing lists, with the wider community of Swift users. This is important, because it means that the direction of the language is in the hands of users and not the exclusive domain of a central planning group.

In this report, we're going to look at Swift 3, released in September 2016, and the first significant release from the open source Swift project. Specifically, we're going to look at three facets of Swift 3 programming and the ecosystem around its use:

Chapter 2
: We'll begin with a discussion of Swift 3, exploring what's changed at a high level, and how the community organizes the

evolution and open source development process for Swift. Here, we'll also give you an overview of what's in Swift 3, how it differs from Swift 2, and the new features and changes you can expect if you're already developing with Swift.

Chapter 3
: In this chapter, we'll explore the new standard library features, syntax changes, and other new parts of the Swift 3 release, and how they differ from the old stuff. We'll focus on the most impactful and interesting changes for those programmers already familiar with Swift 2.

Chapter 4
: We'll discuss Swift on the server, Linux, and non-Apple platforms.

Finally, the report will conclude with some suggestions on where to go next as you learn, work with, or convert your projects to Swift 3, or Swift in general.

CHAPTER 2

Understanding Swift 3

In this chapter, we'll explore the Swift 3 release of the Swift programming language. First, we'll explain how the Swift evolution process works. We'll then look at the high-level changes and objectives for Swift 3, as well as how it differs from Swift 2, and the list of changes and additions that you can expect in Swift 3.

The Swift Evolution Process

To understand Swift 3, we're going to first touch on the Swift evolution process before examining the list of changes planned in Swift 3. It's important to understand how these changes made it into the Swift language, because this is the first time Apple has developed a language in consultation with a larger community of developers, rather than it being limited to Apple's own engineers. As a result, a broader range of changes have been made in the language than would ordinarily appear.

One of the most impressive aspects of the Swift open source project is how the Swift evolution processes established by Apple allowed users to contribute to version 3. While this report isn't about the Swift evolution process, it's necessary to understand it in order to comprehend the way Swift 3 is changing.

The Swift evolution process is designed to give everyone a chance to discuss changes to the Swift language and the Swift standard library, including additions, removals, and modifications to language features and APIs, no matter how small they might be. The evolution

process is not for bug fixes, optimizations, and other small improvements; those happen through a more conventional open source contribution process.

Changes proposed through the Swift evolution process must not duplicate existing ones, must be presented to and discussed by the community, and must be presented using a template, before being submitted for review via a GitHub pull request on the Swift evolution project repository. Proposals are typically in support of the goals of upcoming Swift releases, which are defined in advance; otherwise they may be deferred for a future release or rejected. Through this process, complete and reasonable Swift evolution (SE) proposals are numbered (e.g., SE-0004); scheduled for review; and eventually accepted, rejected, deferred, or revised. Once a Swift evolution proposal is accepted, it's allocated to an upcoming Swift release and scheduled for development by the Swift team.

So, Swift 3, then?

The Swift open source project reports that the primary goal of the Swift 3 release is "to solidify and mature the Swift language and development experience." Thus they plan to make future versions of Swift, following Swift 3, as source compatible as is reasonably possible. The core goals, in support of the primary goal, are:

To provide API design guidelines
: Lots of libraries and APIs for Swift have already been created by the community, and with Swift 3, the team saw fit to create a set of guidelines for the naming and design of APIs to assist in this.

To establish naming guidelines and Swiftification for key APIs and Objective-C APIs
: Objective-C APIs that get imported are designed for an improved Swift programming experience and are automatically mapped to names complying with the Swift naming guidelines; and the Swift standard library itself now complies with the new API design guidelines.

To refine the language
: In line with the objective of making Swift 3 the last release to have major changes that break source code compatibility, there are refinements to the syntax and semantics as a whole (e.g., parameter labels and the type system).

To improve the tooling
: Swift was created by compiler and language design academics, and performance of the compiled output is a focus of the Swift 3 release.

To improve portability to non-Apple platforms
: The Swift language is designed to be adopted and used across a myriad platforms, and a useful, functional version for Linux is already available. Swift on the server will enable a lot of interesting projects, and we'll touch on some of them later on in this report.

Swift 3's focus is on stability and ease of use of the language. Judging from the changes that were made from Swift 2, we can see a clear indication of the direction that Apple's taking: one in which the community is working to improve the language, which itself is becoming cleaner and more expressive and enjoyable to write in.

CHAPTER 3

What's Changed in Swift 3?

In this chapter, we'll explore the new standard library features, syntax changes, and other new parts of the Swift 3 release, and explain how they differ from the old stuff.

In the previous chapter, we explained how the Swift evolution process works. When a proposal gets accepted for a Swift release, in this case Swift 3, there are two states it could be in:

1. Accepted and implemented, or mostly implemented
2. Accepted but not implemented yet

Proposals in the second state have the potential to be held for a subsequent Swift release if they're not implemented in time.

The full list of accepted proposals, in both states, is available at the Swift evolution project. There are too many to list here, so you should take a look at the website to get an idea of the magnitude of the changes. Go ahead; we'll wait.

Using the New Stuff

In this report, we've selected some of the most important, impactful, or otherwise interesting changes that are being made to the language in version 3. Here's the list:

- The API guidelines are applied to the Swift standard library.
- The `++` and `--` operators have been removed.

- C-style `for`-loops have been removed.
- `libdispatch` now has a nicer, Swiftier API.
- First parameters in functions now have labels.
- Foundation types are now imported as Swift types.
- Objective-C lightweight generic types are imported as Swift generic types.
- Function parameters may no longer be variables, and are now always constants.
- Selectors and key paths are now type-checked.
- `UpperCamelCase` has become `lowerCamelCase` for enums and properties.
- `M_PI` is now `Float.pi`.
- Some symbols have been deprecated.
- Functions can be marked as having a result that can be ignored.
- Debugging identifiers have been made nicer.

Let's take a look at each of these, one by one, to get a better idea of how they impact the Swift language, and how things differ from Swift 2.

The API Guidelines Are Applied to the Swift Standard Library

One of the largest changes in Swift 3 is the adoption of a single, consistent set of guidelines that apply to the naming of methods and types, as well as the design of your programs.

The full specification of the API design guidelines, while lengthy, is not hugely complex. It's primarily concerned with consistent naming schemes, and establishing coding conventions as part of a larger effort to establish a unifying "Swift style." If you follow these guidelines—and you should!—then your code will feel a lot more Swift-like. You can find the API guidelines on the Swift.org site.

Adopting the API guidelines was a significant task and involves three concurrent Swift evolution proposals: SE-0023 "API Design Guidelines," SE-0006 "Apply API Guidelines to the Standard Library," and SE-0005 "Better Translation of Objective-C APIs Into Swift." The first proposal establishes the guidelines themselves, the

second describes how the standard library needs to be modified in order to comply with them, and the third describes how to import Objective-C code in order to make the imported APIs comply with the guidelines.

As part of the efforts to apply the API guidelines, several methods in the standard library have been renamed. In the new API guidelines, methods whose names are verb phrases (like `sort`) have side effects, while methods whose names that have no side effects and simply return a value have "-ed" appended (like `sorted`).

For example, if we have a variable containing an array of numbers:

```
var numbers = [5, 17, 1]
```

and then we run `sort` on it:

```
numbers.sort()
```

`sort` is a verb, and this modifies the numbers variable in place; so if we then print the numbers variable:

```
print(numbers)
```

the output will be `1, 5, 17`. Whereas if we start with an array of numbers, and call `sorted` on them, we'll end up with the sorted results returned, rather than changed in place:

```
var moreNumbers = [10, 42, 3]

print(moreNumbers.sorted()) // prints [3, 10, 42]

print(moreNumbers) // prints [10, 42, 3], unchanged
```

Finally, SE-0005 ("Better Translation of Objective-C APIs Into Swift") creates compatibility between Swift that follows the API guidelines and earlier Objective-C code that follows its *own* API guidelines. This is big news in a variety of ways, but a core feature of this SE proposal is that wordy Objective-C APIs will, when used with Swift, omit needless words; therefore, words that restate things that Swift's compiler already enforces will no longer be needed. In Swift 2, you might have added an entry to an array:

```
myArray.insert("Fido", atIndex: 10)
```

Now, in Swift 3, a needless word is omitted:

```
myArray.insert("Fido", at: 10)
```

Of course, this also applies to the significantly more verbose Cocoa/CocoaTouch APIs, which originated back in the Objective-C days. For example, the Swift 2 code:

```
"Take command, Mr Riker"
    .stringByReplacingOccurrencesOfString("command",
    withString: "the conn")
```

becomes, in Swift 3:

```
"Take command, Mr Riker"
    .replacingOccurrences(of: "command", with: "the conn")
```

The ++ and -- Operators Have Been Removed

The accepted Swift evolution proposal SE-0004 advocates for the removal of the legacy increment and decrement operators, which were originally included, inspired by C, without much thought.

For example, the following operators are available in Swift 2:

```
// post-increment, returning a copy of x
// before it's incremented
let q = x++

// pre-increment, returning a copy of x after it's incremented
let p = ++x
```

In Swift 3, these increment and decrement operators (we only showed increment in the preceding example), while they're reasonably expressive shorthand, and provide consistency with C-based and inspired languages, aren't obvious to new programmers and are not particularly shorter than the alternative. Additionally, many of the reasons for using these kinds of operators, such as `for`-loops, ranges, enumerations, and maps are less relevant to Swift.

Instead, in Swift 3, you can increment and decrement through the standard operators:

```
var x = 1

x = x + 1 // x is now 2

x = x - 1 // x is now 1
```

Or using the addition and subtraction assignment operators:

```
var x = 1

x += 1 // x is now 2
```

```
x -= 1 // x is now 1
```

You can learn more about the rationale behind the change in the original Swift evolution proposal document.

C-style for-loops Have Been Removed

The accepted Swift evolution proposal SE-0007 suggests the removal of C-style `for`-loops, suggesting that they're a thoughtless carry-over from C, rather than something useful, and Swifty, for Swift. As a reminder, a C-style `for`-loop in Swift looked like this:

```
for (i = 1; i <= 5; i++) {
    print(i)
}
```

The more Swifty way of doing things is:

```
for i in 1...5 {
    print(i)
}
```

or, using shorthand arguments and closures, and being very Swifty in style:

```
(1...5).forEach{
    print($0)
}
```

If you want to learn more, check out the original Swift evolution proposal document for the change.

libdispatch Now Has a Swiftier API

Accepted Swift evolution proposal SE-0088 suggests that the API for Grand Central Dispatch (GCD) be modernized and made Swiftier in style. GCD is a collection of features and libraries that provides relatively straightforward multicore concurrency support on Apple platforms.

Formerly, using GDC in Swift involved using calls that were very C-like and often verbatim copies of the underling C-based API, for example:

```
let queue = dispatch_queue_create("com.test.myqueue", nil)

dispatch_async(queue) {
   print("Hello World")
}
```

In Swift 3, the surface-level GCD API has been renamed with something that better resembles a Swifty approach and is easier to read and understand:

```
let queue = DispatchQueue(label: "com.test.myqueue")

queue.async {
   print("Hello World")
}
```

It may seem like a simple change on the surface, but this will make concurrent code a lot more readable and easier to understand for Swift programmers. You can learn more about this change from the accepted Swift evolution proposal document.

First Parameters in Functions Now Have Labels

The accepted Swift evolution proposal SE-0046 proposes to "normalize the first parameter declaration in methods and functions." This means that parameters will be simpler to read (since they're actually all parameters, instead of part of the method name, which was somewhat of a carry-over from Objective-C). Here's an example—in Swift 2, you might have called a method:

```
Dog.feedFoodType(specialMix, quantity: 5)
```

But in Swift 3, the same thing would be:

```
Dog.feed(foodType: specialMix, quantity: 5)
```

This change makes it a lot easier to read methods and will make Swift easier to learn since this is consistent with the way other languages behave. It also makes Swift methods and functions consistent with the way Swift initializers already work.

You can read more about this change in the accepted Swift evolution proposal document.

Foundation Types Are Now Imported as Swift Types

Right now, most Foundation types have a prefix of `NS`. This is for historical reasons that we don't need to go into; but if you're curious, check out the "Historical Note" in Apple's documentation. Swift evolution proposal SE-0086 suggests that this prefix be removed, and accepted Swift evolution proposal SE-0069 further documents the changes.

For example, `NSArray` is now imported as just `Array`, `NSString` is imported as `String`, and so on. This is important not just because of the name simplification, but because Swift `String`s are value types, while `NSString` is a reference type.

Additionally, certain Foundation types have been renamed to follow this pattern. For example, in Swift 2, you'd create a *mass formatter*, a tool for formatting the values of physical masses, like this:

```
let formatter = NSMassFormatter()
```

In Swift 3, this becomes:

```
let formatter = MassFormatter()
```

This simplification of the API means cleaner, easier-to-read code. Note that this simplification only applies to the Foundation classes, and not to other frameworks. This means that you'll still need to use `NS` and `UI` prefix on classes from those frameworks.

Objective-C Lightweight Generic Types Are Imported as Swift Generic Types

In recent versions of Objective-C, you can define an `NSArray` with a type. For example, this defines an array of strings:

```
NSArray* <NSString*> arrayOfStrings;
```

This is now imported into Swift as:

```
var arrayOfStrings : [String]
```

There are some caveats and limitations, which result from the fact that Objective-C generics aren't represented at runtime. For more information, see the Swift evolution proposal.

Function Parameters Are No Longer Variables, but Constants

In Swift 2, you used to be able to declare a parameter as a `var`, which allowed you to make modifications to the local copy of the value that the function received. This has been removed in Swift 3, because there's no significant benefit in being able to do so: even if you make changes to a parameter, that won't change anything outside the function.

For example, you used to be able to do this:

```
func foo(var i: Int) {
    i += 1
}
```

In Swift 3, if you really need to be able to do this sort of thing, you'll need to create your own `var` from a parameter and modify that:

```
func foo(i: Int) {
    var localI = i
    localI += 1
}
```

Selectors and Key Paths Are Now Type-Checked

In Swift 2, selectors and key paths were strings:

```
control.sendAction("doSomething", to: target, forEvent: event)
```

This isn't type-checked, which means that typos can cause problems at runtime.

In Swift 3, the compiler now uses the `#selector` and `#keypath` keywords to signal when selector or key path is used:

```
control.sendAction(#selector(MyApplication.doSomething),
to: target, forEvent: event)
```

UpperCamelCase Has Become lowerCamelCase for Enums and Properties

In Swift 2, UpperCamelCase is used for enums and properties, so if you wanted to access a property or enumeration, you'd have done something like this:

```
UIControlEvents.EditingDidBegin
```

In Swift 3, this has been changed to make things more consistent with the rest of the language:

```
UIControlEvents.editingDidBegin
```

M_PI is now Float.pi

As part of broader efforts to obey the new API design guidelines and make Swift easier to write, in Swift 3 `pi` is a constant on the type for which you want to access `pi`:

```
Float.pi

Double.pi
```

Because Swift has a powerful type inference system, you can actually omit the type entirely and work directly with `pi`, like this:

```
let r = 3.0 / .pi
```

Additionally, the `random` function is gone, and you should now use `arc4random` instead.

Functions Can Be Marked as Having a Discardable Result

You can now flag that a mutating function of method can have its return value ignored. The compiler will warn you if you don't use the return value of a function or method and it doesn't have `@discardableResult` above it:

```
@discardableResult
func add(a: Int, b:Int) -> Int {
    return a + b
}

add(a: 1,b: 2)
// won't emit a warning, even though we didn't use the result
```

Debugging Identifiers Have Been Made Nicer

Swift 2.2 included a collection of useful debugging identifiers, and Swift 3 builds on this. Identifiers include `#function`, `#line`, and `#file`.

They are primarily designed for ease of debugging, and you can use them like this:

```
func a() {
    print ("I'm in \(#function) in \(#file) at" +
        "line \(#line) column \(#column)")
}

a()
```

You can read more about these in the original Swift evolution proposal, SE-0028.

Putting It All Together

One of the best ways to get a holistic understanding of what makes Swift 3 a little different from Swift 2 is to assemble a more complete

program. Here's a simple program, designed to be run in Swift Playgrounds (on iOS or Mac OS), that showcases some of the changes.

The complete playground is available at *https://github.com/thesecretlab/Swift3Report*.

The playground shows a big circle and a button that you can tap or click to increment a counter that changes the color of the circle each time. Let's take a look!

First, we import a few things, like `UIKit`, as well as `PlaygroundSupport`:

```
import UIKit
import PlaygroundSupport
```

`PlaygroundSupport` is available in Mac OS playgrounds as well.

Next we'll create a class for our demo, with some variables for our label, which will show how many times we tap the screen; our image view, which will display the colored circle; and an integer to count the taps:

```
class DemoViewController : UIViewController {

    var label : UILabel!
    var imageView = UIImageView()
    var tapCount = 0
```

Now we need a function to actually draw our colored circle. It's going to take a size and a color as parameters:

```
// Draws and returns an image
func drawImage(size : CGSize, color: UIColor) -> UIImage? {
```

Create a canvas to do the drawing in, as well as a deferred call to end the context when everything is over:

```
// Create a canvas for drawing
UIGraphicsBeginImageContext(size)

// When this function exits, tear down the canvas
defer {
    UIGraphicsEndImageContext()
}
```

Then, get a context to draw in. We're using the newly simplified and Swiftier Core Graphics API. In Swift 3, we can use the Core Graphics context like a regular object, and we don't have to repeatedly call verbose Core Graphics functions. We'll also fill the context with the color that was passed into the function and create an ellipse:

```
    // Get a context for drawing with
    let context = UIGraphicsGetCurrentContext()

    // 'context' can now be used like an object, whereas
    // it couldn't in Swift 2
    context?.setFillColor(color.cgColor)
    context?.fillEllipse(in: CGRect(x: 0, y: 0, width:
        size.width, height: size.height))
```

Then, finishing off our image drawing function, we'll return the image from the context:

```
        // Return the image now in the canvas
        return UIGraphicsGetImageFromCurrentImageContext()
    }
```

Next we need to override the function called when a view loads and actually put things on the screen, starting with the label to count clicks:

```
    override func viewDidLoad() {

        label = UILabel()
        label.frame = CGRect(x: 50, y: 50, width: 200, height: 50)
        label.textColor = UIColor.white
        label.text = ""

        self.view.addSubview(label)
```

and then the image, which we created the function to draw earlier:

```
    imageView.frame =
        CGRect(x: 50, y: 150, width: 250, height: 250)
    imageView.image =
        drawImage(size: imageView.frame.size, color: UIColor.red)

    self.view.addSubview(imageView)
```

then a button to tap, for which we'll use the new selector syntax:

```
        let button = UIButton()

        button.setTitle("Tap This Button!", for: [])
        button.frame =
            CGRect(x: 0, y: 0, width: 200, height: 40)
```

```
        // Note the #selector syntax, camel-case enumeration,
        // simplified parameter names
        button.addTarget(self, action: #selector(buttonTapped),
            for: .touchUpInside)

        self.view.addSubview(button)

    }
```

Finally, we need a function to call when the button is tapped. It's going to increment the tap counter, change the label text to reflect that, and randomly pick a new hue for the circle's color:

```
    func buttonTapped() {
        tapCount += 1
        label.text = "Tapped \(tapCount) times"
        let hue = CGFloat(arc4random()) / CGFloat(RAND_MAX)
        let newColor = UIColor(hue: hue,
            saturation: 0.7, brightness: 1.0, alpha: 1.0)
        imageView.image = drawImage(size: imageView.frame.size,
            color: newColor)
    }
}
```

To get it all working in the playground, we set the view controller to be an instance of the new class we defined:

```
let viewController = DemoViewController()
```

Then we tell the playground support system that it should run until we stop it, and that the live view component of the playground (where buttons and such are displayed) should be our view controller:

```
PlaygroundPage.current.needsIndefiniteExecution = true
PlaygroundPage.current.liveView = viewController
```

On a Mac, this playground should now be running, and on an iPad it will run if you press the "Run my code" button! You can see the final result in Figure 3-1.

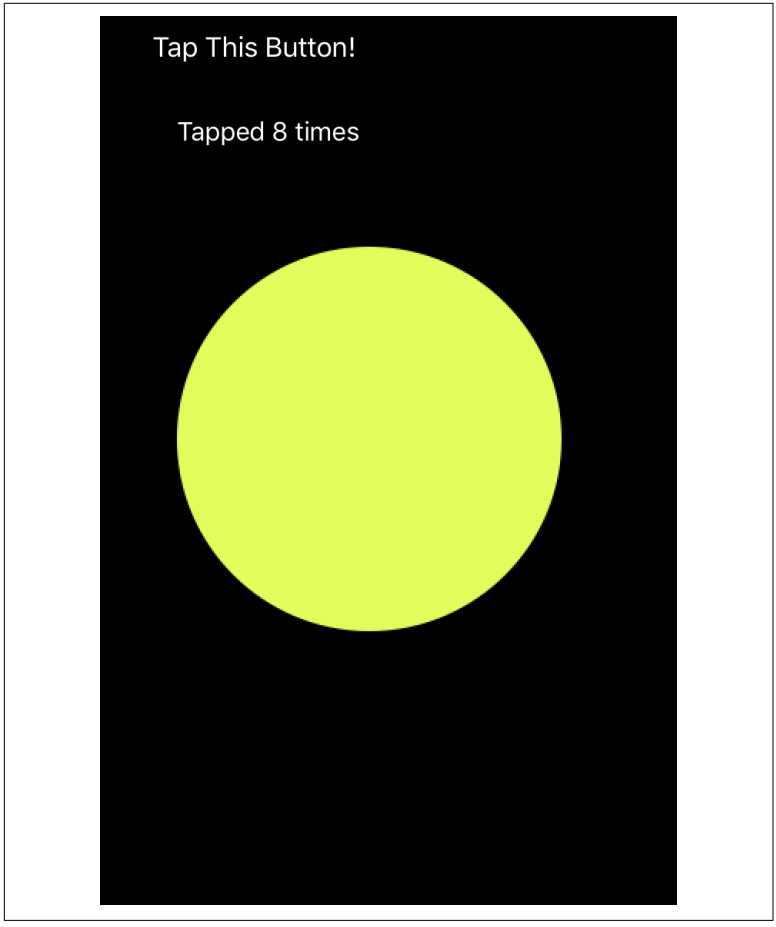

Figure 3-1. The result of running the playground

Summary

A lot has changed in Swift 3. Almost all of these changes take the form of subtle refinements that smooth the rough corners off the language. Human interactions with the Swift language are being made smoother: the removal of prefixes, unnecessary words, and more straightforward bridging of types between Objective-C and Swift are some of the stand-out examples.

CHAPTER 4

Swift on the Server, and Swift on Linux

One of the most exciting aspects of the Swift project is that the language works on non-Apple platforms. You can download binaries of the latest version of Swift for Ubuntu 14.04 and Ubuntu 15.10 from the Swift project website and make use of them right away.

Various contributors to the Swift community are also working on support for Windows, as well as the potential beginnings of Android support. It's an exciting time to be working with Swift!

NOTE

Swift for Windows isn't nearly as ready for production use as Swift for Apple platforms or Swift for Linux. But, in time, we would expect it to reach parity with, at the very least, Swift on Linux. Ars Technica interviewed Apple's SVP of Software Engineering, Craig Federighi, who reported that Windows support isn't something that Apple and their Swift team wishes to take on directly, but that Apple thinks it possible that the development community would take it on.

To get the basics up and running on Linux, you can follow the Getting Started guide provided by the Swift project, but if you want to go a little deeper, we're going to briefly touch on getting Swift set up for server-side development.

Swift on Linux

Swift on Linux offers a huge range of exciting possibilities, from running Swift-based web frameworks (as we'll discuss shortly), to eventually building apps for Linux desktop, Raspberry Pi, or even Android.

In this section, we'll take a quick look at the process of installing Swift on Linux, how to work with Swift on Linux, and what works and what doesn't, yet, in the Linux version.

If you're running a Mac, or Windows, and want to have a go with the Linux version of Swift, you can set it up in Docker, or run Linux inside VirtualBox. Vagrant makes the configuration of Linux, and then Swift, within VirtualBox trivial. We'll set up Vagrant in the next section.

Installing Swift on Linux

We're primarily developers for Apple platforms, so for us the best way to run Swift on Linux is on a Mac using Vagrant and VirtualBox. This lets you play with the version of Swift available for Linux from the comfort of your Mac, in a way that allows you to clear things out and experiment with different versions.

To get Swift on Linux running, on your Mac:

1. Download and install VirtualBox.
2. Download and install Vagrant.
3. Make sure you have Git installed, and clone the following repository: *https://github.com/IBM-Swift/vagrant-ubuntu-swift-dev.git*.
4. Once you've cloned the repository, navigate into its directory: `cd vagrant-ubuntu-swift-dev`.
5. Run the command `vagrant up`.
6. Wait. The vagrantfile included in the repository you cloned, which tells Vagrant what to do, downloads Ubuntu 15.10, the Swift prerequisites, the prerequisites for `libdispatch`, the Swift concurrency library, the Sphinx documentation system, and then clones the Swift repository and creates a script that allows

you to build Swift. This might take a while and will download a few gigabytes of stuff.

7. Once Vagrant is done, you can run the following command to connect to the Linux installation: `vagrant ssh`.
8. Then, once in Linux, run the following script to build Swift: `/vagrant/swift-dev/swift/utils/build-script` (This might also take a while, depending on the speed/capabilities of your computer.)
9. You can then run the following command to verify Swift is up and running: `swift --version`.
10. You can then create some *.swift* files, and compile them with the `swiftc` command. We'll cover this in the next section, as well as in more depth later in the report.

Using Swift on Linux

Once you've got Swift installed, whether on a real Linux machine or within VirtualBox or similar, you can start using it! To confirm that you've got Swift installed properly, enter the following command in your terminal:

```
$ swift --version
```

If everything is up and running, you should be greeted with something resembling the following:

```
Swift version 3.0-dev (LLVM 834235, Clang 3434, Swift 354636)
Target: x86_64-unknown-linux-gnu
```

To test that your compiler is actually functioning, create a new file named *hello.swift*. Inside the file, add the following line:

```
print("Hello from Swift!")
```

Now, using your terminal, compile your Swift program by running the following command:

```
$ swiftc hello.swift
```

Ideally, you'll then have a compiled binary in the same folder, which you execute from your terminal as follows:

```
$ ./hello
```

You should be greeted by the output: `Hello from Swift!` Tada! Swift is working on Linux.

Of course, you can do more than just print output to the console (you'd hope so, wouldn't you?). One of the most interesting parts of the Swift open source project is Foundation framework, an open source implementation of the basic Foundation library that comes with Mac OS and iOS.

A Working Example

There's no better way to get a feel for using Swift on Linux to write actual programs than to write an actual program! In this section, we'll write a simple little Swift program, using the Swift Foundation libraries provided as part of the Swift 3 release.

If you want to work with Swift for Linux on your Mac, check back to "Swift on Linux" on page 22 to see how to set it up.

The program we'll write will read a text file of comma-separated data containing a date, a price, and a note. The data file will look like this:

```
2016-07-13,2.52,Bus ticket
2016-07-12,1.21,Coffee
2016-07-15,5.00,Orange Juice
```

And, using this data, it will print out an easier-to-read report, like this:

```
Wednesday, 13 July 2016: $2.52 for Bus ticket
---
Wednesday, 13 July 2016: $1.21 for Coffee
---
Wednesday, 13 July 2016: $5.00 for Orange Juice
```

This program makes use of Swift 3 and showcases a number of different elements changed through Swift evolution proposals, many of which we discussed earlier, including:

- Removed prefixes
- Changed function parameter labels
- Simplified parameters
- Enums have become `camelCased`

Let's get started!

First, we need to import the Foundation framework that we're going to use:

```
// Import the necessary classes from the Foundation framework
import Foundation
```

Now, we'll read the data file containing the comma-separated data:

```
// Get the data from a file called "Data.txt"
let data = try! String(contentsOfFile: "Data.txt")
```

You'll need to make sure you create a text file in the same folder as your Swift program for this to work!

Next, we'll split the data into different lines, filtering to remove any blank lines:

```
// Split into lines and remove blank lines
let rows = data.components(separatedBy: "\n")
    .filter({ $0.characters.count > 0 })
```

We need to be able to understand dates in order to print them nicely in the output, so we need two data formatters: one for the input and one for the output. Create the input date formatter:

```
// Create the first date formatter, for reading the date
let dateInputFormatter = DateFormatter()
dateInputFormatter.dateFormat = "dd-MM-YY"
```

Now, create the the output date formatter:

```
let dateOutputFormatter = DateFormatter()
dateOutputFormatter.dateStyle = .fullStyle
```

Next we'll create a formatter to format the currency:

```
// Create a number formatter for formatting the currency
let numberFormatter = NumberFormatter()
numberFormatter.numberStyle = .currency
```

And a place to store the output lines of the pretty report we're creating:

```
// The lines in our report will go in here
var reportLines : [String] = []
```

Finally, we'll iterate through the rows of data from the input file:

```
// Process each row
for row in rows {
```

splitting each row into columns, based on the position of the comma:

```
// Split the row into columns
let columns = row.components(separatedBy: ",")
```

and then extracting each piece of data we want, from each column of the row we're working with:

```
// Extract the data from each column
let dateColumn = columns[0]
let amountColumn = columns[1]
let noteColumn = columns[2]
```

We'll grab the currency amount, pulling a double variable out of the string we are working with:

```
// Read the price as a number
let scanner = Scanner(string: amountColumn)

var price : Double = 0

scanner.scanDouble(&price)
```

and format it as currency, using the currency formatter we created earlier:

```
// Format the number
let priceFormatted =
    numberFormatter.string(from:price) ?? "$0.00"
```

Also formatting the date, using the date output formatter we created earlier:

```
// Format the date
let dateFormatted = dateOutputFormatter.string(from: Date())
```

And then add a nice, pretty line to the report variable:

```
    // Add the line to the report
    reportLines.append(
        "\(dateFormatted): \(priceFormatted) for \(noteColumn)")

}
```

Last, to display our nice report, we'll create one big string, with each line separated by a new line and a series of dashes:

```
// Turn the report lines into a single string,
// separated by lines of '---'
let report = reportLines.joined(separator: "\n---\n")
```

and print the report:

```
// Finally, print the report
print(report)
```

To test this program, put this text in a file called *Data.txt*, and make sure it's in the same folder as your Swift file:

```
2016-07-13,2.52,Bus ticket
2016-07-12,1.21,Coffee
2016-07-15,5.00,Orange Juice
```

Then compile your Swift, like so:

```
swiftc SimpleDemo.swift
```

Then run the newly compiled program:

```
./SimpleDemo
```

If everything worked as intended, then you'll get this:

```
Wednesday, 13 July 2016: $2.52 for Bus ticket
---
Wednesday, 13 July 2016: $1.21 for Coffee
---
Wednesday, 13 July 2016: $5.00 for Orange Juice
```

This Swift code will work just fine on Mac OS, and likely even in Swift Playgrounds on an iPad.

Kitura: A Web Framework for Swift

IBM has been doing some amazing work with Swift, and one of the most interesting pieces that it's produced is the Kitura Swift web framework and HTTP server. Kitura features the basics you'd expect to find in a modern web framework, and…not much more…yet. It's got:

- URL routing, with GETs, POSTs, PUTs, and DELETEs
- URL parameters
- Static file serving
- JSON parsing

And really, that's about it so far. But it's a phenomenal start—it's very Swifty in approach, and everything feels like it should feel in a Swift web framework. Kitura supports Swift 3 and can be run on Mac OS and Linux (as well as in Docker or Vagrant, if that's your thing).

To install Kitura, follow the guides available on the project page for your preferred platform. The basics of Kitura should be familiar to you if you've used web frameworks on other platforms.

First, you import the Kitura framework and create a constant to store a router in:

```
import Kitura

let router = Router()
```

Then you sit up the router to response to requests, and display something when the root URL (/) is hit with a GET request:

```
router.get("/") {

    request, response, next in

    response.send("Hello from Kitura!")

    next()

}
```

Finally, you can start Kitura's built-in HTTP server and fire up the Kitura framework:

```
Kitura.addHTTPServer(onPort: 80, with: router)

Kitura.run()
```

To run your simple web app, you'd then need to compile it. (You hadn't forgotten that Swift is a compiled language, had you? It's easy to forget!) To compile it, you'll need to run some variant of the Swift `build` command on your terminal. On Mac OS, that's likely to be:

```
$ swift build
```

You'll then end up with a compiled binary that you can fire up, and then surf to the URL it's serving to be greeted by "Hello from Kitura!" Pretty nifty!

CHAPTER 5
Conclusion

We hope this tour of Swift 3 and the ecosystem around it has been useful for you! A good starting point for continued learning are the videos from Apple's WWDC conference—a lot of developer-focused things, beyond Swift, were announced at the last one:

- Mac OS, iOS, watchOS, and tvOS have all received huge updates, with a lot of new features.
- On the Apple Watch, fitness apps can run in the background during workouts, and the SpriteKit, SceneKit, Game Center, and CloudKit APIs are now available.
- On the Apple TV, ReplayKit, PhotoKit, and HomeKit APIs are now available.
- On iOS, apps can now make better use of the MapKit, iMessage, HomeKit, and Siri.
- On Mac OS, a complementary set of features have been added that keep Mac OS in line with iOS's abilities.

If you're itching to learn how to build apps, we're quite proud of our own books: *Learning Swift* and *Swift Development for the Apple Watch* (both O'Reilly), which are up-to-date with the current public release of Swift (version 2.x). *Learning Swift* teaches you Swift, as well as the Cocoa, CocoaTouch, and watchOS frameworks, for building apps for OS X (now macOS), iOS, and watchOS respectively. *Swift Development for the Apple Watch* teaches you how to use the various watchOS frameworks to build an app for Apple Watch.

While both of these books target Swift 2.x, not Swift 3, the syntax changes between the two languages are minimal, and Swift 2.x is the only version of Swift you can submit apps to the store with until late in 2016, when Swift 3 comes out. Additionally, Xcode 8, when it becomes public, will assist you in migrating your code from Swift 2.2 to Swift 3.

We also highly recommend our friend Tony Gray's *Swift Pocket Reference*, which is available as a free ebook from O'Reilly Media.

If you'd prefer to go straight to the source, Apple also offers an eBook, *The Swift Programming Language*: it's available from the iBooks store.

About the Authors

Jon Manning and **Paris Buttfield-Addison** are co-founders of the game and app development studio Secret Lab. They're based on the side of a mountain in Hobart, Tasmania, Australia.

Through Secret Lab, they've worked on award-winning apps of all sorts, ranging from iPad games for children to instant-messaging clients to math games about frogs. Together they've written numerous books on game development, iOS software development, and Mac software development. Secret Lab can be found online and on Twitter at @thesecretlab.

Jon frequently finds himself gesticulating wildly in front of classes full of eager-to-learn iOS developers. Jon used to be the world's biggest Horse ebooks fan, but has since come to accept their betrayal. Jon writes so much code you wouldn't believe it, has a PhD in Computing, and can be found on Twitter at @desplesda.

Paris has coded for everything from 6502 assembly to Qt to iOS, and still thinks digital watches are a pretty neat idea. Paris speaks constantly at conferences and enjoys the company of greyhounds and whippets. He has a PhD in Human-Computer Interaction. He can be found on Twitter as @parisba.

Tim Nugent is a mobile software engineer, game designer, and recently submitted a PhD in Computing. He writes books for O'Reilly Media, and can be found online at *http://lonely.coffee*.