Swift: a reactionary language?

Kevin Roark Jr.

I’m going to be talking about Apple a lot.

I just want to make it clear that I am *not* obsessed with Apple. I think it is a somewhat-questionable corporation that can make very good things but that can also have a bad attitude and bad habits. It should not be lauded as something beyond human or idolized.

However, take this with a grain of salt, as I have an iPhone, a MacBook, and really enjoy mobile iOS development; all three are real nice.

The Incremental evolution of Objective-C

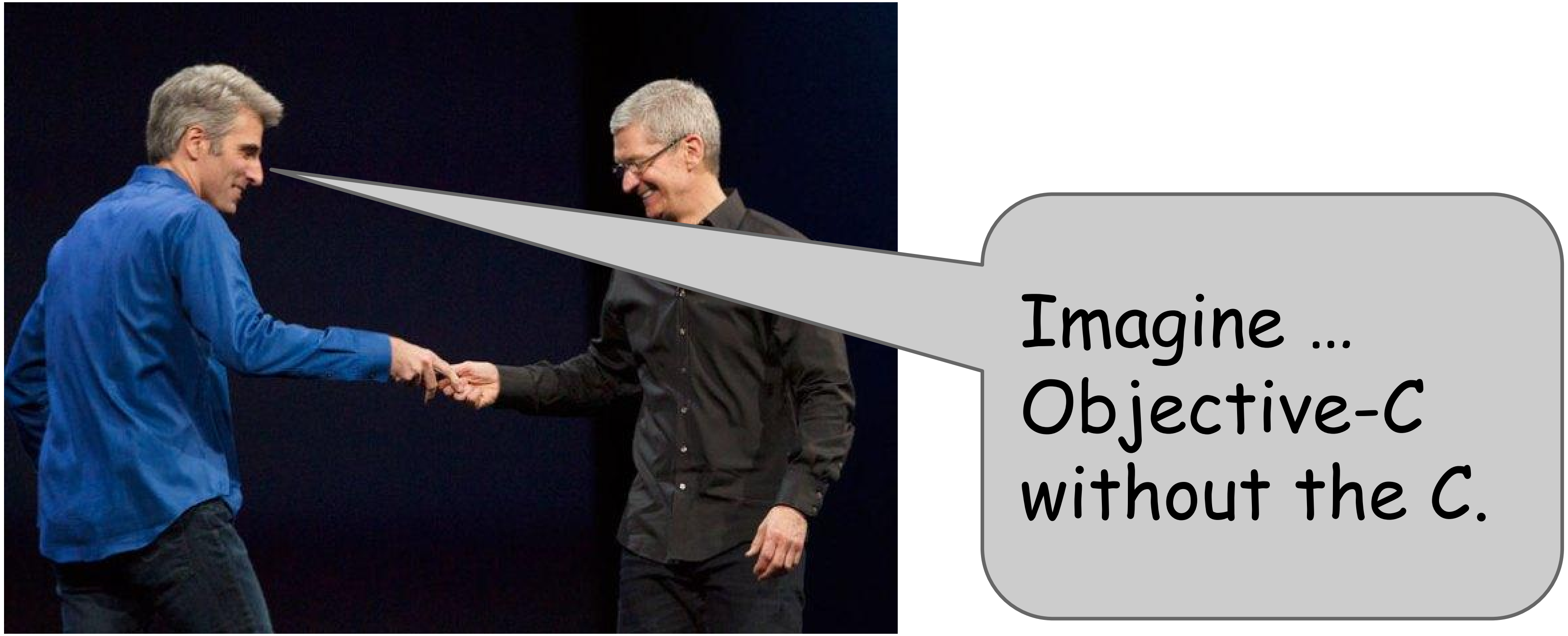
things have been added to the language over the years
(not that I really have been around to see it …)

- subscripting
- block-syntax
- ARC
- dot-syntax
- the move away from “id”
- literals
- and so on

… but eventually there’s a tipping point

Some things you can’t fix incrementally. There are flaws in Objective-C that would have never been overcome. Horrible, horrible flaws. Sometimes you have to …

BREAK BACKWARDS COMPATIBILITY AND START FRESH!!!



Quick Note on last week's presentation

Java 8's lambdas are an interesting example of incremental evolution. Their implementation is weird, but potentially could work. That can't happen forever though … in some ways I think of Swift as akin to scala -> a brand new language with the runtime of something old.

I built some hype. Here’s a quick overview of Objective-C’s worst parts.

- it is too dynamic. you can make the compiler do anything. you will get a lot of runtime crashes.
- it is a strict superset of c. it is thus syntactically limited by a very old language.
- “faking” of properties, enums, and quite a bit more
- verbose to the point of absurdity
- header files
- you can call “malloc” ...
- no visibility control
- accidental overrides
- mutable and immutable copy …
- behavior of nil
- …. on and on for a long time

great little programming blog: http://nearthespeedoflight.com/

BUT Objective-C really is not so bad and has a lot of good ideas

- robust libraries
- multi-threading is nice (GCD)
- categories and extendibility
- readability
- range of communication patterns
- … this list also actually goes on too.
- applications on the personal computers and phones we use every day for the most part run on Objective-C and are sometimes pretty cool … so it can’t be the worst.

Language language language

Language Influences and Features

Features features features

Influences from other languages

- **Javascript** -> closure syntax, func keyword, identity operators, var, etc.
- **Python** -> ranges, object-oriented & functional elements, inferred static typing
- **Ruby** -> string interpolation, optional binding, implicit returns
- **Java** -> generics are quite similar, the extending a “rock solid runtime” idea
- **Rust** -> null is much less of a concept, safety and correctness are heralded
- **Objective-C** -> Cocoa, LLVM, emphasis on enumerations, ARC, categories, XCode is actually great.
- Clearly a wide net was cast

Speaking of programming languages as vehicles

Interesting Language Features

A quick note: I clearly don’t have time to dive into everything, and this is not going to be a full picture of Swift.

I *will* try to focus on things that I think are interesting. I will also try to go quickly.

Small, but really excited about this one.

- “=” is no longer an expression
- think about it
- no more unnecessary “=” vs “==” bugs
- why did language designers ever think this was a good idea

Variable and Function Declarations: a Hodgepodge

```
var x : Int = 25
var y : String?
var z : [Double] = [2.0, 3.0]
```

```
func addOne(x: Int?) -> Int? {
        if x != nil {
                return x + 1;
        } else {
                return nil;
        }
}
```

```
func funct(externalName localName: String) { … }
```

```
private func ...
```

```
func funct(#externalName: String) { … }
```

```
func funct(externalName: String = “Swift”) { … }
```

Type inference

small but nice and safe way to cut down on redundancy in code. Expressive Static Typing.

var x = “hi!!!!” // this is totally encouraged, if a variable is being assigned a value at initialization.

The function type of a closure can be inferred as well.

Immutability baked into declarations

let x = 2; // think of x like a #define constant, or a final variable in java. this provides safety and allows for compiler optimizations

var x = 2; // capable of change

The Mutable and Immutable variants of containers in Objective-C are no longer necessary.

@final func cool() -> String { return “cool” } // can’t be overridden

The inout parameter keyword

I thought this was a very neat idea that is certainly more explicit than doing it with pointers in C

```
func swapValues<T>(inout a: T,
inout b: T) {
    let tempA = a
    a = b
    b = tempA
}
```

Switch statement overhaul

- switch case on any fundamental types, and enums
- “break” is implied, aka fallthrough is not the default behavior!!
- mandatory exhaustiveness
- ranges in a switch
- ** and more ** -> look how fun

```
switch i {
        case 0: println(“none”)
        case 1...3: println(“a lil”)
        case 4..<7: println(“some”)
        case _ where i >= 7: println(“a lot”)
}
```

```
switch name {
        case “kev”: println(“hi”)
        case “al”: println(“hello”)
        default: break // check it out
}
```

Strings

Strings *seem* like they should be easy to get right. In many ways programming often comes down to string manipulation. In Swift they are a fundamental pass-by-value and immutable type.

In Objective-C they really are not good.

NSMutableString string = [NSMutableString alloc] initWithString:@”Hello”];

[string appendString:@” world”];

NSString greeting = [NSString stringByAppendingString:@”my name is Kevin”];

Swift looks script-like and fun in comparison (but pretty standard compared to other new languages)

var string = ”Hello” + “ world”;

var greeting = string + “my name is Kevin”;

Optional Types

Objective-C:

```
NSString x = @”hello”; // great
NSString y = nil; // fine
```

Swift

```
var x : String = “hello”; // great
var x : String? = nil; // ok
var x : String = nil; // COMPILER ERROR
```

breaks the paradigm / idea of every pointer being to x or to nil. An explicit annotation of allowing nothingness is required. Complexity is reduced. The compiler *can check everything*.

Unwrapping an optional

```
var potentialTopic: String? = “swift”
if potentialTopic != nil {
  println(potentialTopic!)
}
```

Optional binding is short and sweet and neat

```
var potentialTopic: String? = “swift”
if let topic = potentialTopic.uppercase() {
    println(topic)
} else {
    println(“there never was a topic”)
}
```

Optional Chaining

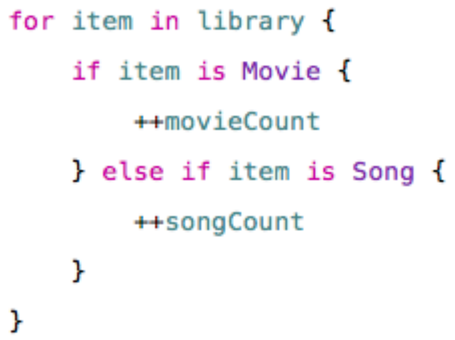
or, the options continue (***killer pun***)

*let count = object.items!.count // **ERROR***

vs.

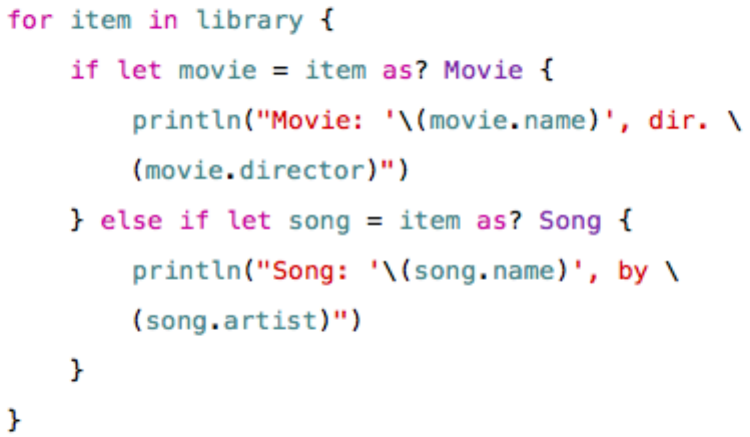
*let count = object.items?.count // **count is implied to be an optional type and will contain nothing in this case***

Type Casting



```
for item in library {
    if item is Movie {
        ++movieCount
    } else if item is Song {
        ++songCount
    }
}
```

The **is** operator checks the type of an object.



```
for item in library {
    if let movie = item as? Movie {
        println("Movie: '\(movie.name)', dir. \
        (movie.director)")
    } else if let song = item as? Song {
        println("Song: '\(song.name)', by \
        (song.artist)")
    }
}
```

if item is Movie {
  let movie = item as Movie
  movie.play()
}

The **as** operator downcasts the object into a specific subclass. It returns a non-optional type and can lead to runtime errors.

The **optional as** operator is obviously cool.

I think all of this casting syntax in general is much nicer than what you see in C and its derivatives

Look at this crazy and useful thing you can do

```
for thing in things {
    switch thing {
    case 0 as Int:
        println("zero as an Int")
    case 0 as Double:
        println("zero as a Double")
    case let someInt as Int:
        println("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        println("a positive double value of \(someDouble)")
    case is Double:
        println("some other double value that I don't want to
        print")
    case let someString as String:
        println("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        println("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        println("a movie called '\(movie.name)', dir. \
        (movie.director)")
    default:
        println("something else")
    }
}
```

told y'all switch-case was overhauled and is going to be used all the time

Multiple Return Types, aka Tuples

```
NSError *error;
NSData *data = [NSData dataWithContentsOfFile(file, &error);

let (data, error) = Data.dataWithContentsOfFile(file)
```

This is just an example, but it reduces the need for pointer-based in-out parameters and for *stupid* small structs or classes.

The worst part of Objective-C is its type system.

Swift introduces generics. Many things are better. Take a look.

Here’s an example

NSArray can contain pointers to objects of any type (and not any primitives). You can keep track of what is inside and be reasonably sure based on an API, but come on … (also, why have an array that can’t have integers in it?)

Here’s what can be bad:

```
NSArray *arr = @[@”cool”, [UIButton new], [NSNumber numberWithInt:2]];
for (NSDictionary *dict in arr) {
        [dict setObject:nil forKey:@”ok”]; // this breaks for so many reasons
}
```

Generics Fix some problems

- we can know specifically what is inside of a collection.
- functions can be written to support nice “polymorphic” behavior, while having guarantees of what the parameters they support can do.
- no more “object 0xas234a2323b does not respond to selector ‘count’ …”

Where clauses & generics syntax

```
func containersAreEquivalent<C1: Container, C2: Container
where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
(someContainer: C1, anotherContainer: C2) -> Bool {
    // body of the function
}
```

taken directly from the Swift Reference Book

Allows very detailed specification of the constraints on a generic function's potential types

Collections are much better (than in Objective-C

- they are typed (using generics)
- they can contain objects or non-objects
- they can contain optional values
- these all seem like no-brainers, but you probably haven’t programmed in Objective-C

Powerful structs

small model classes aren’t actually so bad, like I alluded to earlier. It’s just that they are a big pain to write in Objective-C.

```
@interface MyClass
        @property (nonatomic, strong) NSString *name;
        @property (nonatomic, assign) NSInteger age;
        - (instancetype)initWithName:(NSString *)name age:(NSInteger)age;
@end
@implementation MyClass                // not writing this implementation is a runtime crash. I’ve done it.
        -(instancetype)initWithName:(NSString *)name age:(NSInteger)age { self = [super init]; if (self) { _name = name; _age = age; } return
self; }
@end
```

Structs are another answer to that problem. Structs are first-class types very similar to objects, but that are always bassed by value. Swift encourages creation of many small and useful models.

```
struct MyStruct {
        let name: String
        let age: Int
};
```

Closures are really nice

*Objective-C's block syntax is *not*.*

if you will pardon my language, there is a very nice website called fuckingblocksyntax.com that Objective-C developers have to use every time they need to use a block. Let's go there.

A progression of valid sorting closures in Swift

1.

```
sorted(objects, {
(s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Type inference

sorted(objects, { s1, s2 in return s1 > s2 })

2.

Implicit returns in single-line closures

sorted(objects, { s1, s2 in s1 > s2 })

3.

Shorthand argument names

sorted(objects, { $0 > $1 })

4.

And a very specific case …

sorted(objects, >)

5.

Feature-rich Enumerations

- think “abstract data type” style
- can have behaviors and methods
- do not need to have associated “raw” values
- first-class types that lend themselves really well to a lot of things. I think enumerations will be very popular “idiomatic” in Swift.

Objects and Properties

computed properties /
getter and setters

```swift
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
```

```swift
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            println("About to set totalSteps to \
        (newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                println("Added \(totalSteps -
        oldValue) steps")
            }
        }
    }
}
```

property observers

examples from the Swift Programming Language Book

Extensions

- an evolution of Objective-C’s categories
- allow you to extend the functionality of an existing type in a separate definition
- makes sense for adding protocol conformance, edge-case functions that perhaps don’t belong in the main implementation
- also is really a great thing for extending classes whose source code you do not have access to (a la libraries)

Operator Madness

- operator implementation for classes by naming a function “+” with the correct signature

- can also make custom operators named anything with these symbols: “/ = - + * % < > ! & | ^ . ~”

Rare feature: literal convertibles

```
class Dog { var name … }
var dog : Dog = “sally” // obviously an error

class Cat : StringLiteralConvertible {
    var name
    class func convertFromStringLiteral(value: StringLiteralType) -> Cat
    { return Cat(value) }
}
var cat : Cat = “iceman” // works !!!
```

note the “class” modifier

C O M P I L E R

Notes on how swift works

R U N T I M E

Chris Lattner seems smart

- Went to grad school at University of Illinois and worked on LLVM, now works at Apple as the lead of developer tools
- big part of ARC
- primary author of LLVM
- and a big part of clang
- …
- worked alone on swift secretly for ~1 year starting in 2010.

The Swift Model - “The Minimal Model”

- Statically compiled
- Small runtime
- Flexible, Predictable, Efficient
- Transparent interaction with C and Objective-C
- Easily deployed to previous versions of iOS and OS X
- No non-deterministic JIT or GC pauses
- Native code with no artificial abstraction enabling bare-to-the-metal programming

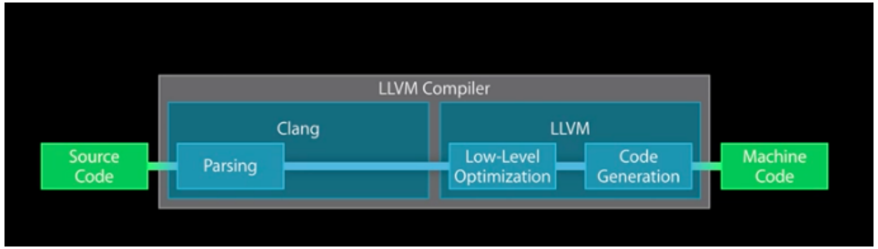
`From the “Advanced Swift” session of WWDC ‘14`

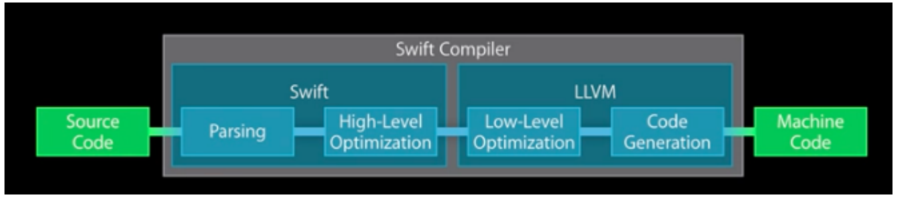
Swift’s compiler

- not hard to guess that Swift uses LLVM
- what is potentially more interesting is that it also uses a modified Clang, which is traditionally a c-language front-end
- Swift is in many ways an abstraction and optimization on top of the Objective-C runtime, making the bridge between them and utilization of Apple’s existing tools relatively smooth
- “new language, established tools” is a consistent pattern we’re seeing

Compiler Architecture

The only difference from the standard LLVM flow is an additional high-level optimization step allowing language-specific analyses.





`From the "Advanced Swift" session of WWDC '14`

These language-specific optimizations

- Removing abstraction penalties
  - Performs global analysis of app. Structs shouldn't hurt. Internally even the lowest level types like Ints and Floats are written as struct wrappers around native LLVM types. So struct zero-abstraction-cost is essentially guaranteed
- Generic specialization
  - Rather than constructing all called versions of generic functions, generic specialization allows compiler to run generic code directly.
- Devirtualization
  - Resolving dynamic method calls at compile-time.
- ARC optimization, Enum analysis, Alias analysis, Value propagation, Library optimizations on Strings and Arrays, etc.

`From the "Advanced Swift" session of WWDC '14`

The Swift <-> Objective-C bridge

Apple couldn’t afford to discard its rich Cocoa history and expect its developers to follow.

As a result, anything** written in Objective-C can be used from Swift, and visa-versa (the signatures of methods are mangled to look good both ways, it’s quite cool). This how Apple compromised: a fundamentally new language that somehow doesn’t break compatibility.

Swift is a break in paradigm from Objective-C in terms of philosophy, but not programming legacy.

** really, most things. Anything with an advanced Swift feature (like tuples) that can’t readily be expressed in Objective-C. To guarantee that a piece of code is accessible in Objective-C, use the @objc attribute to have the compiler check for you.

What looks like direct frontend translation goes on behind the scenes

In Objective-C, a UITableView looks like this:

*UITableView *tableView = [[UITableView alloc] initWithFrame: CGRectZero style:UITableViewStyleGrouped];*
[tableView insertSubview:subview atIndex:2]

In Swift:

var tableView = UITableView(CGRectZero, .Grouped)
tableView.insertSubview(subview, atIndex: 2)

more details in the Apple “Swift Interoperability In Depth” video

This happens automatically, for any piece of code in either language from the other, so long as you import the necessary files.

Of course the actual implementation is more complex, but in the most part it involves a single relatively efficient copy (especially efficient for immutable types)

Secret:

- Swift objects *are* Objective-C objects under the hood.
  - same layout as an Objective-C class
  - the bridge from a swift object to an Objective-C object is essentially free
- subclassing any Objective-C object from Swift prevents the static swift optimizations and makes the class fully accessible from Objective-C

Performance differences

benchmarks from http://www.jessesquires.com/apples-to-apples/

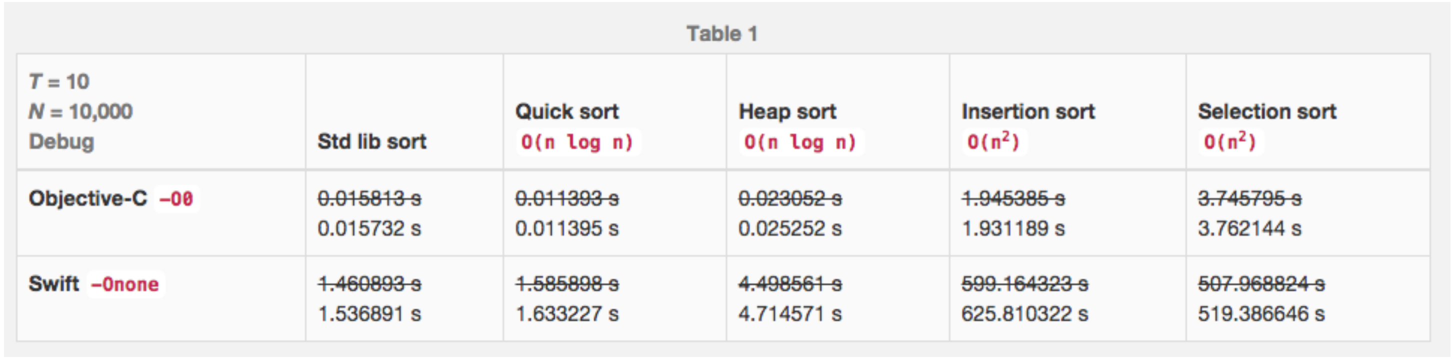


Table 1

T = 10 *N* = 10,000 Debug	Std lib sort	Quick sort `O(n log n)`	Heap sort `O(n log n)`	Insertion sort `O(n^2)`	Selection sort `O(n^2)`
Objective-C `-O0`	~~0.015813 s~~ 0.015732 s	~~0.011393 s~~ 0.011395 s	~~0.023052 s~~ 0.025252 s	~~1.945385 s~~ 1.931189 s	~~3.745795 s~~ 3.762144 s
Swift `-Onone`	~~1.460893 s~~ 1.536891 s	~~1.585898 s~~ 1.633227 s	~~4.498561 s~~ 4.714571 s	~~599.164323 s~~ 625.810322 s	~~507.968824 s~~ 519.386646 s

CRAIG LIED?

After Swift Beta 3

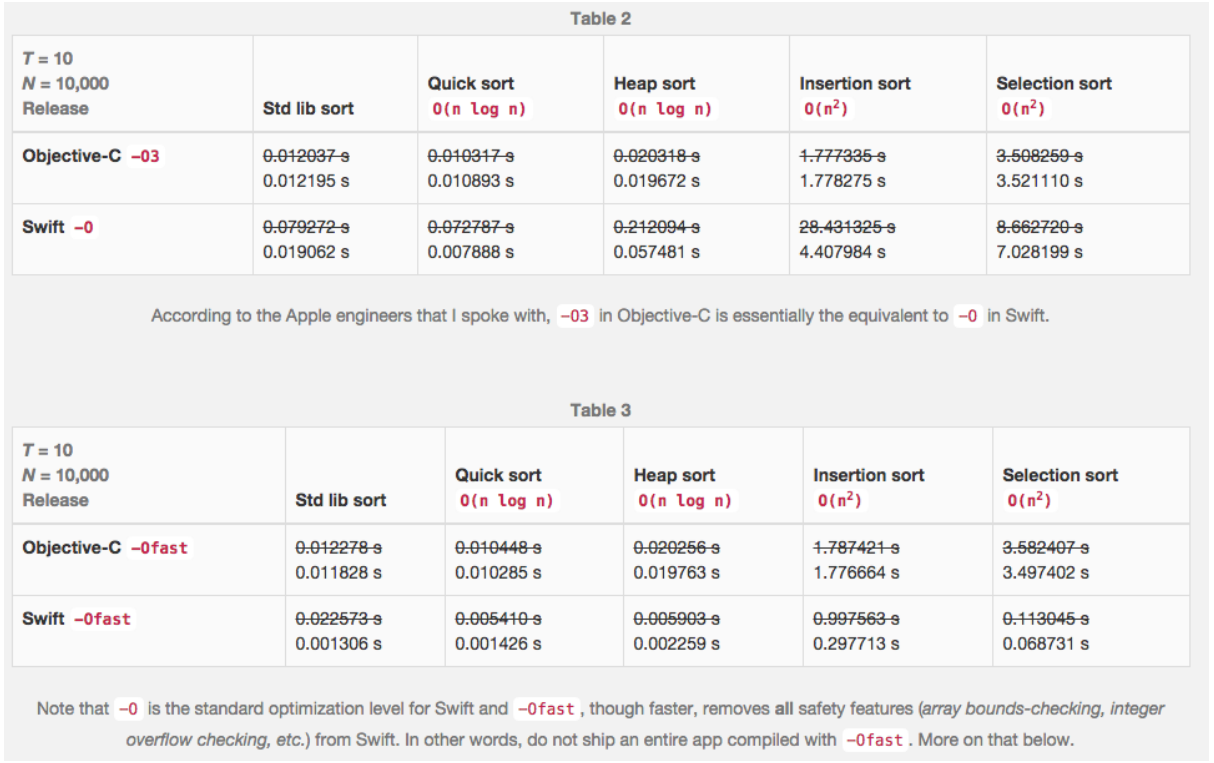


Table 2

T = 10 N = 10,000 Release	Std lib sort	Quick sort O(n log n)	Heap sort O(n log n)	Insertion sort $O(n^2)$	Selection sort $O(n^2)$
Objective-C `-O3`	~~0.012037 s~~ 0.012195 s	~~0.010317 s~~ 0.010893 s	~~0.020318 s~~ 0.019672 s	~~1.777335 s~~ 1.778275 s	~~3.508259 s~~ 3.521110 s
Swift `-O`	~~0.079272 s~~ 0.019062 s	~~0.072787 s~~ 0.007888 s	~~0.212094 s~~ 0.057481 s	~~28.431325 s~~ 4.407984 s	~~8.662720 s~~ 7.028199 s

According to the Apple engineers that I spoke with, `-O3` in Objective-C is essentially the equivalent to `-O` in Swift.

Table 3

T = 10 N = 10,000 Release	Std lib sort	Quick sort O(n log n)	Heap sort O(n log n)	Insertion sort $O(n^2)$	Selection sort $O(n^2)$
Objective-C `-Ofast`	~~0.012278 s~~ 0.011828 s	~~0.010448 s~~ 0.010285 s	~~0.020256 s~~ 0.019763 s	~~1.787421 s~~ 1.776664 s	~~3.582407 s~~ 3.497402 s
Swift `-Ofast`	~~0.022573 s~~ 0.001306 s	~~0.005410 s~~ 0.001426 s	~~0.005903 s~~ 0.002259 s	~~0.997563 s~~ 0.297713 s	~~0.113045 s~~ 0.068731 s

Note that `-O` is the standard optimization level for Swift and `-Ofast`, though faster, removes **all** safety features (*array bounds-checking, integer overflow checking, etc.*) from Swift. In other words, do not ship an entire app compiled with `-Ofast`. More on that below.

speed vs. safety?

And then Beta 5

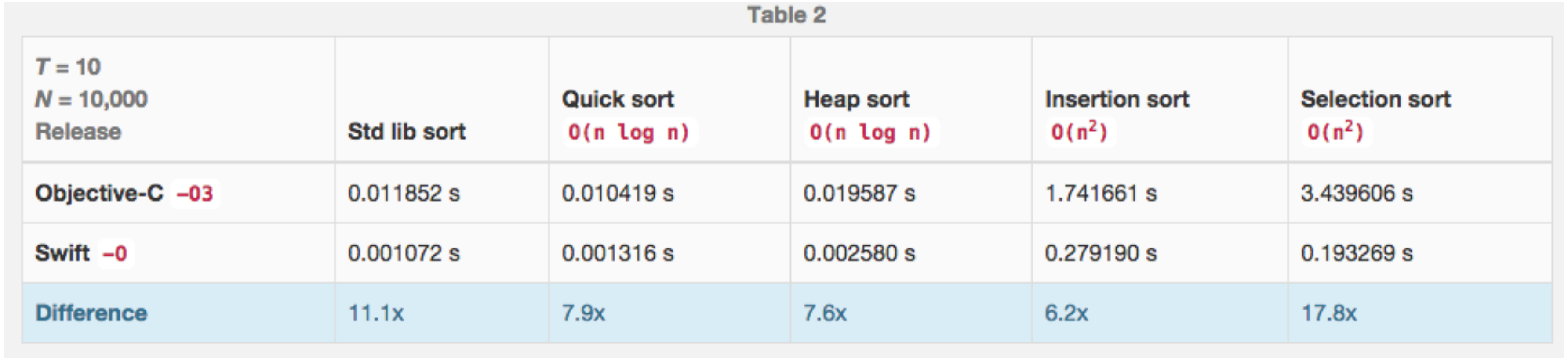


Table 2

T = 10 *N* = 10,000 Release	Std lib sort	Quick sort `O(n log n)`	Heap sort `O(n log n)`	Insertion sort `O(n²)`	Selection sort `O(n²)`
Objective-C `-O3`	0.011852 s	0.010419 s	0.019587 s	1.741661 s	3.439606 s
Swift `-O`	0.001072 s	0.001316 s	0.002580 s	0.279190 s	0.193269 s
Difference	11.1x	7.9x	7.6x	6.2x	17.8x

P.S. the standard library sort even outperforms raw C

Why is it faster?

- Objective-C replaces every call with a weird dynamic method dispatch
- stricter type rules allow the compiler to optimize method lookups
- smart compiler knows when objects can skip the heap
- more efficient register usage (no _cmd parameter, for one)
- no “aliasing” (multiple pointers per memory chunk, think *restrict* keyword)
- the more rigid structure generally allows for more specific optimizations, given a really smart compiler (which we have already seen).

devil’s advocate:

does this speed come at a cost of flexibility?

is this speed necessary for GUI applications?

`https://mikeash.com/pyblog/friday-qa-2014-07-04-secrets-of-swifts-speed.html`

Playgrounds

- making incremental changes for large projects in compiled languages wastes a lot of time.
- Playgrounds act almost like a python or node.js REPL, but for fancy GUI apps
- there is also a traditional REPL for non-interactive code
- something the developers of Swift are **very** proud of

p e r f o r m a n c e

Language Design Constraints

e x p r e s s i v e n e s s

Performance

- has to have *buttery scroll* on an iPhone (and an Apple watch…)
- C-based culture of performance baked into the company and community
- will be compared to Objective-C
- strength of a systems programming language

Interface With Existing Infrastructure

- Cocoa and everything developers expect from last 30 years. It doesn’t make sense to start anew.
- Apple’s existing suite of compilation tools
- Apple’s culture (it’s hard to change, the cult, etc.)

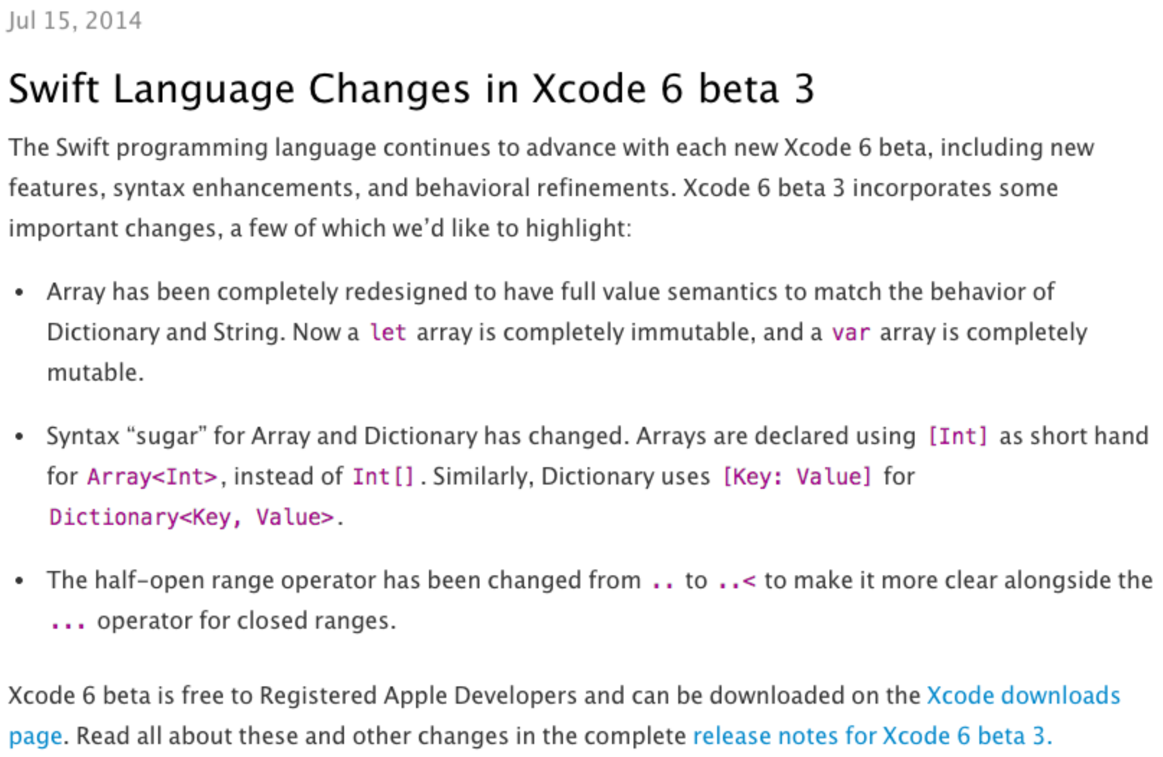
Responding to Objective-C criticisms

- type safety
- expressiveness (*clearly one of the most important features*)
- correctness
- safety
- irregular and obtuse syntax
- incorporation of aspects from modern programming language theory

The developers of Swift see it, in many ways, as a protocol-based language. They imagine a common pattern of using extensions of a class to conform to specific protocols (composition pattern). I think this idea is good and in interesting way to approach a language.

`Idea stated in the “Advanced Swift” session of WWDC ‘14`

Language evolution



Jul 15, 2014

Swift Language Changes in Xcode 6 beta 3

The Swift programming language continues to advance with each new Xcode 6 beta, including new features, syntax enhancements, and behavioral refinements. Xcode 6 beta 3 incorporates some important changes, a few of which we'd like to highlight:

- Array has been completely redesigned to have full value semantics to match the behavior of Dictionary and String. Now a `let` array is completely immutable, and a `var` array is completely mutable.
- Syntax "sugar" for Array and Dictionary has changed. Arrays are declared using `[Int]` as short hand for `Array<Int>`, instead of `Int[]`. Similarly, Dictionary uses `[Key: Value]` for `Dictionary<Key, Value>`.
- The half-open range operator has been changed from `..` to `..<` to make it more clear alongside the `...` operator for closed ranges.

Xcode 6 beta is free to Registered Apple Developers and can be downloaded on the Xcode downloads page. Read all about these and other changes in the complete release notes for Xcode 6 beta 3.

each swift binary contains its own small version of the runtime so that apps written in swift beta will always run

REALLY BAD PARTS

Swift's Bad Parts

THINGS ARE SOMETIMES BAD

Things lost from Objective-C

- the dynamic nature of Objective-C can be beneficial at times. Sometimes you can be smarter than a compiler, I take it.
- Instantiating classes by name can be helpful
- so can message passing
- intentional swizzling

SWIFT IS WALLED INSIDE OF APPLE

Optionals are great ...

but they do not *fully solve* the null-reference problem.

The Cocoa libraries are built with the expectation that *nil* is used to signify *nothing*. Swift allows implicit optional unwrapping, which saves a lot of time if you believe that an optional *will* have a value. But it makes things unsafe. The safe thing to do is verbose, but unsafe thing is easier and still exists.

In a similar vein …

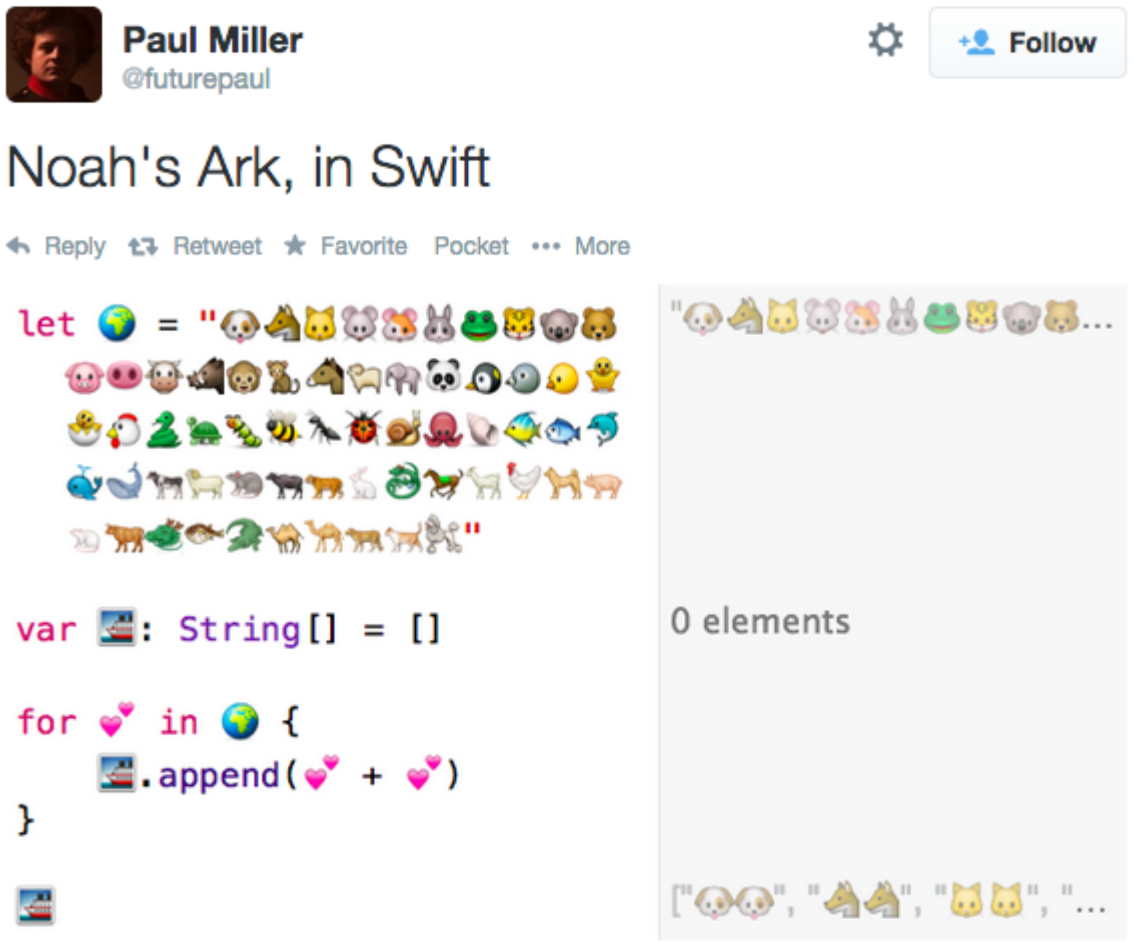
Immutable declarations with *let* are a great idea, but the path-to-least-resistance is to use *var* for everything. Additionally, value semantics with collections and mutability *can be* a whole separate mess.

In short, The mutability problem is also not solved.

Multi-paradigm languages can always have the problem of redundancy. Swift has been said to suffer from having its cake and eating it too.

Controversial features

I’m just going to say “operator overloading” and “custom operators” and “emojis are valid in identifier names”.



Paul Miller
@futurepaul

Follow

Noah's Ark, in Swift

Reply Retweet Favorite Pocket More

```
let 🌍 = "🐶🐴🐱🐭🐹🐰🐸🐯🐨🐻
    🐷🐽🐮🐗🐵🐒🐴🐑🐘🐼🐧🐦🐤🐥
    🐣🐔🐍🐢🐛🐝🐜🐞🐌🐙🐚🐠🐟🐬
    🐳🐋🐄🐏🐀🐃🐅🐇🐉🐎🐐🐓🐕🐖
    🐁🐂🐲🐡🐊🐪🐫🐆🐈🐩"

var 🚢: String[] = []

for 💕 in 🌍 {
    🚢.append(💕 + 💕)
}

🚢
```

```
"🐶🐴🐱🐭🐹🐰🐸🐯🐨🐻...

0 elements

["🐶🐶", "🐴🐴", "🐱🐱", "...
```

T H E F U T U R E

Notes on Swift's Future

OF SWIFT

Swift Reached 1.0

On september 9th. What does that mean?

Swift is like Rust or Go (sort of), except it is going to be used immediately by a lot of people. That’s where the whole “closed ecosystem” and “single option for developer tool” thing comes in.

Takeaway

Swift is clearly a **revelation** for Objective-C developers with some interesting new features.

However, Swift is clearly not perfect, and constrained from being as “progressive” as Rust or Haskell by things like Cocoa compatibility, performance concerns, and conflicting aspirations.

Would it be used if Apple didn’t ask for / force it? Hard to say.

But for the most part, thing is good.