The Tao of Swift

Adrian Kosmaczewski

2014-09-09

I gave this talk in September 2014, in Zürich, the same day Apple announced the Apple Watch.

- Introduction
- Warning
- The beauty
- Reason
- Who made it
- Interoperability
- Features
- Instantiation
- Type inference
- Constants
- Functions, Methods and Closures
- Optional Types
- Unicode
- Structs, Enums and Classes
- Protocols, Extensions and Generics
- Custom Operators
- Interoperability
- A Sample Application Using WebKit
- Many things I have not talked about

Tonight is a night that is really interesting for Swiss developers. First we are going to learn about Swift, whose name is similar to that of the "Society for Worldwide Interbank Financial Telecommunications" which is a very well known institution in Switzerland. Anyone working in a bank? And then we're going to laugh at Apple trying to sell a watch that crashes and which runs with batteries that last only for 6 hours. So, what we need is a bit of cheese and to dress in military costumes, and this would be a truly 100% Swiss evening.

Who has been using Swift in the audience? Well, this is an introductory talk, so there might be some things that you already know. Feel free to correct me if I say something wrong during my presentation, as we have all been exposed to the same information in the same amount of time. I hope I can, however, bring

something new to your knowledge.

Introduction

Let me quote the Tao of Programming, to bring back some context:

> "Thus spake the Master Programmer: The Tao of Programming flows far away and returns on the wind of morning. The Tao gave birth to machine language. Machine language gave birth to the assembler. The assembler gave birth to the compiler. Now there are ten thousand languages. Each language has its purpose, however humble. Each language expresses the Yin and Yang of software. Each language has its place within the Tao. But do not program in COBOL if you can avoid it."

If the world followed the Tao, Apple would make the hardware, Microsoft would make the developer tools, and everything would run under QNX. Alas, QNX has been bought by BlackBerry, Microsoft makes operating systems and Apple makes developer tools.

Welcome to our world. A world where the Tao is not being followed closely, where we have lost touch with the essential stuff.

Let me start this presentation by introducing myself. I am Adrian Kosmaczewski, and my two primary functions in this world is to write and to talk. I write code and books, and I talk a lot. I also come from a place far, far away.

Now let me introduce you to Xcode 6. This is the tool, currently in its seventh beta iteration, that allows you to crash very often. Whenever you need to crash, Xcode will be there to help you do that quickly and efficiently. The funniest thing about Xcode is that after it crashes it displays a dialog box that says "Xcode quit unexpectedly" which as everybody knows, it's just not true at all. We all know Xcode is going to crash. The question is "when."

Xcode 6 includes a gazillion new features, but there's one I will use tonight extensively: Playgrounds. These are like REPLs (Read-Eval-Play-Loop) but with all the crashing power of Xcode and none of the awesome autocompletion, refactoring, embedded help and other usability and productivity features of Xcode, which are actually very few.

Playgrounds give you very weird error messages, and then as soon as you try to help it, it crashes. So, there you go. Of course, this makes people very angry.

Warning

The same applies now to Swift. As much as Swift is a beautiful language, and we are going to see that closely tonight, I have to be blunt and clear with you right away: Swift is not ready for prime time. The current version of Swift still lacks many important features, for example there are no class variables as

of now; and even worse, there is a project in GitHub that clearly states many known valid language constructions (at least valid following the specifications already available) which make the compiler crash.

So, before I begin this presentation, a warning: do not use Swift in production applications right now. But, should you spend time learning it? Yes, indeed, I think it is a good idea. I actually try to learn a new programming language every year. But keep in mind that everything you are going to learn might change, and that if you write production code with it right now, you will have to rewrite it to comply with future changes in the language.

And if you do not believe me, here's Erica Sadun and Rob Napier saying the same. So, you've been warned.

The beauty

However unstable and incomplete, Swift is a beautiful language to write applications in. It is as simple and approachable as Ruby and Python, it has complex features borrowed from more complex languages like Haskell and C#, and it compiles into native code, which is executed without intermediaries, and some compiler errors and warnings are very similar to those from C . So, yes, the language borrows features from everywhere.

Apple says that Swift is modern, safe, fast and interoperable.

"Swift is modern, because it has closures; tuples and multiple return values; generics; fast and concise iteration over a range or collection; structs that support methods, extensions, protocols; and functional programming patterns, e.g.: map and filter.

Swift is safe, because variables are always initialised before use, arrays and integers are checked for overflow, and memory is managed automatically.

Swift is fast, because using the incredibly high-performance LLVM compiler, Swift code is transformed into optimised native code, tuned to get the most out of modern Mac, iPhone, and iPad hardware.

Swift is interoperable, and can be used together with Objective-C in the same project."

Sounds too good to be true, huh? Yeah.

Finally, Swift has a beautiful name that make Google requests extremely hard. This is because Swift is not only this language, but also:

- A bird
- A fox
- A singer
- A car
- A protocol for interbank communication
- And another, completely unrelated, programming language

Although, to be fair, in terms of naming programming languages, we've seen worse.

Reason

Why Swift? Well there are many reasons; to begin with we can all agree that Objective-C as a language felt outdated. It's basically a good old sitcom from the 80's that reruns every day on the screen of our laptops. Meanwhile, lots of different languages have captured the imagination of developers everywhere, so we can say that we needed to change.

Another reason is that, well; many developers simply disliked the whole idea of a language with pointers and weird square brackets all over the place. But that's an opinion, and of course the Interwebz is filled with them.

Who made it

Swift was created by Chris Lattner and his team at Apple, who worked on the language as a secret project since 2010. The announcement of the language came as a surprise during the latest WWDC keynote in June, so everything that we know about the language has been discussed in the past 3 months.

To bring a bit of perspective, Chris Lattner is the chief developer of the LLVM project, which brought as many nice things, such as the CLang Static Analyzer and also, of course, Automatic Reference Counting (ARC.)

The language jumped to the 16th place in the TIOBE ranking in July, then dropped to the 22nd in August and right now it is in the 18th place. Given the amount of interest given to the language online, it is clearly becoming a very interesting topic on the internet.

Interoperability

Swift has been thought from the ground up as a transitional language from Objective-C. Swift classes can inherit from Cocoa classes, and Swift can use any Cocoa API. Inversely, Xcode makes available Swift classes to Objective-C, and both languages can (more or less) coexist on the same project.

However, the following list of Swift features are not available in Objective-C: in no particular order, generics, tuples, enumerations, structures, top-level functions, global variables, typealiases, variadics, nested types and curried functions do not work in Objective-C. Which clearly leaves out lots of nice Swift features.

How does Swift compare?

Apple markets Swift as "Objective-C without the C," but given the previous slide, I guess you can imagine that Swift is really different from Objective-C. So, what does it compare to?

Programmers love to talk about how Java was inspired by C , or about how NewtonScript inspired JavaScript or not. The truth is, this does not matter, but for the sake or throwing flames, here is what we know about Swift:

- Its syntax looks incredibly similar to Scala.
- It has lots of features borrowed from C#.
- It does not suck too much like JavaScript.
- And apparently it feels a lot like Haskell.

But the Haskell comparison is the one that won; even Chris Lattner stated in his home page that Swift took lots of inspiration from Haskell, and this has led the whole internet community to rejoice.

I have made my own investigation, and indeed, there was a certain Charles Haskell Swift, born in Massachusetts in 1838; then I found out that you can buy a Suzuki Swift in Haskell, Oklahoma; not only that, but both Taylor Haskell and Taylor Swift have Facebook profiles; I can safely say that there is a clear connection in there.

So, Haskell developers; apparently you are going to love Swift. And Apple is wrong; Swift is not Objective-C without the C; it is more Objective-Haskell, actually.

Features

So, what features does Swift bring to the table? Well, all those you have seen in the previous slide plus the following:

- Strong typing
- Type inference
- Closures
- Optionals / Nullable types
- Generics
- Custom operators
- Tuples
- Interoperable with Cocoa
- Changes every week

Is Swift functional? Not really. Although the language could have been designed as a purely functional language, it is not; to state it simply, it allows for side-effects and mutable values, and it also lacks many functional libraries usually available to operate on lists recursively. Even more, Swift has to operate and work with the Object-Oriented world of Cocoa, so clearly Swift is a transitional language, which will take us, app developers, from an Object-Oriented Cocoa library to a new one, probably based on functional concepts.

So, here is my blunt statement: “Cocoa is the new Carbon.” The standard library brought by Swift will, I think, grow in the future, and probably allow

for 100% pure Swift implementations, without requiring the use of the Cocoa Frameworks.

But, this is an opinion, of course. You never know with Apple.

In any case, Swift is built for speed. One of the biggest goals of Swift is to provide a high-level language that provides raw metal performance at the lowest possible level. It has abstractions of sufficiently higher level for getting things done fast, but it also brings malloc() and pointer manipulation in case you need it. Right now, many early tests show that Swift code is already faster than the corresponding Objective-C code.

Instantiation

Let me open a Playground. An Xcode Playground allows us to write code and to see it crash in real time. It is a practical implementation of the "fail fast" mantra; if you are going to fail, you'd better fail as fast as possible.

So, let us start by importing some code. I am going to import my own framework, one that I've created for this presentation (yes, the code will be available in GitHub, of course) so I can show you some cool features of the language.

So I create a Playground by selecting "File - New - Playground" in Xcode, all while I pray for Xcode not to crash on me. I save the playground on my Desktop, because that's where I save most of my stuff anyway.

The first thing I'm going to type in my playground is import PresentationKit. Most frameworks in OS X and iOS have the "Kit" moniker, so when I created mine I figured out that I might as well use that suffix, too.

The first thing that strikes about Swift is the lack of semicolons. In Swift semicolons are really optional, unlike JavaScript that tells you the same until you minify your code and your application breaks. In Swift you only need semicolons if you have two statements in the same line; otherwise, hold the semicolons.

After I import my framework I can start to use my own classes in it. I'm going to create an instance of my presentation slides, and for holding those slides in memory I am going to create a variable. How do you create a variable in Swift? Just like in JavaScript: var presentation.

The keyword var does not convey, as you might imagine, any type information with it; every variable is declared using the same type.

The next thing I'm going to do is to create a new instance of a class I have defined in my framework. The syntax is the same as in Python, and I just have to write: var presentation = Presentation().

So now I have a presentation variable holding a reference to an instance of the Presentation class in memory. The Swift code above is technically the same as the following Objective-C code:

```
Presentation *presentation = [[Presentation alloc] init];
```

In Spanish there is a saying that goes "Lo bueno, si breve, dos veces bueno," which is roughly translated as "What is good, if brief, is twice as good." Well, not roughly because I was born in Argentina and Spanish is my first language, so I can assert that the translation is correct.

Type inference

One of the most fundamental features of Swift is type inference. We do not need to explicitly provide type information to the presentation variable, as we leave the Swift compiler to "guess" that for us. Unlike in JavaScript, variables in Swift are strongly typed; you cannot assign a String to a variable that has been used to hold an Array previously; Swift will complain about it!

```
var presentation = Presentation()
presentation = "some string" // -> this yields a compiler error
```

Swift automatically infers the types of the variables at compile time and will use that information to make sure that we do not screw things up, which is one of the basic capabilities of human beings. (minute 9)

Constants

Instead of using the var keyword, we could have used the let keyword, and that would have created a constant value instead of a variable value. Why they chose let instead of const is beyond me.

One of the nice things about let is that it brings a smaller version of a very nice feature of C , namely "const correctness." Any C developers in the room? You know const correctness, is when you have method signatures that go like const string const * methodName() const and you have a lot of const keywords everywhere, and each has a particular meaning; but the one I talk about is the one at the end, which states to the C compiler that this method is not allowed to have "side effects" on the current instance. And this is very important, as it allows the compiler to optimise things a little bit, and brings Swift closer to the functional programming world.

When using the let keyword, Swift will similarly block the objects assigned with it to be mutated – but of course there is a mutating keyword that you can use in case you want to make your life even more miserable.

Functions, Methods and Closures

Swift can be seen as a "federation of languages," a bit like Scott Meyers has described C ; you can use Swift as:

- A procedural language
- An object-oriented one

- A functional one

There are some "metaprogramming" features introduced by Generics, but at least in the current implementation they are very limited, particularly compared to what Templates have brought to C .

Functions are a basic block in Swift. Actually, Swift functions are blocks, in the Objective-C sense of the word; you can pass a Swift function whenever you see an Objective-C API that expects a block, but without the "caret" (^) character everywhere. That's already a bonus.

Even better, functions in Swift are closures AND first-order objects; you can pass them as parameters, return them from other functions, and even better: class methods are curried functions. So functions are a very important part of the life of a Swift developers, and it is very important to know them well.

To define a function, use the func keyword:

```
func functionName(parameterName: InputType) -> OutputType { }
```

Following the tradition of Objective-C, Swift functions have parameter names, which is one of the best features of our old beloved language. You do not have to guess the semantics of the 4th parameter of that function; just read the name and you'll see what it is about.

Functions can be overloaded; you can have several functions with the same name, as long as they take (or return) different parameter types.

Optional Types

This is also a very important feature of Swift. Any type in Swift can be defined as optional, just by using the ? interrogation sign after it. What does it mean? It means that the type can be nil or not; which is the default behaviour of pointers in C, C and Objective-C. When a type is specified as not optional, it cannot handle a nil value, and the application will crash if that situation arises. This safeguard helps app developers uncover "null reference" situations as early as possible.

Unicode

This is arguably the most important feature of the language. Unicode variables and values allow for incredibly stupid code, which is actually extremely hard to write and read, but which looks great in a small handful of editors that are able to handle Emoji correctly.

So, in all of its glory, let us check some examples of this powerful feature.

Structs, Enums and Classes

Swift has Classes, of course, which follow the classical pattern of Objective-C and Java; single inheritance with protocols. You have only one base class, but you can implement a large number of protocols.

Class instances are instantiated on the heap and are passed as references; they work exactly just like in Objective-C. However Swift has Structs, which are instantiated on the stack and passed around by copy. They are much more powerful than C structs, in spite of the name, and they can comply with protocols (that is, implement interfaces in Java speak) and have methods, just like classes do.

Another difference between classes and structs is that the latter cannot use inheritance; this means that structs are well suited for small… structures of data that you want to pass around as values: points, vectors, coordinates, etc.

Enums in Swift are a completely different beast; they can have associated values, and developers can convert from and to those associated values. Enums also support methods, and can implement protocols, which make them extremely powerful. They are used to describe entities who have a limited number of possible values.

Protocols, Extensions and Generics

Protocols are to Swift what… @protocol are to Objective-C, or interface to Java and C#; in pure C speak, they are abstract classes with pure virtual methods. They allow for inheritance of interface, instead of inheritance of implementation, like the model provided by C .

Extensions in Swift are equivalent to Objective-C categories, that is, a mechanism used to extend existing types without using inheritance. They are extremely useful as well to separate the implementation of a class into distinct files or chunks, allowing developers to clearly separate concerns and to organise the code clearly.

Finally, generics in Swift are very similar to their counterparts in C# or Java; for the moment at least they do not allow for the creation of the same structures as those allowed by C templates, such as those described by Alexandrescu.

Custom Operators

Ah, custom operators. I can hear C developers screaming. Please, C , scream. Yes, we have custom operators in Swift, and they are weird beasts. They are defined globally, outside of any class, and not inside the class on which they operate. There can be operators infix (that is, in between values), prefix and postfix (which are obvious, I suppose.)

Interoperability

You can mix and match Swift and Objective-C classes in your project, however Swift is much more advanced and many of its features are strictly not compatible with Objective-C.

A Sample Application Using WebKit

I'm going to end this presentation by building a small application on the command line, this time not using a Playground but actually using Vim and iTerm and tmux and the command-line compiler. Because, you know, I'm rebel that way.

Many things I have not talked about

- Standard library
- WillSet / DidSet in setters
- Memory management in closures
- Convenience initializers
- Lazy properties
- Nesting of classes, enums and structs
- Pattern matching in switch statements
- reStructuredText documentation headers
- Default parameter values in functions
- Monads, futures, promises and other functional programming constructions
- Nested comments