What's Different About Swift?

Craig A. Will

Apple's announcement in June, 2014 of their new Swift language shocked the programming world and very probably makes Objective-C--the language normally used for building iOS apps--a dying language.

The need for a new language has been quite clear for some time. The surprise among developers was mainly that Apple actually did it and also that they had managed to keep it secret.

In looking at the language in detail there is another surprise. This language is very ambitious. Typically, mainstream programming languages, which Swift is, fall into one of two camps. They are either high-level scripting languages that are designed for ease of use (e.g., Python, Ruby), or they are lower level systems languages that are designed to be efficient (e.g., C, C++). Swift aims to be both as easy to use as a scripting language while at the same time run very fast. In addition, Swift intends to have a high degree of safety, meaning that it is designed to minimize and quickly detect errors by programmers.

Apple sets a very high bar for Swift by claiming that it is "the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language".

In this paper I will describe the design of Swift and discuss the extent to which Swift meets its stated goals.

I will first briefly describe Objective-C, how it came to be, and why there has been a need for a new language.

Next, I will discuss some of the philosophical assumptions behind the Swift language.

I'll then go through the four aspects of Swift that Apple has indicated as the main goals for the language and describe what has been done in Swift for each of these characteristics. Swift is intended to have (1) Safety; (2) Clarity; (3) Modernity; and (4) Performance.

Finally, I'll draw some conclusions about Swift. In this discussion I will try to answer two questions:

(1) Will Apple achieve the goals it has set for Swift? and (2) What does the introduction of Swift mean for iOS app development?

What's Wrong With Objective-C?

Objective-C is peculiar as a programming language in that it was created by adding a new language on top of an existing language. The old language was C. The new language was based on Xerox's Smalltalk-80 object-oriented language and was initially a very thin layer indeed. The "Objective part of Objective-C" simply allowed the passing of "messages" to objects that either existed as part of a library or that were created by defining them in Objective-C. A "runtime" program, written in C and assembly language, interpreted the messages, which resulted in the call of methods that were eventually executed as C code. The real capability of the language was contained in the library of classes that implemented sophisticated user interface functions and systems functions.

The motivation for this hybrid language was driven largely by practical and business considerations. The original Objective-C was developed in the early 1980s to run in a computing environment that needed C because of existing code in that language. When the language was adopted by NextStep, a company that Apple later acquired so that the operating system it had developed could be used for the Macintosh, agreements were made that allowed NextStep to use the language itself and compiler as open source while the extensive library that NextStep developed that contained most of the functionality was proprietary and owned by Apple.

Objective-C was modified in 2006 to provide properties (a way of accessing instance variables of objects from outside an object) and a few other capabilities as part of "Objective-C 2.0". Later closures (called "blocks") were added, a way of passing a small chunk of code to an API along with variables and their values so that they could be executed at a later time. Capabilities for defining literal expressions (e.g. strings like @"Hello") in arrays, dictionaries, and numbers were added that made the language a little better, but still clumsy. The ability to access arrays and dictionaries with subscripts (e.g., anArray[5] = @"Hello") was also added.

A major addition was Automatic Reference Counting, an extension of the reference counting memory management scheme used to allow tracking and reuse of memory for objects in the heap part of the random access memory. This allowed reference counting to be used with relatively minimal effort by the programmer, (who now only needed to make sure there were no memory leaks caused by improper structures that the compiler could not detect, known as reference cycles.) This was a major advance in ease of use for programmers, who no longer had to track memory usage and specifically write code to deallocate memory. It also meant increased safety, since failure to deallocate and thus reuse memory (leading to memory leaks) and programmers attempting to deallocate memory that had already been deallocated (causing a crash) were major problems.

Despite these changes, Objective-C still has major limitations. It has safety issues. It is messy, ugly, and often does things in convoluted ways. And, being long in the tooth, it lacks many of the things that new scripting languages like Ruby and Python have, much less specialized languages like Haskell.

It's Not Very Safe

A safe programming language helps programmers avoid errors.

Objective-C is heavily dependent upon pointers to access objects in memory. A pointer holds a direct address of the memory location that an object (stored in the heap memory) resides in. The problem is that if a pointer gets corrupted, code that intends to write to an object's data ends up writing to the wrong object. Or, as one commentator put it, "With pointers you are one dereference (conversion to an address) away from scribbling all over memory."

Although Objective-C requires data types to be declared, it is nearly as flexible as many scripting languages in its willingness to convert data to another type when it seems desirable. Keeping types flexible was a major part of the design philosophy, which including making decisions about which method would get executed for a given message at runtime ("dynamic dispatching") rather than predetermined by the compiler.

Another common safety issue with C is programmers forgetting to put in code to deal with nil values.

In addition, Objective-C allows almost anything to be evaluated as a Boolean in an if statement, and has a confusing mess of data types that are defined in some way as Booleans.

It's Messy, Ugly, and Cluttered

Objective-C retains the division of header (.h) and implementation (.m) files used in C, requiring one for each class. Every class that refers to another class has to include a header for that class to get a program to compile. Things get messy when class A requires a header for class B, but class B requires the header for class A.

Properties must be defined in header files, along with their types, whether they are "atomic" or not (need to be written as a whole) and an ownership attribute ("strong", "weak", etc.). Header files must have signatures (names and types of input parameters and return values) of all methods called by other classes. In general, it takes a lot of code to define a class. Often, you'll write the code in Objective-C to do something, and when you are done, you think, "I sure had to write a lot of code to do that!"

Clumsy aspects of long-ago-decided C syntax, such as that used in switch statements, continue as a legacy.

A look at the syntax for blocks suggests how convoluted they can be. This declares a variable so that it can be assigned a block, then does the assignment, including the code in the block:

```
double (^blockVariable) (double a, double b);
blockVariable = ^double(double a, double b( {
    return a * b;
};
```

A look at modern scripting languages suggests many other aspects of its lack of clarity.

For example, the code below concatenates two strings in JavaScript:

```
var s1 = "Hello ";
var s2 = "friend.";
var s3 = s1 + s2;  // s3 is now "Hello friend."
```

In Objective-C, the conventional way of concatenating two strings is:

```
NSString *s1 = "Hello ";
NSString *s2 = "friend."
NSString *s3 = [s1 stringByAppendingString: s2];
```

It's Not Very Modern

Objective-C does not allow functions (and methods) to be "first class citizens" and thus passed as data values, assigned to variables, or returned from (other) functions.

Enumerations in Objective-C are crude, defined as integers. Objective-C allows type overloading (meaning that the same method names can be used in different classes), but not function overloading (different types or mixtures of types with the same name) or operator overloading (using the same operator but different types for operands). Custom operators cannot be defined. Generic programming, or writing code that can operate with input parameters of multiple types without implicit type conversion, is not in the language (although it can be added), often resulting in essentially the same functions having to be written multiple times.

Other languages allow operations on strings, arrays, and dictionaries to be done in the language itself. For Objective-C, however, this is done more clumsily by calling the library, which handles arrays (NSArray and NSMutableArray), dictionaries (NSDictionary and NSMutableDictionary), and strings (NSString and NSMutableString).

One commentator writing in Ars Technica, John Siracusa, suggested in 2005 that Objective-C's failure to keep up with other languages and their higher level of abstraction was a serious potential problem. By 2010 he was calling the language (and the associated Cocoa API, designed to work with a low-level language) a "ticking technology time bomb". He suggested that it was only Apple's success with its mobile products--and the constraints involved to make software on those products work that favored lower-level languages--that has distracted developers from this reality, giving Objective-C "new life".

Swift's Basic Philosophy

The four stated goals of Swift (Safety, clarity, modernity, and performance) reflect both an accurate analysis of the deficiencies of Objective-C and an fair assessment of the various developments that have taken place over the last three decades in programming language design--mostly object oriented languages like C++, Java, Python, and Ruby, but also more specialized languages such as the functional language Haskell.

The Swift project was initiated by Chris Lattner in July, 2010 and Lattner has specifically pointed to some of the languages mentioned above as having influencing Swift.

At least four ideas seem major philosophical underpinnings of the new language. These include: (1) Static typing; (2) Make it easier to be safe but don't be heavyhanded; (3) Passing functions and closures as data; and (4) Not everything is an object.

Static Typing.

The use of static typing, and the particular methods used in Swift to enforce a high level of type safety, are some of the biggest differences between Swift and Objective-C. Objective-C adopted the extreme approach of Smalltalk-80 of deferring whatever decisions could possibly be deferred from compile time to runtime, including, particularly, the type of a variable. This has been completely reversed in Swift, in which types are known to the compiler and will not change during runtime.

Make it Easier to be Safe but Don't be Heavyhanded

Swift is designed to encourage safe practices and to discourage bad ones. At the same time, the programmer is still in control. Often, engaging in a less safe practice requires the programming to take an explicit step that he or she must be

aware of, but the language rarely prohibits it or makes it especially difficult.

Passing Functions and Closure Expressions as Data

Swift allows functions, methods, and closure expressions to be handled very much like data. This is common in other languages, and was added to Objective-C relatively recently for closures/blocks only. It makes many interactions with an API (especially those that might otherwise have used delegation) simpler and cleaner and allows some forms of functional programming.

Not Everything is an Object

Swift avoids some of the extreme aspects of languages like JavaScript that have chosen to make everything an object. Instead, Swift has extended the idea of a structure (from C), made it a sort of lighter weight and safer version of a class, and implemented many of the primitives of the language in structures, such as numbers, strings, and characters.

Safety

Swift has a number of capabilities that promote safety. These include type safety, an avoidance of pointers, value existence safety (the use of optionals), and a few lesser steps, including realtime checking for arithmetic overflow and underflow, array bounds checking, some added rigidity in if clause testing, and encouragement to use constants rather than variables.

Type Safety

Type safety is the most obvious and the most important aspect of Swift's focus on safety. Knowing the data type of a stored value is necessary for interpreting what would otherwise be just an incomprehensible sequence of bits that has been retrieved from memory. If the data type is not correct, it's a big problem: that sequence of bits will be interpreted as something it is not. This is a major source of programming errors.

Swift has a high degree of type safety, obtained by having relatively rigid rules about how types are handled. The type of a variable must be defined, and, once set, it cannot be changed (with a few narrow exceptions). Values cannot have their types changed implicitly, such as changing an integer value of 2 to a floating point value of 2.0 when the compiler sees that the integer will be added to another floating point value. This contrasts with most scripting languages, which are much more flexible about types.

Types can be explicitly defined by the programmer as follows:

```
var x: Int
```

The ": Int" after the variable name x defines the type of that variable. Types can also be set by type inference, such as:

```
var x = 5
```

In this case the compiler will assign a type based on an analysis of the literal value 5, which it infers to be an integer, or Int. Types can be inferred based on literal values or on types of values received from an API. In typical programs, the vast majority of type definitions are done by the compiler using type inference. So Swift's rigidity about types is not very burdensome on the programmer.

Types can be explicitly converted by the programmer, such as in the following:

```
var x = 5
var y = Double(x) + 3.4
```

If an attempt were made to just add x to 3.4 there would be a compiler error because x was inferred to be an integer. But the Double constructor creates a floating point type from the integer value of x, which can then be added to the value 3.4 (which is also a Double).

Types that are class instances in an inheritance hierarchy can also be modified to change the type to a one that is higher or lower in the hierarchy. This is known as *type casting.* Type casting of class instances is the only modification to a type that is allowed.

Another aspect of type safety is restricting arrays to holding elements of only a single type. Swift's adherence to the object-oriented principle of polymorphism is interpreted here as allowing types that have a common ancestor in an inheritance hierarchy to be effectively of the same type, and thus allowed to be stored in the array. The array in such a case is then marked as having the type of that ancestor.

Arrays that result from reading an array from an Objective-C API may also have heterogeneous types, with the Swift array marked as an Any or AnyObject type, Swift's version of Objective-C's `id` generic type. Defining an array with a broad type is something of a loophole that avoids the restriction, but the practice should be to avoid doing this when creating arrays, and to quickly convert any arrays read from an API to lower-level types as soon as possible.

Dictionaries in Swift have a similar restriction, with all keys in a dictionary required to be of the same type, and all values required to be of the same type (though the keys do not have to be of the same type as the values).

No Pointers

Objective-C and other C-like languages use *pointers.* A pointer contains a number that specifies the location of the address in heap memory where a block of memory for a particular object is stored. The syntax looks like this:

```
NSString *pointerToHelloString = @"Hello"
```

There is a big problem and a little problem with pointers. The big problem is that if a pointer happens to be written to with the wrong address (which can happen with bugs in arithmetic, and especially if a programmer likes to play tricks with pointers), it can result in a large block of memory being overwritten with the wrong information.

The little problem with pointers is the asterisk, which indicates that the variable is a pointer. It's easy to leave the asterisk off, a small annoyance for the programmer.

Value Existence Safety and Optionals

Another important aspect of safety is dealing with situations in which variables do not have actual values. This is a common source of programmer error. Code is written with the assumption that a variable will have a value. At some later time, the code is executed but the variable does not have a value and bad things happen. This can occur because the programmer makes a wrong assumption, with prototype code written fast that turns into production code, or because the programmer plans to fix it later and then forgets. Ensuring that this does not happen might be called *value existence safety.* A common solution is the use of what are known as *optionals.*

In Swift, a variable either has a value or it has a `nil`. Also in Swift, an ordinary variable, known as a nonoptional, is required to have a value, which is to say that it is not allowed to have the value `nil`. An attempt to store nil to a nonoptional variable will cause a program exception. This might be considered a first line of defense in the optionals strategy for safety.

To be allowed to not contain a value (that is, contain nil), a variable must be declared as an optional, which is done by appending a question mark to its type, such as `Int?`.

Normally, a variable that is an optional must *unwrap* its value. This step is the second line of defense in the optionals strategy. Unwrapping can be done by appending an exclamation point to the variable name, like x!. The trap is that an attempt to unwrap an optional that does *not* have a value will trigger a program exception. So a common thing to do is to test the value of an optional variable and unwrap it only if it has a value.

Swift has a number of additional capabilities to deal with optionals--ways of testing for a value and unwrapping that are a little safer or easier to use, and ways of indicating that a variable is trusted to not be `nil`.

Optionals help to avoid errors resulting from values that should be there but aren't, but programmers must do the work. Especially when converting, say, Objective-C code to Swift it is easy to look at a large number of variables that produce compiler errors because the API they were read from said they were optionals. Programmers will then pepper the code with "?" characters to shut up the compiler on this issue, then pepper the same code with "!" characters to stop the runtime errors because the optionals have not been unwrapped. Doing this in a through less way is easy but circumvents the value of optionals.

Checking the Boundaries of Arrays and Overflow/Underflow of Values

When a program is executing and an array is accessed, Swift will check the index used to access that array to see if it is within the bounds of the array and an exception will result if it is not. Similarly, when an arithmetic calculation is done a check will be made to see if the result is larger (overflow) than the type of the variable where it will be stored allows, or smaller (underflow) than allowed. An overflow or underflow will trigger an exception. This behavior can be turned off if desired with compiler flags to improve performance.

Booleans and Assignments in If Clauses

In Objective-C and many C-like languages, an `if` clause can test an arithmetic or logical expression that contains arithmetic values. A result of 0 is interpreted as `false`, while a result of anything nonzero is interpreted as `true`. Values of `YES` or `true` also count as true. Swift is much more rigid. Only logical expressions, with Swift Boolean types with values of either `true` or `false`, are allowed.

It is a common practice to put assignment statements within if clauses, such as:

```
if ( x = y - 4) { println("do something");
```

If the resulting assignment is 0, it counts as false, while if the result is nonzero, it counts as true (and the statement is executed). This makes a language vulnerable to a programmer who intends to enter a conventional "==" operator but accidentally leaves off one of the equal signs.

Encouragement for Using Constants

In Swift, constants have a syntax similar to variables and are treated similarly. A variable is declared as follows:

```
var x = 5
```

A constant is declared like this:

```
let y = 8
```

The only real difference between a constant and a variable is that a constant can have a value stored in it only once.

Using a similar syntax makes constants easier to use (and not ugly, like X EQU 5). More importantly, it allows programmers to realize that they can often use a constant when they might normally expect to use a variable. This is safer.

Overriding Inherited Methods

Swift has conventional classes, objects, and inheritance. As in Objective-C, a method that has been inherited can be replaced for that class, or *overridden,* by providing code in a method in the class definition with the same name (and input parameter and return value types). Unlike Objective-C, the keyword `override` must be used in Swift. This is to prevent a programmer from accidentally overriding an existing method by creating what he or she thinks is a new method. If the keyword `override` is not used there will be a compiler error. If a programmer attempts to override a method that has not been inherited, there will also be a compiler error.

Access Control

Swift controls access to code based on both where the code is and on tags in the code that specify it to be `public`, `internal`, or `private`. Every app runs in a particular module that can consist of multiple .swift source files. Code in a particular source file has access to all the code in that same source file. Code in a particular source file has access to code in other source files in the same module only if the particular entities in the code have been tagged to have an access level of `public` or `internal`. Code in a particular source file has access to code that has been imported only if that code has an access level of `public`. If code is not tagged it is assumed to be `internal`.

Clarity Helps Safety

The next section describes what has been done in Swift to make programs have better clarity. Although intended mainly to make programming easier and more fun, clarity also helps avoid errors.

Clarity

Code that is clear and clean helps make programming pleasant and fun.

Simple File Structure with No Cruft

Gone are the two header (.h) and implementation (.m) files for every class that you would see in Objective-C. You don't need to import header files. Swift figures it out. You can have a file (.swift) for every class if you like, or put as much as you like in a single file. You'll likely just need a single import file for the framework packages that you'll be using, often just:

```
import UIKit
```

And no messing around trying to figure out whether the framework names should be in angle brackets or quotes.

You won't need a lot of cruft to define properties and list signatures of methods that you use. And you won't see _variable names with underscores.

Semicolons Not Necessary at the End of a Line

In Swift, both of the following statements are valid:

```
var x = 5
var x = 5;
```

Semicolons are still necessary to separate statements if there is more than one statement on a single line:

```
var x = 5; y = 7
```

Even in this case a semicolon is still not required at the end of a line.

Parentheses Not Required for Conditional Statements

Conditional expressions statements in C-like languages usually look like the following:

```
if( x == 5) {
    printf("x is equal to five")
}
```

In Swift, you can leave out the parentheses:

```
if x == 5 {
    printf("x is equal to five")
```

```
    }
```

Parentheses are still *allowed,* and in the case of complex expressions might be desirable, either to group things to make them more readable or give some parts of the expression precedence over others:

```
    if x == 7 && y ==9 || (a != "basic" || z == "hey") {
         println("The parentheses make this expression
easier to read.)
    }
```

Nested Comments / */*

Perhaps you have written code in a C-like language, putting comments in like this:

```
    x = 5;
    /* This is a comment, using the slash asterisk style to
allow
        more than one line for a given comment, in this
case
         three lines in total */
    y = 7
```

And then, later, you wanted to comment out the *all* the lines, including the assignments to x and y, to see if it makes a difference.

What you want to do is this:

```
    /*
    x = 5;
    /* This is a comment, using the slash asterisk style to
allow
        more than one line for a given comment, in this
case
         three lines in total */
    y = 7
    */
```

You want to nest your comments using the /* */ comment style. This is not allowed in C, but it *is* allowed in Swift. This is a small thing, but it tells you something about the willingness of the designers of Swift to worry about the smallest details.

Closure Expressions

Swift has *closure expressions,* which are pieces of code that can be manipulated in very flexible ways.

A closure expression is Swift's version of an Objective-C block, but has a cleaner and more flexible syntax.

Closure expressions can be very simple. The expression to the right of the equals symbol in the second line in what follows is a closure expression:

```
var m = 0
var a = { println(m) }
m = 5
a() // prints 5
```

In that second line, the closure expression `{ println(m) }` is assigned to the variable a. The closure expression itself is not executed.

In the fourth line, the a() syntax causes the closure expression to be executed. The interesting thing here is that what gets printed is 5, not 0. This is because the code in the closure is not executed until the `a()`, and after the value of the variable has been changed to 5.

(In other contexts, the value printed might be 0, if the closure expression saves the value of the variable at the time the closure expression is defined, but that' does not happen if the closure is defined, as it is above, at the global level.)

The closure shown above is about the simplest possible closure. An example of one of the more complex form of a closure is shown below:

```
var a = (a: Float, b:Float) -> Float in {
    var c = a * b
    return c
}

println(a(2.0,3.0))   // prints 6.0
```

This is a closure that is assigned to the variable a. It accepts two floating point input values, multiplies them together, and provides the result as a return value. (The "->", known as a *return arrow,* is part of the syntax of a more complex closure. The type to its right indicates the type of the return value. The last statement does the same thing as the a() in the first closure example, but in this case the closure code accepts the two input parameters. This prints the value 6.0. This variation works like a function, and in fact the main difference (aside from some syntax differences) is that the closure does not have a name.

Closures can have many variations of syntax that range between the simple version that has only a pair of braces (curly brackets) and the more complex version that has input parameter names and types, a return arrow and return type, the keyword `in`, the braces and a `return` keyword. In general, the most minimal form of the syntax that can be used given what needs to be in the closure expression is what is used.

Closures are often used to pass code as an argument in an API call that is intended to be executed when an API performs some action and initiates a callback. This is a common alternative to the use of a delegate method, because it allows the code to be placed in the call to the API rather than buried in a delegate method. This provides cleaner, more readable code.

Strings

One of the more antiquated statements in Objective-C is the standard way to concatenate strings:

```
NSString *a = @"This is rather ";
NSString *b = [a stringByAppendingString: @"absurd"]
```

This is replaced in Swift by:

```
var a = "This is rather "
var b = a + "absurd"
```

Earlier, in discussing safety, I noted that pointers were discarded because of their lack of safety and because a programmer could forget the required asterisk.

Now, Swift allows us to use just a pair of double quotes rather than an at-sign and double quotes for a string, cleaning that up.

And, finally, we've gotten rid of the standard Objective-C way of concatenating two strings. (Most programmers found another way to do this.)

Mattt Thompson (NSHipster.com) is skeptical that using an "addition" operator to mean concatenation makes much real sense. But it beats stringByAppendingString for brevity.

Appending to the End of a String or Array

A variation on the above is simply appending a string, or elements of an array, to an existing string or array. Swift not only allows the addition operator, but the *addition assignment operator.*

Thus:

```
    var a = "This is rather "
    var a += "absurd"
    // a is now "This is rather absurd"
```

And for arrays:

```
    var cities = ["San Francisco", "Oakland"]
    cities += ["Monterey", "Los Angeles"]
    // cities is now "[San Francisco", "Oakland",
"Monterey", "Los Angeles"]

    airports = ["San Francisco", "Oakland", "San Jose"]
```

Arrays can be very nicely addressed with subscripts, as is common in other languages:

```
   airports[2] = "Monterey"
```

(Some of this has been added fairly recently to Objective-C.)

A simple syntax for ranges has been defined:

```
   airports[2...3] = ["Monterey", "San Luis Obispo"]
```

The `[2...3]` notation indicates the elements in the array accessed by the subscripts 2 through 3. An alternative `[2..<4]` notation means the same thing, subscripts 2 through 3. The names for these are a little obnoxious. `[2...3]` is called a "full closed" range. `[2..<4]` is called a "half open" range.

Tuples

Tuples are a simple way of handling multiple values that are of different types. They are most obviously useful when you want to return multiple values from a function. Functions are designed to allow multiple input values, but only one output value. Putting such values in an array seems like overkill, and in Swift all of the values in an array are supposed to be of one type.

```
    func getErrorCodeAndMessage () {
        return (345,"Not Found")
    }
    var error = getErrorCodeAndMessage()
    println(error.0)  // prints 345
    println(error.1)  // prints Not Found
```

Defining and Using a Class

In Objective-C, defining a class is a bit of a mess. First, in a header (.h) file, you need:

```
fruit.h file
    #import <Foundation/Foundation.h>
    @interface Fruit : NSObject {
    }
    @property (nonatomic, strong) NSString *color;
    - (void) printColor;
    @end
```

Then, in an implementation (.m) file, you need:

```
fruit.m file:
    @implementation
    - (void) printColor {
        NSLog(@"The color of the fruit is %@",self.color);
    }
    @end
```

To create an object, set a property, and execute a method, you need:

```
    Fruit aParticularFruit = [[Fruit alloc] init];
    aParticularFruit.color = "green";
    [aParticularFruit printColor];
```

Swift does it much simpler. To create the class, you just need one file:

```
fruit.swift file:
    #import Foundation
    class Fruit {
        var color = "red"
        var lengthInInches = 3.5
        func printColor() {
            println("The color of the fruit is \
(self.color)")
        }
    }
```

To create an object, set a property, and execute a method, you need:

```
    let aParticularFruit = Fruit()
    aParticularFruit.color = "green"
    aParticularFruit.printColor()
```

Requiring Braces Around Conditional Expressions

Most C-like languages have conditional statements like the following:

```
    if(x == 5) println("I am a single statement in a
conditional");  // Not valid in Swift

    if(x == 5) {
        println ("I am the first statement of two in a
conditional");
        println ("I am the second statement of two in a
conditional");
        }
```

In C and most similar languages the first statement is valid. It is not valid in Swift. Consistently requiring braces reduces errors, particularly when statements are moved in and out of the braces. (This isn't really a safety issue because the compiler can catch it.)

Changes to the Switch Statement

In C and most C-like language the Switch statement looks like this:

```
    switch(a) {
        case: 10 {
            println("value is 10")
            break;
        }
        case: 11 {
            println("value is 11")
            break;
        }
        case: 12 {
            println("value is 12")
            break;
        }
    }
```

In Swift the same `switch` statement would look like this:

```
      switch(a) {
        case: 10 {
            println("value is 10")
        }
        case: 11 {
            println("value is 11")
```

```
        }
        case: 12 {
            println("value is 12")
        }
        default: {
            println("value not matched")
        }
    }
```

The `switch` statement works a little different in Swift. In C, if a `switch` value matches the value associated with a case, the code in that case is executed, after which control will fall through to the next `case` and potentially execute additional code. To prevent this, a `break` statement is normally added as the last statement in a `case`.

In Swift, if there is a match, the code for a `case` is executed and then the `switch` statement is done. There is no need for `break` statements. (If you want the fall through behavior you can get it with a `fallthrough` statement._

Modernity

Much of Swift's claim to be a "modern" language is fulfilled by what it has done to improve safety. Modern languages typically do not have pointers. They often have something like optionals. Checking to see if a reference to an array is out of bounds, and not allowing assignments within if clauses are similarly modern, as are most of the other minor things that Swift does to promote safety.

Most modern languages, however, do not not have typing as rigid as does Swift, and in this respect Swift is deliberately non-modern, if "modern' is about the convenience (and lack of safety) of implicit type conversion. Swift's use of type inference to make its strict typing easier for programmers to deal with, though, is very modern.

Modern languages also have the clean syntax like that shown in the previous section. And they have the closures that have been described.

Modern languages do have strings, arrays, and dictionaries implemented directly in the language and allow elements to be accessed with subscripts.

Swift has modern capabilities beyond these. They include (1) The ability to pass functions and closures as if they were data; (2) Adding many modern capabilities to classes; (3) Some additional string capabilities; (4) The ability to iterate through a string, array, or dictionary; (5) The addition of structures; (6) The addition of enumerations (7) The ability to overload operators and functions; (8) Defining custom operators; and (9) The use of generic programming.

Functions and Closure Expressions as First Class Citizens

An important capability is that of treating functions and closure expressions as so-called "first class citizens". The term comes from a comment made in the 1970s about the programming language Pascal. According to this comment, it would be helpful if things like functions could be used in the same way as data, meaning that they could be passed around as arguments in calls to other functions, be passed from a function as a return value, and be assigned to a variable. Why can't a function have the same rights as a mere piece of data? Why can't a function be a "first class citizen"?

Swift defines both functions (and thus methods, which are just functions associated with classes and other instance-creating types) and closure expressions as first class citizens. It thus allows them to be passed as input arguments to other functions and closures, to be returned from functions and closures, and to be assigned to a variable.

The following shows an example of how a closure can be assigned to a variable:

```
var m = 7
var a = { println(m) }
a() // prints 7
```

This ability to pass functions as if they were data allows Swift to be used as a functional programming language. It makes for clearer code, as discussed earlier, and makes Swift more powerful.

Functional Programming

Functional programming is a very different programming paradigm from either the typical object-oriented languages of Objective-C, Java, C++, and Ruby, and procedural languages like Pascal.

All of these languages are based on the idea of statements that perform a computation by steps and write the result to memory after every step, known as *imperative computing.* This is about as basic an idea to today's programmers as the idea of water is to a fish--so much a part of the environment that he or she might not even be very specifically aware of it.

Functional programming is completely different. Functional programming attempts to specifically *not* write results to memory. To use the term commonly used, functional programming is *stateless.*

Swift is not as easy to use for functional programming as a specialized functional programming language like Haskell. However, it does have the basic capability.

Although functional programming cannot be used for every problem, where it can it can result in highly reliable, easily debuggable code that is usually much shorter than comparable non-functional code.

Adding Modern Capabilities to Classes

In addition to cleaning up the way classes are defined, Swift also adds several new, modern capabilities to classes:

Computed Properties. These are accessed like properties of an object but their values are calculated at the time they are accessed. This is useful for values that rely on real-time information (e.g.,time of day), or where it is desirable to maintain only one stored property for certain information and compute more than one value based on this information (e.g., use a stored property for Fahrenheit when storing temperature but a computed property for Celsius). A computed property can also be "written to" with the result the execution of setter code that often writes values to other properties.

Lazy Properties. Lazy properties wait to be initialized unit they are accessed. They are used in cases where it takes some effort to initialize them (e.g., accessing a server) and the property may not always be used (and thus the effort of initializing it wasted) or when it is better for a delay to occur at time of first access rather than initialization.

Flexible Initializers. Swift has a flexible (and rather complex) set of ways to initializing an object. A special method-like syntax using the keyword `init` is used, rather than the initialization methods like those of Objective-C. Defining an initializer isn't actually required if properties have an initial value defined. A more complex initializer can allow properties to be set with arguments when an object is created. So-called convenience initializers can also be defined that can be called by other initializers to provide additional flexibility.

Subscripts. Classes can be defined so that data associated with an instance can be accessed with a subscript when that way of accessing data makes sense, For example, a class PeoplesTopTenMovies might be defined, and an instance `georgesMovies` created, referring to the favorite movies for someone named George. The title of George's 3rd most favorite movie might then be accessed with `georgesMovies[3]`. Storing to and retrieving from the instance is defined by getter and setter functions in the class as part of the use of the keyword `subscript` in the class definition.

String Capabilities

As indicated earlier, strings in Swift can be concatenated by the simple addition ("+") operator and appended to by the addition assignment operator ("+=").

Strings in Swift comply with the Unicode standard, and can represent characters like Asian logographs and emojis:

```
var s = "的"
var hamburger = "🍔"
```

The Unicode characters can even be used in variable names:

```
var 的 = 37.6
```

Iterating Through An Array, String, or Dictionary

In Swift it is possible to directly iterate though an array, string, or dictionary, without having to worry about indexes. Thus:

```
var s = "This is a 🍔 hamburger"
for ch in s {
    println(ch)
}
```

This is particularly useful in the case shown, with the hamburger emoji: an iteration is done for each visible Unicode character, not for each internal character. (The hamburger emoji takes 4 internal characters to represent.)

Structures

Structures in Swift are used when you have data that has multiple instances with each having the same properties. They are in a sense lighter weight, slightly safer versions of classes. They have instances and properties and can have methods associated with them, but cannot perform inheritance. They are value types, meaning that they are stored in the stack, not the heap, and memory allocation and deallocation is not performed.

Enumerations

An enumeration in Swift is a custom data type that can only have a value that is one of a set of values specified in the definition of the enumeration.

A classic example of an enumeration is that of the day of the week. The value must be one of the following: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday. A Swift enumeration defining a DayOfTheWeek type has the following syntax:

```
enum DayOfTheWeek {
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
    case Sunday
}
```

A variable declared with a type of `DayOfTheWeek` that had a card value stored in it would be like this:

```
var today:DayOfTheWeek = DayOfTheWeek.Friday
```

The value is actually that of `Friday`, not the string "`Friday`" or an integer value representing `Friday`.

Enumerations in Swift are, like Structures, a lightweight, slightly safer version of a class. It's safer than the Objective-C version because the latter just defines each possible enumeration as a particular integer value, and it is possible for the program to write an invalid nonsensical integer value into the variable representing the enumeration. Defining an enumeration this way also makes the code clearer. Finally, enumerations make Swift more modern and powerful because the enumerations can hold associated values and can have methods and instances in the same way as structures.

"Instance Creating Types"--A Note About Terminology

The addition of structures and enumerations helps make the language more powerful and modern, but at some cost of confusion. Because they have much of the functionality of classes, lots of things in Apple's documentation have to refer to language types that include structures and enumerations as well as classes. Apple often refers to the three types--classes, structures, and enumerations--as a group using the word "types", which is very confusing, because it seems to include *any* type. I suggest calling these *instance creating types,* since this uniquely identifies them.

Overloading Operators

Swift allows overloading operators, as part of its adherence to the polymorphism principle of object oriented programming.

In operator overloading, an operator has different behavior depending upon the

data types of its arguments.

Thus, in Swift the addition ("+") operator allows you to add two integers:

```
var a = 5 + 7    // result is 12
```

However, if the types involved are strings, the result is different:

```
var a = "5" + "7"  // result is "57"
```

If this wasn't already in the language, it would be very easy to add. An operator is in most respects really a function, and it can be redefined:

```
func +(left:String, right: String -> String {
    var concat = left.append(right)
    return concat
}
```

This redefinition only applies to the context in which the operator "+" has a String value on the left and another String value on the right as operands.

Defining Custom Operators

Defining custom operators is done in a similar manner as overloading an operator.

Say we think that the name of the function normally used to calculate a square root is ugly:

```
var y = sqrt(5.7)
```

We'd prefer to use the Unicode square root symbol √. (√ can be entered on a Macintosh by pressing the V key with the option key held down.)

We first declare the operator:

```
prefix operator √ {}
```

We then define the function named √:

```
prefix func √(x:Double) -> Double {
   return sqrt(x)
}
```

And we can now use the following to calculate a square root:

```
var y = √(5.7)
```

Overloading Functions

Swift allows functions to have multiple definitions, with the same name but different types, or different combinations of types. At compile time, the correct version of the function will be identified based on the type or combination of types in the calling statement. This reflects the principle of polymorphism in object-oriented programming that Swift subscribes to.

Generic Programming

Swift also has the modern capability of generic programming, or generics.

It is often the case that functions have to be written multiple times, with a different version for each type (or set of types) that is used in its input parameters, code, and return values. Generic programming allows a single function to be written by a programmer that covers the different types. This function is then compiled automatically in the different forms for different types as required. Because the actual compiled code uses only particular types, it still has a high degree of safety, as much as if the programmer had written different pieces of code for the different required types.

An example of a function written generically is as follows:

```
func addTwoNumbers<T> (a:T, b:T) ->  T  {
   var c = a + b
   return c
}
```

The symbol T, shown between the angle brackets just after the function name, is the standard way of referring to a parameter type when only one parameter is involved. When that type is to be defined, the symbol T is used.

Performance

As part of Apple's goal of creating an "industrial quality systems programming language", Swift is claimed to have high performance. This is usually interpreted as just fast execution time, although considering the intimacy of a mobile app and Apple's reasons for choosing reference counting over automatic garbage collection, it is also fair to consider not just peak execution time but also the smoothness of processing.

Assessing the actual speed of execution of a programming language compared to another language is quite difficult. Obviously, it is easy to write the same essential program in both languages, run them, and time the results. But many issues come up. Is this particular task representative of the kind of processing that will be done by the code in a real app? Would this task even normally be done by code or would some module that is part of the API be called, better optimized and perhaps written in a different language?

Computers are fast enough, and the way interactive applications use processing power so sporadic (on average not very much and occasionally with a very high demand), that the programming world isn't necessarily that enamored of brute processing power. Scripting languages like Python are popular because whether processing is ten times slower that it might otherwise be is often less important than how easy the language is to use, how much time it takes to write the program, and how big the source program ends up being.

The normal practice is to simply write the code, and then, if there is a problem with an app running slow, to tweak the (some claim) one to five percent of the lines of code that are typically performing a time-sensitive function.

Still, Apple has set a high bar of claiming Swift to be "industrial quality", implying that it has the speed of low-level compiled languages like C and C++.

There is some benchmark data available comparing Swift, Objective-C, and C. In an August, 2014 test with the beta version of Swift then available, Jesse Squires found Swift to be surprisingly fast, ranging from six times faster to nearly eighteen times faster for different kinds of sorting. When compiler flags were set to not check for such things as overflows, Swift was even faster.

This test was later criticized because of the way that Objective-C handles arrays. While Swift stores actual integers in arrays, Objective-C doesn't store integers directly in arrays--it wraps them up in an NSNumber object. To sort such an array, you have to constantly unwrap and wrap the objects to get at the integers. The example shows in part why speed comparisons across languages are difficult. This test obviously exaggerates the difference between Swift and Objective-C. Yet wrapping up primitive values as objects is common in Objective-C, and along with dynamic dispatching of methods (see below), is a serious reason for why Swift is, or at least should be, faster overall than Objective-C.

We don't, at this stage, have seriously credible evidence comparing the speed of Swift to other languages. Interestingly, the principal person who designed Swift, Chris Lattner, is an expert in compilers. But designing the language and getting the compiler and other tools to work at all has taken priority over optimizing the compiler. And so we will have to wait until the compiler is in a more mature state to get any reliable measurements.

However, we can get some idea by looking at how Swift does things. It is pretty

clear that Swift will be faster than languages like Python and Ruby, because they are interpreted rather than compiled.

It is also clear that Swift will normally be faster than Objective-C, although, on average, probably not by a large amount. An Objective-C method is called by passing a message to an object that dynamically decides what method is to be executed. This is done by a program, part of the Objective-C runtime, known as objc_msgSend. Objective-C, based on Smalltalk's love of dynamic dispatch and "late binding of all things" (to quote Alan Kay, the principal designer of Smalltalk), uses dynamic dispatch. However, only a tiny proportion of typical Objective-C code actually needs it. And, although the dynamic dispatch code has been carefully designed to run fast--written partly in assembly code and using caching extensively when possible--it still significantly wastes processing cycles in the case of the vast majority of code that doesn't need the dynamic dispatch capability.

In contrast, Swift doesn't use dynamic dispatch at all, but simply looks up the methods in a fixed table that is produced by the compiler, which can create this because of Swift's static typing.

For various reasons, usually related to its static typing, the Swift compiler also has a greater capability than the Objective-C compiler to perform optimization, potentially eliminating code in circumstances in which it can determine that that code would not actually do anything productive. It can use memory more efficiently, in some cases allowing the use of stack memory instead of the less efficient heap memory. It can also use registers of the CPU more efficiently.

A particularly interesting case is a comparison between Swift and C. Surprisingly, Swift in some circumstances can actually be faster than C. C is required to allow multiple references to point to the same location in memory (known as *aliasing*). This makes it difficult for the C compiler to do certain kinds of optimization it might otherwise do. Swift doesn't have this requirement, and can thus in some circumstances produce code that runs a little faster than C.

(This analysis of Swift's potential speed is based on Mike Ash's analysis in addition to my own knowledge.--See mikeash.com and his blog for July 4, 2014.)

Swift's performance also depends on how it is compiled. In some tests Swift has been shown to be many times slower than C when run normally, but nearly as fast as C, or even slightly faster, when the compiler has optimized it to run as fast as possible. However, such optimization may not be a good idea. This typically turns off checking such as overflow/underflow tests when doing arithmetic and array boundary checking. It also turns off compliance with certain iSO and IEEE standards for mathematical functions, running a risk that certain algorithms will not work properly,

With regard to Swift's performance in allowing a smooth presentation of the user

interface, Swift's use of reference counting rather than automatic garbage collection indicates that Swift will be better than languages like Java running under Android. Reference counting, and automatic garbage collection, are two different schemes for managing memory and allowing its efficient reuse. Automatic garbage collection requires less work on the part of the programmer, but can cause user interfaces to freeze or "stutter" when the time comes for the system to reclaim unused memory. Reference counting, which is also used with Objective-C but no other major language environments, is far superior in allowing smooth processing. Automatic garbage collection can also significantly slow down processing speed under conditions in which memory is very constrained, a phenomenon that reference counting is far less subject to.

We can reasonably expect Swift to be far faster in terms of raw processing speed than languages like Python and Ruby, and slightly faster than Objective-C. Proof of this, however, will have to come later. We can also expect Swift to provide significantly smoother presentation of user interfaces.

Conclusions

The two questions I posed at the beginning of this paper were:

(1) Will Apple achieve the goals it has stated for Swift? and (2) What does Swift mean for iOS app development?

Achieving Apple's Goals

Will Apple's goals be achieved? The answer is yes, with only a few caveats. I'll take up each goal first.

Safety. Swift is clearly far safer than Objective-C and scripted languages like JavaScript. This is primarily due to the use of static typing and of optionals. It is also due to a syntax that is simple, clear, and well-designed. The use of type inference allows a high degree of type safety while putting little burden on the programmer. Optionals do put a significant burden on the programmer, and it is quite possible that many programmers will find ways to avoid using them in the way they are most effective, so there is some uncertainty about their real impact.

Clarity. Swift is a very clear, well-designed language, with many developers saying that it is "a joy to use", and a huge improvement on Objective-C's ugliness and clutter.

Modernity. Swift's closure and functions, including defining them as "first class citizens" that can be passed around as parameters and return values, are well defined. The ability to iterate through strings, arrays, and dictionaries, the addition of structures and enumerations, the ability to add new operators and

overload operators and functions and to do generic programming are also substantial ways in which the new language is modern.

Performance. Swift's level of raw processing performance has yet to be credibly demonstrated. But there is good reason to believe that this will be achieved, with Swift performing far faster than Python and Ruby, at least modestly faster than Objective-C, and even slightly faster than C in some circumstances.

An Easy-to-Use Industrial-Quality Language. Apple also seems to have largely achieved its overall goal of providing an industrial quality language with the ease of use of a scripting language. This has come at a cost: the language, though often touted as being simple and as helping to attract web developers to build native language iOS apps because of this, is ultimately pretty complex. Some aspects of its design, such as optionals, the new features in classes, how classes get initialized, and the complications of structures and enumerations on top of classes aren't necessarily easy to learn. An important question about a complex language like Swift is whether programmers with limited skill can use it effectively. In other words, if you have limited skill in Swift (and iOS), can you easily build a simple app in Swift? Clearly, you can often do without many things: complicated function and closure syntax, passing functions/closures as parameters, using convenience and failable initializers, computed and lazy properties, etc. The question is whether your lack of knowledge will bite you. The answer is probably, yes, but only occasionally. Every developer certainly needs to understand optionals. Understanding the basics of classes, and even inheritance, isn't difficult. Although the ultimate language is more complex than would be desired for a scripting language, Apple has achieved more than might reasonably have been thought possible.

Problems with the Compiler and Tools

Swift is not yet a production language for large-scale projects. Small and perhaps medium-scale apps have been successfully built and are in the App Store, built by developers with a certain persistence and willingness to work around problems. The compiler is slow, with delays apparently exponentially related to the amount of code and dependent upon which files and modules the code is packaged in. Error messages are not very informative, often at an extremely low level. Automatic bridging that is intended to make Swift work "seamlessly" with the iOS API doesn't always work. As discussed earlier, the compiler often produces poorly optimized code, including code with extraneous retain-release statements. The Xcode interactive development environment can be slow to respond with autocompletion and with flagging errors and turning error flags off when they have been fixed.

Apple has a high level of skill with compilers and the task of making them work is a seemingly manageable problem, unlike Apple's ill-fated Copland operating system for the Mac that they eventually abandoned, replacing it with the NextStep OS that is now the foundation of the current OS. Apple is extremely

profitable and has the resources to finish it. I see no reason to believe that Swift will not eventually have the tools that support its already high-quality design. The worst-case scenario is probably just a further delay.

What Does Swift Mean for iOS App Development?

The reaction to Swift from the developer community has generally been very positive. Criticism has been minor. This is perhaps surprising given the programmer culture in which people have strong and often crazy opinions about everything, no matter how ill-informed.

Some of the negative reactions have been predictable: Enthusiasts of scripting languages like JavaScript think that not having implicit type conversion (automatically converting a data value's type when necessary) makes the language too inflexible. This is mostly a philosophical disagreement that has no solution. Some programmers feel that errors are inevitable and want their programs to run no matter what. Others want them to quickly fail in hopes of getting every last bug out.

Others think that Apple should have developed a general language that could serve not only iOS but be cross-platform and serve Android and other operating systems. (This is actually possible with Swift but unlikely.)

Most existing Objective-C developers will likely switch to Swift as soon as the compiler and tools are stable and run fast and as managers recognize that the language is ready. Some major projects that were started in Swift reverted to Objective-C when compiler and other issues developed. Most of these projects are looking to migrate to Swift as soon as they reasonably can.

Some programmers will not be so quick to switch. Automatic Reference Counting has solved the most significant annoyance with Objective-C (having to allocate and deallocate memory manually) and its source of the most critical errors (memory leaks when memory not deallocated properly and crashes when deallocation is done unnecessarily). Many programmers will have worked with Objective-C so long that they have adapted to its quirks, are blind to its confusing aspects, and can work productively with it (although they surely spend a lot of time debugging.) There are other programmers who like doing tricky (and arguably unsafe) things with low level pointers. Some will require more convincing data about the performance of Swift. Some have already announced their attitude: "You can take my Objective-C only when you pry it from my cold, dead hands". In other words, when Apple decides to no longer allow apps to be written in Objective-C.

Apple is quick to deprecate APIs that it no longer considers those it wants developers to use, and is aggressive about pushing users and developers to the latest versions of iOS and to relatively recent hardware. But Apple clearly needs

Objective-C to maintain the legacy APIs, and parts of the iOS and OS X operating systems, that have been written in it. And there is a large base of app code that has been written in Objective-C. Apple is unlikely to be very quickly aggressive about getting to developers to switch to Swift. But Apple *has* made it easy to mix Swift and Objective-C code. It is quite possible that Apple will slowly nudge developers in the direction of using Swift, without prohibiting Objective-C. This might include requiring apps being submitted to the App Store to be have their root controller written in Swift but allow calling on Objective-C code. It could also involve developing new APIs for Swift that do not work in Objective-C.

The biggest question is what will happen with Cocoa Touch. There is a belief by some that Swift will motivate web developers to move to iOS native app development because the language's similarity to a scripting language. This isn't realistic for a number of reasons. The complexity of Swift will still be something of a barrier, although nothing like the steep learning curve Objective-C has long been for programmers, many of whom never got the hang of it and gave up. But the real problem is the complexity of the Cocoa Touch APIs used with iOS. These are far more complex than the Swift language, and many are arguably just as unpleasant to deal with as the worst aspects of Objective-C. As Mattt Thompson (NSHipster.com) has indicated, many things could be done with Cocoa Touch that would better fit the Swift language than the current APIs.

It's quite possible that Apple could make Swift open-source. Mattt Thompson has suggested that Apple could put a Swift capability in the Safari browser as a first step toward potentially replacing JavaScript as a web browser language. It would be in Apple's interest to make Swift work in a broad range of environments and thus increase the number of skilled Swift programmers. And the fact that Swift's claim to be easy to use, safe, and fast is fundamentally true does make it a language that could be applied far more generally than just for iOS apps.

Apple has produced an excellent language with Swift. There is every reason to believe that Apple will develop effective tools that will make Swift a large-scale production language. Swift's advantages over Objective-C will induce the overwhelming majority of developers to use Swift for building apps, probably within the next couple of years for new projects. Revising the Cocoa Touch APIs to make better use of Swift and to generally be a cleaner, more productive API for programmers should be a next step for Apple.