Objective-C and Swift: A Side-by-side comparison

1.0 The Basics

This document is basically a bunch of notes I have taken with the intent on sharing them. The subjects here are presented in the same order as the **Swift Programming Language** iBook provided by Apple. I encourage you to check out the iBook yourself, after all it's free. https://itunes.apple.com/us/book/swift-programming-language/id881256329?mt=11

Variables

In swift, variables can now be implicitly defined. This is a stark contrast from Objective-C which requires all variables to be typed. Now you have an option.

Standard Variables	
Swift	**Objective-C**

`var anInt= 5`
`var aFloat = 5.0`
`var aString = “Some String”`
	int anInt = 5;
	float aFloat = 5.0;
	NSString *aString = @“Some String”;

Constants	
`let aConstantInt = 5`	int aConstantInt = 5;
Typecasting	
`var aNewInt = anInt + Int(aFloat)`	int aNewInt = anInt + (int)aFloat;

It's worth noting here that in Swift the type of **aNewInt** can be inferred, but you have to explicitly typecast the float to an int for this to happen. In Objective-C aNewInt is defined as an int so the typecast is not necessary. if you explicitly define **aNewInt** as an Int in swift , you still have to typecast the float.

If you would like to explicitly type your variables in swift you can:

Printing	
Swift	**Objective-C**
println(“an int \(anInt) and a string \(aString)”)	NSLog(@"An int %d and a string %@",anInt, aString);

let explicitString: String = “This is a string”

Printing

For debugging, it’s good to know how to print out your variables. The nice thing about swift if you no longer need to worry about the print formatters.

Comments

Comments are exactly the same, woohoo!

Semicolons

Forget em in swift if you want. Unless you want more than one statement on a single line.

Integers

Integers

Swift	Objective-C

`let aSignedInt: Int16 = -1_000`
`let anUnsignedInt: UInt32 = 1_000_000`
	Int16 *aSignedInt = -1000;
	UInt32 *anUnsignedInt = 1000000;

Integers can be defined just as they would be in Objective-C. Well mostly. Inference depends on your platform, if platform is 32 bit then an int variable will be an Int32. One thing that is neat about Swift is the ability to use underscores to make large numbers more readable. Not entirely sure why commas weren't an option for this.

Floating-Point Numbers

Not much to be said here, this is nearly the same as Objective-C. Both have Doubles and Floats. In Objective-c they are lowercase and Swift they are uppercase.

Numeric Literals

The only thing different here is you can define octals in Swift:

let octalInteger = 0o21 //17 in Octal

Numeric Type Conversion

Much like Objective-C , “Int” should be used for most cases you need an integer, types like “UInt8” and “Int32” should be used only when there is a special need.

Integer Conversion	
Swift	**Objective-C**
var someValue: Int = UInt(5) + Int(3500) **This is not allowed**	int someValue = (UInt8)5 + (int)3500; **But this is 3505**

Integer Conversion

Swift is a bit more strict when it comes to adding Integers of different types. In order to add them , they both need to be the same type whereas Objective-C is a little more forgiving.

Integer & Floating-Point Conversion	
Swift	**Objective-C**

`var intValue: Int = 8`
`var floatValue: Float = 0.675309`
`var sumOfFloatAndInt: Float =`
`intValue + floatValue`
This is not allowed
	int intValue = 8;
	float floatValue = 0.675309;
	float sumOfFloatAndInt = intValue +
	floatValue;
	But this is 8.675309

Integer and Floating-Point Conversion

Type Aliases	
Swift	**Objective-C**
`typealias AudioSample = UInt32`	typedef UInt32 AudioSample;

The same strictness with Integers applies to floats. In order to make this work in Swift you need to typecast “intValue” into a float.

Type Aliases

This is the same thing as a typedef in Objective-C. There’s really nothing much to it.

Booleans

Booleans are either “true” or “false” which translates to 1 or 0. The difference here is that there is no “Yes” or “No” like Objective-C.

Tuples

There is no direct parallel for a tuple in Objective-C, but imagine if you could mash an NSDictionary and an NSArray together and allow primitive types as well, then you would have something that resembled a tuple.

A simple example:

```
var dictionaryObject = NSDictionary(objectsAndKeys: "Key","Value")
var intPrimitive = 123
var stringPrimitive = "A String"
var someTuple = (123, "Not Found",dictionaryObject)

println("Tuple: \(someTuple)”)
//Which prints out -> Tuple: (123, Not Found, [Value: Key])
```

In this form a tuple resembles an NSArray, except you can add primitives to it like Int and String. And you can even access individual elements of the tuple as follows:

```
println("Tuple: \(someTuple.0)”)
//Which prints out -> Tuple: 123
```

If you would like more of a dictionary form, you can name the tuple elements. which will give you the functionality of both.

```
var someTuple = (anInt: 123, aString: "Not Found",aDict: dictionaryObject)
println("Tuple: \(someTuple.anInt)”)
//Which prints out -> Tuple: 123
```

These are all simple examples, but tuples can really be useful for functions and returning multiples values. With Objective-C you would need to return an NSArray or NSDictionary, which don’t hold primitives and don’t tell you anything about the types of the objects. In short, tuples can cut down on some work. Yay!

Optionals

Again, this is a swift only feature. This one should cut down a little more on work as it pertains to checking for nil values. As opposed to checking if a value is nil in Objective-c you can now use an optional that will tell you when either a value is present and provide it or there is no value at all.

The handy thing is this works for non-object types. So the absence of a value in an Integer is not simply 0. You can make it an optional.

As an example:

```
var anInteger: Int?
```

Set's an integer value with an optional that indicates the lack of a value. By default if you don't assign a value it will have no value. Alternatively you can assign a value and then assign it to nil to make remove the value.

```
var anInteger: Int? = 1234
anInteger = nil
```

To set anything to nil in Swift, the variable MUST be an optional.

Be careful with "nil" in swift, the definition has changed. In Objective-C nil indicates a pointer to a non-existant object. In swift if represents the absence of a value.

Also you'll need to add an exclamation point to the end of an optional to access the value if it's there. For example

```
if(anInteger != nil){
	println("Our integer value \(anInteger)") //Outputs -> "Our integer value Optional(1234)"
	println("Our integer value \(anInteger!)") //"Our integer value 1234"
}
```

Notice we need to also explicitly check an optional for nil value before accessing the value insid e it. For example without the check and "anInteger" was nil then

```
	println("Our integer value \(anInteger)") //This works and prints "Our integer value nil"
	println("Our integer value \(anInteger!)") //this errors EXC_BAD_INSTRUCTION
```

Error Handling

This was a big shortcoming of objective-C, previously there was an NSError and you could check for it and handle it, but it was all kind of a manual process.

```
func startCar() throws {
   // ...
}

do {
   try startCar()
   //Hey it worked, now do something as a result.
   driveCar()
} catch Error.OutOfGas {
   gasUpTheCart()
}
```

I won't get into all the ways errors can be handled, but notice here you can make a method that throws and error and then define certain error inside the startCar method.

With objective-C you typically had to pass an NSError back and constantly check for the existence of a non-nil error. The fact that swift now supports try-catch is a big deal.

Assertions

Assertions are nearly the same, just a little bit of syntax differences.

Type Aliases	
Swift	**Objective-C**
`assert(status == errStatus, "An error occured with error: \(status)")`	`NSAssert(status == errStatus, @"An error occured with error:: %d", (int)status);`

Basic Operators

Most of this is the same as Objective-C, here are things swift can do that Objective-C cannot:

Use the '+' to concatenate strings e.g..

```
"What" + "up" = "What up"
```

The nil coalescing operator

dogName != nil ? dogName! : dogDefaultName

Translated into english if the dogName optional is not empty then return the dog name, else return the default dog name.

Range Operators

Range operators aren't available in Objective-C, but swift provides two types.

Closed Range Operator

Defines a range from "a" to "b" that includes both.

```
for index in 1...5 {
   println("\(index) times 5 is \(index * 5)")
}
```

Half - Open Range Operator

Defines a range from “a” to “b” that does not include “b”.

```
let array = ["item1”, "item2", "item3", "item4"]
let count = array.count
for i in 0..<count {
    println(“array[i]”)
}
```

which will print all the values. In this example index i=count is never called.

Logical Operators

These are exactly what you would expect them to be from Objective-C

Strings and Characters

Strings and characters are more like standard C.

Strings	
Swift	**Objective-C**
String Literal	
`let aString = "A string literal"`	`NSString *aString = @"A string literal";`
Empty string	

`var emptyString = ""`
`var altEmptryString = String()`
	`NSString *emptyString = @"";`
	`NSString *altEmptyString = [NSString stringWithString:@""];`
	`Note: this is pointless as the string is immutable.`

String Mutability	

`var startString = "There’s a Mouse"`
`startString += "about the House."`
`Result: “There’s a Mouse about the House.”`
	`NSMutableString *startString = [NSMutableString alloc] initWithString:@"There's a Mouse";`
	`[startString appendString:@"about the House."];`

Swift strings are “Value Types”, which basically means they are copied when passed into a method or function. In Objective-C NSString is passed by reference and not copied.

Characters are not Objects in Objective-C but Strings are. So any manipulation of a string with characters requires NSString class methods.

Characters	
Swift	**Objective-C**
Defining Characters	

`let character: Character = ")"`	`char character = ‘)'; < – Actually`
	`just C`

Mutating Strings and Characters	

`let string = "ABC"`
`let character: Character = "D"`
 	
`var newString = ""`
`newString.append(character)`
	`NSString *string = @"ABC";`
	`char character = 'D';`
	
	`NSString *newString = [NSString`
	`stringWithFormat:@"%@`
	`%c",string,character];`

Mutating Characters with strings is done in much the same was as Objective-C. Although swift has the “append” method and Objective-C does not. Adding characters requires using methods in both languages. (Note: This seems to have changed between Swift 1.0 and 2.0. In 1.0 You could append a character simply with a ‘+’ or ‘+=’).