Swift, I fell in love. I found it just as easy to read while still having the Pizzazz that makes a language exciting to use. Objective-C on the other hand proved to have a bit of a learning curve because of the way it is generally structured. Let's compare a sample Objective-C class and a Swift class.

Objective-C

Writing classes in Objective-C can be a bit daunting. This is because Objective-C requires an implementation file and a header file for each class.

A header(.h) file

```
#import <Foundation/Foundation.h>
@class ThisTest;
@interface ThisTest : NSObject
@property (nonatomic) NSString *objectiveCString;
- (void) someMethod;
@end
```

An implementation (.m) file

```
#import "ThisTest.h"
@implementation ThisTest
NSString *objectiveCString = @"some string";
- (void) someMethod {
NSLog(@"This is how you print out");
}
@end
```

When I first started out, I found this to be quite intimidating. It also proved to be hectic, as I ended up obsessing about coordinating the naming of variables and methods, figuring what goes where and the whole business of importing.

Swift

With Swift, I realised I could throw everything into one file and concentrate more on logic. All I needed to do was, figure a way to create classes that grouped functionality together.

```
class ThisTest {
    var swiftString: String = "This is a Swift string"
    func someMethod() {
        print("This is how you print out")
    }
}
```

Are you sold? No? Well, Swift is generally more automated. You do not have to concern yourself with what goes on under the hood, memory management is taken care off already. This is not to pit one language against another. Objective-c is a great language, I just would not recommend that anyone who is looking to dip a toe into iOS start off with it.

Starting out with Swift

It took me approximately a month to get completely comfortable with the Swift syntax. We will discuss some of the things I discovered I needed to get to a place where I could comfortably contribute to projects. I recommend you start out with a playground. A playground is basically where you can test out anything. It is easy to use, you do not have to concern yourself with print statements to see whether your code is working. Simply open Xcode, press cmd+N , select playground and get your hands dirty.

Variables and Constants

In swift as in other languages, variables have to be declared before they are used. Variables are mutable and are declared with the keyword `var` while constants are immutable and are declared with the keyword `let` .

```
let someType: String = “I am explicitly declared”
var secretType = “You can infer what type I am”
```

The first one is a constant that has been explicitly declared as a string, the second example is a variable whose type has not been declared. The type is however inferred and the variable is known to contain a string. Some common types you will come across in Swift are `String`, `Bool`, `Int`, `Float` .

One of the most exciting things about Swift is **optionals**. Optionals are used where a variable may not contain a value.

```
var integer: Int = 9

var optionalInt: Int?
```

The first variable is an integer, it can never be nil. However, with the `optionalInt` the value can be nil. When it is first declared as an optional, it is nil by **default**.

Unwrapping Optionals

Optional variables have to be unwrapped before they are used. If you tried to perform an operation between `Integer` and `optionalInt` you would get an error that `optionalInt` is not unwrapped.

```
var optionalInt1: Int? = 10

var optionalInt2: Int? = 4

var sum = optionalInt1 + optionalInt2 //Buildtime error: Value of
optional type not unwrapped
```

There are two ways to unwrap optionals; optional binding and forced unwrapping.

Forced Unwrapping

```
var optionalInt1: Int? = 10

var optionalInt2: Int? = 4

var sum = optionalInt1! + optionalInt2!
```

The above example works, it is however not recommended. This is because whenever the optional is nil, the app crashes. Nobody likes a buggy app! Therefore, unless you are absolutely sure the value will never be nil, `never` use `forced unwrapping` . There is a better and prettier way to solve the unwrapping problem. Meet our friend, optional binding!

```
func add() -> Int? {

    let optionalInt: Int? = 10

    let someInt: Int = 8

    guard let unwrappedInt = optionalInt else { //optional binding
using guard statement

        return nil
    }
    return (unwrappedInt + someInt)
}
```

Collection Types

Collection types in swift are a way to group related items together. In swift, the values in the types are strongly typed. This means the compiler is aware of the types and you can therefore for example, not insert a string into an array of integers.

- Arrays
  These store items in an ordered list.

Creating, accessing and manipulating an array

```
var list1 = ["cookies", "cakes", "muffins"] //explicitly declare
contents

var list2 = [String]() //initializer

var list3: [String] = [] //type annotation

list1.count //returns 3, the number of items in the list
```

```
list1.append("cupcakes") //this adds the object

list1[0] //access the object on index(0); the first item

list1[1] = "1" //attempts to replace item on subscripted index, this
raises an error as it is a different type from the one expected

for item in list1 {
    print("\(item) are yummy")
} //prints each of the items and adds the "are yummy" at the end
```

There are numerous operations that can be performed on arrays. The swift documentation does a great job at explaining these.

- **Sets**
  A set is very similar to an array except that it stores distinct items in an unordered manner. Modifying and accessing a set is very similar to the same in an array.

```
var planets = Set<String>(),

var coolPlanets: Set<String> = ["Earth", "Pluto", "Saturn"]

var sadPlanets: Set = ["Uranus", "Pluto"]
```

- **Dictionaries**
  A dictionary is used to create key, value pairs.

Creating, accessing and manipulating a dictionary

```
var dictionary: [Int: String] = [:]           // type annotation

var dictionary2 = [Int: Int]()               // initializer

var swahiliDict = ["shoes": "viatu", "tea": "chai"]

swahiliDict["shoes"] //print the value associated with subscripted key
swahiliDict.count //print the number of key, value pairs

swahiliDict["shoes"] = "njumu" //replace the value at given key

swahiliDict["shoes"] = nil //remove the key as it no longer has a
value associated with it

for (key, value) in swahiliDict {
```

```
print("\(key) in Swahili is \(value)")

}

//will print "tea in Swahili is chai"
```

Control Flow

In swift, various statements are used to achieve control flow. These are statements that make decisions based on some conditions. Control flow statements include; those that perform tasks multiple times, those that iterate and those that perform certain tasks based on some conditions.

- **while and repeat-while**

```
var counter = 0

while counter < 5 {

    print("Counter is at \(counter)")

    counter += 1
}
//this loop executes until the given condition is false. In this case,
it executes while counter is <5
```

- **if and if-else, guard, switch**

if-else: It executes a certain block only if the condition is true. In Swift, the condition to be evaluated has to be either `true` or `false` unlike in Objective-C.

```
var time = 1

if time > 4 { //this returns true or false

    print("I am starving!")

} else {
    print("i could eat!") //this block is executed
}
```

Guard: it is used to transfer control. The conditions should evaluate to true or false.

```
func printEvenPostiveNumber(_ num: Int) {
    guard num > 0 else { print("\(num) is not > 0")
        return
    }
    guard num % 2 == 0 else { print("\(num) is not even")
        return
    }
}
printEvenPostiveNumber(-1) // "-1 is not > 0\n"
printEvenPostiveNumber(3) // "3 is not even\n"
```

Switch: this compares a value with several matching patterns then executes a block based on which it matches first.

```
let temperature: Int = 20
switch temperature {
case 1..<15:
    print("It's sweater weather")
case 15..<30:
    print("Less Monday more summer")
default:
    print("We don't recognise what is happening here")
}
```

for-in: these are used to loop over collection types or a range of numbers

```
for values in 1...5 {

    print (values)

} //this prints all the values between 1 and 5
```

Functions

Functions are chunks of code that perform a task.

In the spirit of writing self-documenting code. It is important to give a function a name according to what it does. One of the things I came to appreciate when I was getting into Objective-C is developers who give functions accurate names, these men and women are the heroes who don't wear capes.

```
func basic() {}
basic()
//this function has no arguments

func withParameter(parameter: Int) {}
withParameter(parameter: 3)
//has an argument that matches the parameter name

func withReturn() -> Int { return 2 }
withReturn()
//returns a value 2, you can ignore it, if you do not write a
//return, it raises an error as the function expects a return

func withMultipleParameter(parameter1: Int, parameter2: Int) {}
withMultipleParameter(parameter1: 3, parameter2: 5)

func returnsTuple() -> (Int, Int) { return (2,3) }
returnsTuple()
//return (2,3)

func drawLine(from point1: Int, to point2: Int) {}
drawLine(from: 3, to: 6)

func optionalParameterName(_ optionalParameterName: Int,
justParameter: Int) {}
optionalParameterName(3, justParameter: 5)
//a _ before the parameter label tells the compiler that we do not
//want to write the argument label when calling the function
```

There is special type of functions known as **closures.** Closures are fun, closures will make you a bad ass. I recommend swift documentation on this.

Structs and Classes

If you have some previous experience with programming, you have definitely come across classes. Classes are ideally the same across the board. However, in swift there is this sweet little thing we call structures. Structures are ideally similar to classes save for a neat little difference. Classes are reference types while structures are value types. One of the things I suffered through when I was trying to navigate swift was the difference between these two, where to use them and the advantage of one over the other. This post made things so much easier. Here is an example that will hopefully clear out the differences.

- **Reference Types**
  An object in a class is a reference type.

```
class ClassObject { var a = "Hey" }

var object1 = ClassObject()

var object2 = object1
//creates a shared instance, both variables refer to the same
//instance

object1.a = "There!"

print(object1.a, object2.a) //prints "There There"
```

- **Value Types**
  An object in a structure on the other hand is a value type.

```
struct StructObject { var a = "Hey" }

var object1 = StructObject()
```

```
//data
object2.a = "There!"
print(object1.a, object2.a) //prints "hey there"
```

Other than this difference, structures and classes are essentially the same.

```
class Example { //give the class a meaningful name
    var subject: String = "Structures" //class property
    func giveDefinition() { //class method
        print("This is the definition for \(subject)")
    }
}
var example = Example() //instantiating a class
example.giveDefinition() //calling the class method
```

Classes and structures have variables or constants that are referred to as properties and functions that are referred to as method. If you do not give an initial value for properties in the classes, you will get an error.

```
class Example { var subject: String
    init() { subject = "swift" }
    func giveDefinition() {
        print("This is the definition for \(subject)")
    }
}
var example = Example()
example.giveDefinition()
```

Extensions

Sometimes you have classes that have some functionality in only special cases. Say you have a class that draws a chart. This class may be used in several places in your app. However, you may need it to have special divisions in some cases. It does not make sense to add such functionality globally yet it will be applicable in limited places. Enter extensions; these add computed properties, methods, initializers or protocols. We have not spoken about protocols I know. Worry not, protocols is another super power that swift possesses that we will look at shortly.

```
class BareMinimum {}

extension BareMinimum { //add functionality to the above class}
```

Extensions are very exciting to study and work with. You could take a deep dive and discover all the cool things you can do with extensions.

Enumerations

These group related values so that you are able to work with them in a type safe way.

```
enum Color { case blue, green, yellow, red, black}
var colorToWear = Color.blue
colorToWear = .red
//after first assigning it, you can use dot syntax to access other
//values
```

Matching enumerations with a switch statement

```
enum Color { case blue, green, yellow, red }

var colorToWear = Color.blue

colorToWear = .red

switch colorToWear {
```

```
case .blue:
    print("someone's not feeling too good")
case .yellow:

    print("It's celebration time")
case .green:

    print("we have a sunflower!")
case .red:

    print("It's a siren")
default:

    print("Sorry, who are you?")

}
```

Access Control

Access control restricts access to parts of your code from code in other source files and modules.

A module is defined as a single unit block of code that is built and shipped as a single unit. It can be imported by another module. A classic example you could come across is `UIKit` . This deals with all things user interface(UI). When working with the UI, you import it and use it’s various functionalities. Sometimes, you want to restrict control to some parts of your code. This is achieved using access control. There are various levels of access control;

- **Open:** this is the least control. Entities can be used in defining module and in any module that imports the file. These entities can be subclassed and overriden as well. This is typically used for frameworks.
- **Public:** this is essentially the same as open control except that the module cannot be overriden/subclassed by the module that imports it.
- **Internal:** this is the default access level. Entities are only used inside the defining module.

- **File-Private:** used when an entity is to be used within one single swift file
- **Private:** the entity can only be used inside the defining method, class or function

An important thing to note however, an **entity cannot be defined in terms of another with a more restrictive level**. This answer perfectly explains what this means.

Protocols

You might have come across Object Oriented Programming(OOP) in other languages. In swift we have OOP, which is generally used as in other languages. However, because swift just won't quit delighting us, we have Protocol Oriented Programming. Here is a great differentiation of the two.

A protocol is a blueprint for how things are to be done. Classes, structures and enumerations can conform to protocols. This means they have to implement the functionality given in the protocol.

```
protocol thisProtocol { var someName: String { get } }
//this protocol specifies a variable

struct thisStruct: thisProtocol { var someName: String }
//the struct has to have the variable that was specified in the
//protocol
```

If `var someName { get }` makes you shake in your boots, worry not. There is a great explanation on **stored** and **computed properties** here.

Xcode

Now you have the basic knowledge of Swift in your tool-belt, what next? I would recommend you get familiarised with Xcode. Just like all the things Swift, Xcode is graceful, friendly and elegant. However, if you do not familiarise yourself with it, you may find yourself greatly frustrated by the numerous tabs, the warnings and the occasional crashes. Here are a couple of things I have found helpful while working with Xcode;