**Closures in Swift**

I have been learning Swift for the last nine months. To say I am having a blast is an understatement. However, whenever I am having a swift party, there is a cop car down the street waiting to shut it down. Oh closures, who hurt you? So I finally decided to learn closures and thought to document my **struggles**, **learnings** and finally **Aha moments**. Most of these will be things I have learnt from different sources but found myself wishing I had found in one place. These include but are not limited to [Bob the Dev’s blog](https://www.bobthedeveloper.io/blog" \t "_blank) , [swift documentation](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html" \t "_blank)and [this](https://medium.com/@abhimuralidharan/functional-swift-all-about-closures-310bc8af31dd" \t "_blank) amazing piece on Medium.

**Why the name?**

Closures can capture and store reference to any constants and variables from the context in which they are defined. This is known as closing over them, hence the name closures💡.

**You look familiar, where have I seen you before?**

Closures are in so many places due to their clean and optimised syntax. You may have come across them when;

* Using animations; this is because some functions are performed asynchronously. This is an amazing opportunity to use escaping closures.
* Fetching data from third party API; this is also another great opportunity to use escaping closures.
* Passing data between view controllers

A number of Swift’s in-built functions require closure. Here is one;

**Functions to closures**

Apple defines closures as self contained blocks of functionality that are passed around. Before we get too technical we could see how to transform a function into a closure.

Here is a simple function I am using to find out how long I have to learn closures. It takes in a two parameters, days and namethen returns a string that tells me how many days I have remaining. Classy huh? 😃

To convert it to a closure;

* Remove curly braces

Of course if you attempt to run the above snippet as is, you would get an error. This is something in between a closure and a function, maybe we could call it a funlosure. No? Okay! 😄

* Add in keyword between argument list and body i.e between String and return

This still has an error.

* Although this is a bit of a controversial statement with a lot of views and opinions, closures are essentially headless functions. We therefore take out the head 😈. In this case the head isfunc learnClosures

Still does not run??

* Surround it with the curly braces

Almost there 🤞. This is what is called a closure expression in it’s most **verbose** form. As we will see later on, you can strip it down to one line.

* Now, we essentially have declared a closure but do not have a way to call it. Solution? we assign it to a variable. This is essentially possible because closures are referred to as **first class.**This is also true for **functions**.

Here we have a closure in it’s complete form.

**Defining closures**

As with functions, closures take many forms;

* **Closures that do not take any parameters and do not return anything**

In Swift, void is the same as (). As a rule, you have to specify a **return** **type**for closures. The above closure takes in **no parameter** and **returns nothing**.

We can simplify this to be **less verbose**;

What have we done?

1. We take out the **type** from the declaration, this is because the closure can infer the type from context. Then we go ahead and remove the return type. This is why it is said **closure expressions can infer parameter and return type from context**.
2. We have also taken out the in keyword, this is because with the removal of the parameters, we have no **head** to separate from the **body**

* **Closures that takes in a parameter and return nothing**

The closure takes in **one parameter** string and returns nothing.

Again, we can strip this down to

What have we done?

1. We remove the argument name and use one that Swift automatically generates
2. Each argument is assigned a name in Swift. Say we have three parameters, let’s call them (a,b,c). Swift would assign them ($0, $1, $2).
3. This is another amsing ability of closure expressions that is known as assigning **Shorthand Argument Names.**

* **Closures that take in parameters and return stuff**

The closure takes in **one parameter oneString**of type**string**and**returns a**string.

You know the drill, ain’t nobody got time to be typing such verbose closures!!!

**Calling closures**

You can use closures in functions

**Returning a closure from a function**

We will look at two examples;

1. Using a **closure expression**

2. Using a **nested function**; as we will see, nested functions are a special **case**of closures

**Things to note**

Closures exist in 3 main forms

* **Global functions**; these are closures with a name that do not capture their surrounding values
* **Nested function**s; closures with a name that capture values of their enclosing function
* **closure expressions**; unnamed closures that capture values of their surrounding context

**Characteristics of closures**

* Like functions**, closure parameters are constants and cannot be modified,** unless marked as i**n-out.**

However, as in functions you can mark it as in-out and work your way around that error.

Nice and pretty

* Closures capture the variables of the surrounding context

Here we will refer to the nested function example.

Looking at the above snippets, they may justify the need for closures. In the first example, when we use a nested function nee closure, it captures the value of the variable a and keeps using it long after it’s defining scope is over. That is why whenever we call it, it already has a value and increments it from there.

Using a simple function, however, it loses the value of the variable and we start over every time we call the function.

* Closures are reference type

**Flavours/Types of functions**

* **Trailling closures**

These are used when you have a closure as the last parameter in a function and is too long to be used as an inline closure.

**Escaping closures**

Escaping closures have the word @escaping before them while non-escaping closures do not. 😆 Not a good enough definition? let me take another shot at it by explaining their lifecycle.

In an **escaping closure**, the lifecycle looks like so;

1. pass closure in function argument
2. Do some task in function
3. return compiler
4. run closure asynchronously

This means that the closure outlives the function i.e it is called after the function has returned. An example of this is a **completion handler**; a closure that is run only after a certain task has been completed.

**Use cases;**

Escaping closures are used in **asynchronous programing.**You may want to look at multithreading if you are not aware of it in [this](https://www.raywenderlich.com/148513/grand-central-dispatch-tutorial-swift-3-part-1" \t "_blank) awesome tutorial. In multithreading, we use multiple queues. When an escaping closure is used the queue holds onto the closure and only runs it when the queue is done.

Escaping closures are also is also used when accessing global variables. This is because the variables outlive the closure.

In the above snippet, we have a closure that modifies a global variable. It should have access to this variable that is not within the scope of it’s calling function and hence needs to be escaping.

*Before Swift3, all closures were escaping unless explicitly stated to be noescape . However, due to memory management, you now have to state when you want the closure to be escaping.*

LifeCycle of **non-escaping closure;**

1. Pass closure in functions arguement
2. Do some work in function
3. Run closure
4. Return compiler back

As you will observe, a non-escaping closure does not outlive it’s calling function.

**Conclusion**

1. So far, I have used closures when doing animations, using multithreading and fetching data from external API’s. I am looking forward to using them for customising views say buttons, writing cleaner code in functions and most importantly in binding components while using **MVVM.**

2. Great! We know all these things about closures, **next steps**?

* When learning about closures, I came upon **memory management** as the main reason we use escaping closures. I would recommend the [apple documentation](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/MemorySafety.html" \l "//apple_ref/doc/uid/TP40014097-CH46-ID567" \t "_blank) for learning a bit more on that.
* Personally, I have found that closures do not come naturally and I imagine it’s the same for many other developers. I would advise you to get into the habit of reading other seasoned developers’ code to see how they use closures.
* It would be helpful to look at inbuilt Swift functions that have closures when you come across them.