*****INTERVIEW QUESTIONS******

==========

Q1) What are the collection types that are available in swift?
Answer: There are three primary collection types that are available in swift for storing a collection of values. They are dictionaries, sets, and arrays
Arrays: Arrays is an ordered collection of values, which is stored in the same type of values in an ordered list.
Sets: Sets are an unordered collection of unique values, which are stored in a distinct value of the same type in a collection without any defined ordering.
Dictionaries: Dictionaries are an unordered collection of Key and value pair associations in an unordered manner.

==========

Q2) What is “defer”?
Answer: The “defer” is a keyword that provides a block of code that can be executed while the execution is leaving the current scope.

==========

Q3) What is Tuple? How to create a Tuple in swift?
Answer: A tuple is a group of different values in a single compound value. It is an ordered list of elements. There are two ways of accessing the object data in a tuple i.e. by name or by position.
In swift, a tuple can consist of multiple different types. It can support two values i.e. one of integer type, and the other of a string type. It is a legal command.

For Example: let ImplementationError = (501, “Not implemented”).
We can create a basic tuple like this:
let person = (name: “Ajay” , age: 34)

==========

Q #4) What is the difference between class and structure?
Answer: Classes are reference types, whereas structs are value types. Classes can be built on other classes, whereas a struct cannot inherit from another struct. Classes have an inheritance, whereas structs cannot have an inheritance.

In class, we can create an instance with “let” keywords and attempt to mutate its property, whereas there is no Mutability in Structs. Classes have Type Casting, whereas struct doesn’t have Type Casting.

==========

Q #5) What is Optional chaining?
Answer: Optional chaining is a useful process which we can use in combination with the optional to call the methods, properties, and subscripts on the optionals and these values may or may not be nil. In this process, we may try to retrieve a value from a chain of the optional values.
If the optional contains a value, then calling the subscript, method or property of an optional will return a value. If optional is nil, then calling the subscript, method and property will return nil.

==========

Q #6) What is Optional binding?
Answer: Optional Binding concept is used to find out whether an optional contains a value, and it makes that value available as a variable or temporary constant. We use an optional binding concept to check if the optional contains a value or not. Optional binding can be used with the condition (if and while) statements to check for a value inside an optional.

==========

Q7). **Explain the frame and bound of a View in ios?**
Answer: Bound is the rectangle that is relative to its own coordinate system (0,0). Frame is the rectangle that is relative to the superview .
frame = a view's location and size using the **parent view's coordinate system**
Important for: placing the view in the parent
bounds = a view's location and size using **its own coordinate system**
Important for: placing the view's content or subviews within itself
A view tracks its size and location using two rectangles: a frame rectangle and a bounds rectangle. The frame rectangle defines the view's location and size in the superview using the superview’s coordinate system. The bounds rectangle defines the interior coordinate system that is used when drawing the contents of the view, including the origin and scaling.

==========

Q8). **What is a protocol in Swift?**
Answer: A protocol is a type that defines a blueprint of methods, properties and other requirements. A class, structure or enumeration can then adopt the protocol to implement those requirements. A type that adopts the requirements of a protocol conforms to that protocol. The protocol doesn't implement any functionality itself, but rather defines the functionality. You can extend a protocol to provide a default implementation of some of the requirements or additional functionality that conforming types can take advantage of.

==========

Q9). **Difference between Keychain and NSUserDefaults?**
Answer: In Keychain: If a user removes the app from the device the saved UserName and Password still is there.
In NSUserDefaults: If a user removed the app from the device the saved UserName and Password also removed.

==========

Q10). What is the difference between Non-Escaping and Escaping Closures?
Answer: Lifecycle of the @escaping closure:

1. Pass the closure as a function argument, during the function call.
2. Do some additional work in function.
3. Function executes the closure asynchronously or stored.
4. Function returns the compiler back.

Escaping closure means, inside the function, you can still run the closure (or not); the extra bit of the closure is stored someplace that will outlive the function. There are several ways to have a closure escape its containing function:
Asynchronous execution: If you execute the closure asynchronously on a dispatch queue, the queue will hold onto the closure for you. You have no idea when the closure will be executed and there’s no guarantee it will complete before the function returns.
Storage: Storing the closure to a global variable, property, or any other bit of storage that lives on past the function call means the closure has also escaped.
1. @nonescaping closures:
When passing a closure as the function argument, the closure gets executed with the function’s body and returns the compiler back. As the execution ends, the passed closure goes out of scope and has no more existence in memory.
Lifecycle of the @nonescaping closure:

1. Pass the closure as a function argument, during the function call.

2. Do some additional work with function.
3. Function runs the closure.
4. Function returns the compiler back.

Example:

```
func getSumOf(array:[Int], handler: ((Int)->Void)) {
        //step 2
        var sum: Int = 0
        for value in array {
            sum += value
        }

        //step 3
        handler(sum)
}
```

==========

Q11). Explain [weak self] and [unowned self] ?

Answer: unowned (non-strong reference) does the same as weak with one exception: The variable will not become nil and must not be optional. When you try to access the variable after its instance has been deallocated. That means, you should only use **unowned** when you are sure that this variable will never be accessed after the corresponding instance has been deallocated. However, if you don't want the variable to be weak AND you are sure that it can't be accessed after the corresponding instance has been deallocated, you can use unowned. Every time used with non-optional types

Every time used with let By declaring it **[weak self]** you get to handle the case that it might be nil inside the closure at some point and therefore the variable must be an optional. A case for using **[weak self]** in an asynchronous network request, is in a **view controller** where that request is used to populate the view.

1. [weak self] can't prevent completion calls or execution of statements inside the completion block, it's going to perform those statements only if self is not nil.
2. Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime. Conversely, use an unowned reference when you know that the reference will never be nil once it has been set during initialization

==========

Q12). Explain Forced Unwrapping

Answer: When we define a variable as **optional**, then to get the value from this variable, we will have to **unwrap** it. This just means putting an exclamation mark at the end of the variable. The example of the implicitly unwrapped optional type is the IBOutlets we created for your view controller.
We have to use **Forced Unwrapping** when we know an optional has a value.

==========

Q13). What is Concurrency ?
Answer: Concurrency is dividing up the execution paths of your program so that they are possibly running at the same time. The common terminology: process, thread, multithreading, and others. Terminology;

- **Process,** An instance of an executing app
- **Thread,** Path of execution for code
- **Multithreading,** Multiple threads or multiple paths of execution running at the same time.
- **Concurrency,** Execute multiple tasks at the same time in a scalable manner.
- **Queues** are lightweight data structures that manage objects in the order of First-in, First-out (FIFO).
- **Synchronous** vs **Asynchronous** task

==========

Q14). Grand Central Dispatch (GCD)
Answer: GCD is a library that provides a low-level and object-based API to run tasks concurrently while managing threads behind the scenes. Terminology;

- **Dispatch Queues,** A dispatch queue is responsible for executing a task in the first-in, first-out order.
- **Serial Dispatch Queue** A serial dispatch queue runs tasks one at a time.
- **Concurrent Dispatch Queue** A concurrent dispatch queue runs as many tasks as it can without waiting for the started tasks to finish.
- **Main Dispatch Queue** A globally available serial queue that executes tasks on the application's main thread.

==========

Q15). What is the difference between Filter and Map Function?
Answer: Map, we pass in a function that returns a value for each element in an array. The return value of this function represents what an element becomes in our new array.

Filter, we pass in a function that returns either true or false for each element. If the function that we pass returns true for a given element, then the element is included in the final array.

==========

Q16). Explain generics in Swift ?

Answer: Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA= a
    a = b
    b = temporaryA
}
```

==========

Q17). Map vs. FlatMap vs. CompactMap

Answer: Map returns an array containing the results of mapping the given closure over the sequence's elements. So, let's imagine that you need to map an array of integers into an array of strings.

Example: let arrayOfIntegers = [1, 2, 3, 4]

```
let arrayOfStrings = arrayOfIntegers.map { String($0) }
print(arrayOfStrings)
```

compactMap cleans the sequence, therefore, you receive a non-optional sequence of integers map transforms it into strings

```
let arrayOfIntegers = [1, 2, nil, 4]
let arrayOfStrings = arrayOfIntegers.compactMap { $0 }.map(String.init)
```

compactMap helps you to eliminate nil values and map at the same time. It takes your sequence and produces a fancier sequence. Easy as it is

FlatMap: flatMap returns an array containing the concatenated results of calling the given transformation with each element of this sequence.

```
let arrayOfIntegers = [[1, 3, 5, nil], [2, nil, 6]]
let arrayOfStrings = arrayOfIntegers.flatMap { $0 }.compactMap { $0 }
```

- flatMap flattens down your array. After applying .flatMap { $0 }, you receive "[Optional(1), Optional(3), Optional(5), nil, Optional(2), nil, Optional(6)]".

- compactMap removes the nil values, thus making your final result [1, 3, 5, 2, 6].
- flatMap returns an array containing the concatenated results of calling the given transformation with each element of this sequence.
    - If you need to simply transform a value to another value, then use map.
    - If you need to remove nil values, then use compactMap.
    - If you need to flatten your result one level down, then use flatMap.

==========

Q18). What Is The Difference Between Swift And Objective-c Language?

Answer: Swift Programming-

- In swift programming, the variables and constants are declared before use.
- In swift programming, “var” keyword is used for variables and the “let” keyword for constant.
- In swift programming, no need to end code with semicolon
- Swift programming does not require creating a separate interface like Objective-C.
- In swift programming, we can define methods in class, structure or enumeration.

Objective-C Programming-

- In objective-C programming, we need to end code with semicolon
- In objective-C programming, we can declare constants as int and variable as NSString.

==========

Q19). Array vs Set

Answer: A set stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items is not important, or when you need to ensure that an item only appears once.

- Array: “An ordered, random-access collection.”
- Set: “An unordered collection of unique elements.”
- A big difference between Sets and Arrays is the uniqueness of elements. An Array can contain the same value twice while a Set will never contain duplicates
- Testing whether an object is contained in a set is faster than testing for membership of an array.

- Another reason to go for a set is when performance is important or when a large amount of data is expected. Elements in a set need to conform to the Hash-able protocol which makes a set optimised for performance. It takes the same amount of time to lookup an element in a small collection vs a large collection Performance is not important in an array while Performance is an important set.

==========

Q20). What is the difference between ANY and ANY-OBJECT ?
Answer: According to Apple's Swift documentation:

- **Any** can represent an instance of any type at all, including function types and optional types
- **AnyObject** can represent an instance of any class type.

==========

Q21). Explain the Final keyword into the class?
Answer: By adding the keyword final in front of the method name, we prevent the method from being overridden. If we can replace the final class keyword with a single word static and get the same behaviour.
==========

Q22). What is the difference between Open & Public access level?
Answer: open allows other modules to use the class and inherit the class; for members, it allows others modules to use the member and override it.
public only allows other modules to use the public classes and the public members. Public classes can no longer be subclassed, nor public members can be overridden.
==========

Q23). Stored Property vs Computed property?
Answer: Stored properties store constant and variable values as part of an instance, whereas **computed** properties calculate (rather than store) a value.

- Computed properties are provided by **classes**, **structures**, and **enumerations**.
- Stored properties are provided only by **classes** and **structures**.

The benefit of type properties

- You have access to them without needing to create a new object of that specific type.

- These type properties area available to any instance of that particular type

A **stored** property is a constant or variable that is stored as part of an instance of a particular class or structure. Stored properties can be either variable stored properties (introduced by the `var keyword`) or constant stored properties.

Computed In addition to stored properties, classes, structures, and enumerations can define **computed properties**, which do not actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

==========

Q24). What are Property Observers?
Answer: Property observers observe and respond to changes in a property's value. Property observers are called every time a property's value is set, even if the new value is the same as the property's current value. You can add property observers to any stored properties you define, except for lazy stored properties. We can add property observers to 'inherited' property by method 'overriding'.
You have the option to define either or both of these observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the New value is stored.

```
class StepCounter {
     var totalSteps: Int = 0 {
       willSet(newTotalSteps) {
           print("About to set totalSteps to \(newTotalSteps)")
       }
       didSet {
           if totalSteps > oldValue  {
               print("Added \(totalSteps - oldValue) steps")
           }
       }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
```

```
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

==========

Q25). Explain View Controller Lifecycle events order?

Answer: There are a few different lifecycle events

- **loadView :-** Creates the view that the controller manages. It's only called when the view controller is created and only when done programmatically. It is responsible for making the view property exist in the first place.
- **viewDidLoad :-** Called after the controller's view is loaded into memory. It's only called when the view is created.
- **viewWillAppear :-** It's called whenever the view is presented on the screen. In this step the view has bounds defined but the orientation is not applied.
- **viewWillLayoutSubviews :-** Called to notify the view controller that its view is about to layout its subviews. This method is called every time the frame changes
- **viewDidLayoutSubviews :-** Called to notify the view controller that its view has just laid out its subviews. Make additional changes here after the view lays out its subviews.
- **viewDidAppear :-** Notifies the view controller that its view was added to a view hierarchy.
- **viewWillDisappear :-** Before the transition to the next view controller happens and the origin view controller gets removed from screen, this method gets called.
- **viewDidDisappear :-** After a view controller gets removed from the screen, this method gets called. You usually override this method to stop tasks that should not run while a view controller is not on screen.
- **viewWillTransition(to:with:) :-** When the interface orientation changes, UIKit calls this method on the window's root view controller before the size changes are about to be made. The root view controller then notifies its child view controllers, propagating the message throughout the view controller hierarchy.

==========

Q26. What is the difference between Synchronous & Asynchronous tasks ?

Answer : Synchronous: waits until the task has completed

Asynchronous: completes a task in background and can notify you when complete

- The former method, performBlock, is the one that is **asynchronous**. That means that it can run at any time and the function that called it can continue without needing to know whether the block was successful or not. The main thread can continue even if the block is still or hasn't even started running.

- The latter method, performBlockAndWait, is **synchronous**. The main thread will stop and wait for a response from the block before it will continue with the next operation.

==========

Q27. Explain SOLID Design Principle using Swift?

Answer: Concepts

1. **Single responsibility principle:** a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).
2. **Open/closed principle:** software entities should be open for extension, but closed for modification.
3. **Liskov substitution principle:** objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
4. **Interface segregation principle:** many client-specific interfaces are better than one general-purpose interface.
5. **Dependency inversion principle:** one should "depend upon abstractions, [not] concretions."

Q28. What is active & passive in MVVM using Swift ?

Answer: In MVVM, the view is **active**. As opposed to a passive view which has no knowledge of the model and is completely manipulated by a controller/presenter, the view in MVVM contains behaviors, events, and data-bindings that ultimately require knowledge of the underlying model and viewmodel.

Interview preparations - technical questions

Core data
Multithreading using GCD and OperationQueue
Api Calling using NSURLSession
SSL Pinning
Push notification
Constraint and size classes
Difference between leading and left constraint
App architecture
Design patterns

Library and framework
Cocoa pods
Swift UI
ARC
Memory management
Access control
Properties
Generics
Protocols – Hashable, Equitable
What's new in iOS, swift, Xcode – Current version of swift, Xcode and iOS
App slicing, bit code, app thinning
Race condition and deadlock
Shared class
Unit and UI test – mock data
Accessibility
Git/ Bitbucket – Naming branch, master release hotfix branch concept
SOLID
Dependency management -
https://cocoacasts.com/nuts-and-bolts-of-dependency-injection-in-swift

Clean architecture
Any and any object
Localizations
Singleton : https://cocoacasts.com/are-singletons-bad
https://www.swiftbysundell.com/articles/avoiding-singletons-in-swift/

TDD :
https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/
https://clean-swift.com/step-by-step-walkthrough-of-ios-test-driven-development-in-swift/