1. **Explain the different types of iOS Application States.**

You will experience a few app states while using the application. There are five states in the iOS application as follows.

* **Not Running**

The application is in the Not running state, if the app is not launched yet or if the application is not visible on the screen or also there is the possibility that the application is terminated by the user or by OS.

* **Inactive**

An inactive state occurs when the application is not receiving events but the app is running in the background. An app transitions to a different state. when the user locks the screen or the system prompts the user to respond to some event such as a phone call or SMS message is the only time it stays inactive.

* **Active**

In an iOS application, Active State is the main executive state. In this state, the app is running in the foreground and the UI is accessible.

* **Background**

Before being suspended most of all apps enter this state.  If an app requests extra execution time then that will remain in this state for some more time. Also, an application will be launched directly into the background enters this background state instead of the inactive state

* **Suspended**

In a suspended state the application does not execute any code. The system purges suspended apps without any notice to make more space for the foreground app when a low-memory condition occurs.

### ****Differentiate between a frame and a bound?****

A UIView’s bounds are a rectangle with a size (width, height) and position (x,y) relative to its own coordinate system (0,0).

A UIView’s frame is a rectangle with a scale (width, height) and position (x,y) relative to the super view it is located within.

### ****Explain the lazy property in Swift?****

A lazy stored property is one that does not determine its initial value until it is used for the first time. The lazy modifier is written before the declaration of a lazy stored property.

### ****What is the use of deinit in swift?****

Deinitialization in Swift is the method of deallocating or cleaning up unused class instance objects in order to free up memory space used by objects for better memory management.

### ****What is the difference between an ‘App ID’ and a ‘Bundle ID’? What is each used for?****

* An App ID is a two-part string used to identify one or more apps from a single development team. The string consists of a Team ID and a Bundle ID search strings, with a period (.) separating the two.
* The Team ID is supplied by Apple and is unique to a specific development team, while a Bundle ID is supplied by the developer to match either the Bundle ID of a single app or a set of Bundle IDs of a group of apps.

Since most users consider the App ID as a string, they think it is interchangeable with the Bundle ID. Once the App ID is created in the Member Centre, we can only use the App ID prefix that matches the Bundle ID of the application bundle.

The Bundle ID uniquely defines each app. It is specified in Xcode. A single Xcode project can have multiple targets and, therefore, outputs multiple apps. A common use case: an app having both lite/free and pro/full versions or branded multiple ways.

1. **Explain MVVM**

UIKit independent representation of your View and its state. The View Model invokes changes in the Model and updates itself with the updated Model, and since we have a binding between the View and the View Model, the first is updated accordingly.

Your view model will actually take in your model, and it can format the information that’s going to be displayed on your view.

1. **Explain what is defer ?**

Defer keyword which provides a block of code that will be executed in the case when execution is leaving the current scope.

1. **Explain Guard statement**

There are two big benefits to guard statement.

The first benefit is providing an early exit out of the function using break or using return. The last benefit, guard statement is another way to safely unwrap optional.

1. **How is memory management handled in iOS ?**

Swift uses Automatic Reference Counting(ARC). This is conceptually the same thing in Swift as it is in Objective-C. ARC keeps track of strong references to instances of classes and increases or decreases their reference count accordingly when you assign or unassign instances of classes (reference types) to constants, properties, and variables. It deallocates memory used by objects whose reference count dropped to zero. ARC does not increase or decrease the reference count of value types because, when assigned, these are copied. By default, if you don’t specify otherwise, all the references will be strong references.

1. **What do you know about Singletons ? Where would you use one and where would you not ?**

Singletons are sometimes considered to be an anti-pattern. There are multiple disadvantages to using singletons. The main ones are global state, object life-cycle, and dependency injection. When you have only one instance of something, it’s very tempting to reference and use it everywhere directly instead of injecting it into your objects. That leads to unnecessary coupling of concrete implementation in your code instead of working with an interface abstraction.

Another malicious side effect of “convenient” singletons is the global state. Often singletons enable global state sharing and play the role of a “public bag” that every object uses to store state. That leads to unpredictable results and bugs or crashes when this uncontrolled state gets overridden or removed by someone.

1. **Explain and show examples of SOLID principles ?**

**The Single Responsibility Principle (SRP)** is the most important principle of the group. It states that every module should have only one responsibility and reason to change. SRP starts with small concrete and specific cases such as a class and/or an object having only one purpose and being used only for only one thing.

**The Open/Closed Principle (OCP)** states that your modules should be open for extension but closed for modification. It’s one of those things that sounds easy enough but is kind of hard to wrap your head around when you start to think about what it means. Effectively it means that when writing your code you should be able to extend the behaviour of your objects through inheritance, polymorphism, and composition by implementing them using interfaces, abstractions, and dependency injection.

**The Liskov Substitution Principle (LSP)** states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. What that means is that when you inherit from a class or an abstract class or implement an interface (protocol), your objects should be replaceable and injectable wherever that interface or class that you subclassed from was used. This principle is often referred to as design by contract or, as of late in the Swift community, referred to as protocol-oriented programming. The main message of this principle is that you should not violate the contract that your interfaces you subclass from promise to fulfil, and that by subclassing, those subclasses could be used anywhere that the superclass was previously used.

**The Interface Segregation Principle (ISP)** says many client-specific interfaces are better than one general-purpose interface. It also states that no client should be forced to depend on and implemented methods it does not use. What that means is that when you create interfaces (protocols) that your classes implement, you should strive for and depend on abstraction over specificity, but not until it becomes a waste where you have to implement a bunch of methods your new class doesn’t even use.

**The Dependency Inversion Principle (DIP)** states, “depend on abstractions, not concretions.” The best example that showcases this principle is the Dependency Injection (DI) technique. With the Dependency Injection technique, when you create an object, you supply and inject all of its dependencies upon its initialization or configuration rather than let the object create or fetch/find its dependencies for itself.

1. **Difference between flatMap and compatctMap?**

**compactMap:**

Use this method to receive an array of nonoptional values when your transformation produces an optional value.

Imagine that our array constant has some nil values and you do not want them to be mapped to the corresponding string, but rather, ignore them.

**flatMap:**

Use this method to receive a single-level collection when your transformation produces a sequence or collection for each element.

It will convert the 2D array into single array.

1. **What is Tuples**

Tuples are **anonymous structs** that can be used in many ways, and one of them is to make **returning multiple values from a function** much easier.

The advantages of using a tuple instead of an array are:

* multiple types can be stored in a tuple, whereas in an array you are restricted to one type only (unless you use [AnyObject])
* fixed number of values: you cannot pass less or more parameters than expected, whereas in an array you can put any number of arguments
* strongly typed: if parameters of different types are passed in the wrong positions, the compiler will detect that, whereas using an array that won't happen

named: it's possible to associate a name with each parameter

1. **The Migration Process for Core Data**

To start the migration process, Core Data needs the original data model and the destination model. It uses these two versions to load or create a mapping model for the migration, which it uses to convert data in the original store to data that it can store in the new store. Once Core Data determines the mapping model, the migration process can start in earnest.

Migrations happen in three steps:

* First, Core Data copies over all the objects from one data store to the next.
* Next, Core Data connects and relates all the objects according to the relationship mapping.

Finally, enforce any data validations in the destination model. Core Data disables destination model validations during the data copy.

* NSInferMappingModelAutomaticallyOption: If the value of this key is set to true, Core Data attempts to infer the mapping model for the migration based on the data model versions of the data model.

NSMigratePersistentStoresAutomaticallyOption: By setting the value of this key to true, we tell Core Data to automatically perform a migration if it detects an incompatibility.

1. **What is delete rule in core data**

**Deny**

If there is at least one object at the relationship destination (employees), do not delete the source object (department). For example, if you want to remove a department, you must ensure that all the employees in that department are first transferred elsewhere; otherwise, the department cannot be deleted. Let’s say if you want to delete iOS department from department table then first all the employees in the employee table who is belong to iOS department need to change the department name to another department name then delete the iOS department from department table else app will crash.

**Nullify**

It is default delete rule. Remove the relationship between the objects, but do not delete either object. This only makes sense if the department relationship for an employee is optional, or if you ensure that you set a new department for each of the employees before the next save operation. For example, if you delete a department name then department name will be deleted from department table, leave all the employees as they are in employee table, even if they still believe they belong to null. So let’s say iOS department is deleted and all the employees in employee table belongs to iOS department then it will be pointing to **null**.

**Cascade**

Delete the objects at the destination of the relationship when you delete the source. For example, if you delete a department, then all the employees in employee table will be deleted who is pointing to that department. So let’s say iOS department is deleted and all the employees in employee table belongs to iOS department will be deleted.

**No Action**

Do nothing to the object at the destination of the relationship. For example, if you delete a department name then department name will be deleted from department table, leave all the employees as they are in employee table, even if they still believe they belong to that department. So, let’s say iOS department is deleted and employee table consist of one employee belongs to iOS department then it will still be pointing to iOS department which is not good idea. So, No Action is not preferred.

1. **What is property observer**

willSet and didSet

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.

Property observers are declared as a variable and not as constants because it is only a mutable property that can be tracked by property observers. Hence, property observers are declared with var and not the let keyword.

A property with observers on it when declared should have initial value assigned to it. This actually makes the difference between computed properties and property observers. Computed properties returns a value computed from other variable or properties and does not store the value in memory. Hence, to place observers on a property, you need to set the property to an initial value of a particular type. This value could be optional type or any other type.

One thing to note is that willSet and didSet will never get called on setting the initial value of the property. It will only get called whenever you set the property by assigning a new value to it. It will always get called even if you assign the same value to it multiple times.

willSet and didSet both have a default parameters newValue and oldValue.

1. **What is difference between escaping and non-escaping closure**

1. @nonescaping closure:

When you are passing a closure as the function argument, the closure gets execute with the function’s body and returns the compiler back. As the execution ends, the passed closure goes out of scope and have no more existence in memory.

Lifecycle of the @nonescaping closure:

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

 2. Do some additional work with function.

3. Function runs the closure.

4. Function returns the compiler back.

Here, we just called the function with a closure, closure gets executed at the end of the function’s body.  So, we are not escaping the execution of the closure. As the step 4 get executed closure will have no existence in the memory.

2. @escaping closures:

If a closure is passed as an argument to a function and it is invoked after the function returns, the closure is escaping.

API Call is example

When you are passing a closure as the function argument, the closure is being preserved to be execute later and function’s body gets executed, returns the compiler back. As the execution ends, the scope of the passed closure exists and have existence in memory, till the closure gets executed.

1. **Difference between class and struct**

Struct is value type

Class is reference type

Class

* Must declare initialiser (constructer)
* Has deinitialisers
* Can inherit from other classes

Struct

* It has free initialiser for you, you don’t have to declare initialiser if you do free initialiser will be overwritten by your declared initialiser
* Don’t have deinitialiser
* Cannot inherit from other struct

Both class and structure can do:

* Define properties to store values
* Define methods to provide functionality
* Be extended
* Conform to protocols
* Define intialisers
* Define Subscripts to provide access to their variables

Only class can do:

* Inheritance
* Type casting
* Define deinitialisers

Allow reference counting for multiple references.

1. **What is optional chaining**

If you were to access the courses property through an exclamation mark (!) , you would end up with a runtime error because it has not been initialized yet. Optional chaining lets you safely unwrap this value by placing a question mark (?), instead, after the property, and is a way of querying properties and methods on an optional that might contain nil. This can be regarded as an alternative to forced unwrapping.

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil. Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is nil.

1. **What is optional binding**

Optional binding is a term used when you assign temporary variables from optional in the first clause of an if or while block. Consider the code block below when the property courses have yet not been initialized. Instead of returning a runtime error, the block will gracefully continue execution.

1. **What is mutating keyword**

Mutating methods are methods that we use on value types like structs and Enums. These methods are allowed to modify the instance it belongs to and any properties of that instance.

1. **Difference between weak and unowned**

A weak reference keeps a **weak** reference to the instance it references. This means that the reference to the instance is not taken into account by ARC. Remember that an instance is deallocated if no other objects have a **strong** reference to the instance.

The first difference you need to know about is that an unowned reference is always expected to have a value. This is not true for weak references, which are set to nil if the instance they reference is deallocated. When that happens, the reference is set to nil.

Because a weak reference can be set to nil, it is always declared as an optional. That is the second difference between weak and unowned references. The value of a weak reference needs to be unwrapped before it can be accessed whereas you can directly access the value of an unowned reference.

Be careful, though. If the referenced instance of an unowned reference is deallocated, it isn't set to nil. As a result, a fatal error is thrown if the referenced object of the unowned reference is accessed.

1. **What is capture list?**

The references a closure holds to reference types are strong by default.

A capture list is nothing more than a collection of pairs wrapped in square braces. The pairs are separated by commas. A capture list is defined after the opening curly brace and before the parameter list and return type of the closure. An element of a capture list is the

1. **What is closure?**

Closures*are self-contained blocks of functionality that can be passed around and used in your code.*

Closures can capture and store references to any constants and variables from the context in which they are defined, known as Closure. You can think of a closure as being a function that doesn’t have a name of its own and captures any values from its environment. Functions and closures are first-class objects in Swift: you can store them, pass them as arguments to functions, and treat them as you would any other value or object. Passing closures as completion handlers is a common pattern in many APIs. Standard Swift library uses closures mostly for event handling and call-backs.

1. **What is dependency injection?**

The intent behind Dependency Injection is to **decouple objects** by having one **object supplying the dependencies of another object**.

We use dependency injection for loosely coupled. So, we can test the app easily. Or in future we need to change the code we can do easily.

There are several ways to inject services:

**Constructor**: pass the services in the initializer

**Setter**: use a setter method to inject the services on the fly. For example, examine the anyfunctionName() method.

**Protocol**: create a protocol, make the client conform to it by implementing its methods. One method acts as the setter to inject the service.

1. **Difference between NSOperationQueue and GCD**

NSOperationQueue can be more suitable for long-running operations that may need to be cancelled or have complex dependencies. GCD dispatch queues are better for short tasks that should have minimum performance and memory overhead.

It is possible to cancel operations that have been enqueued in an NSOperationQueue (as far as the operations support it). When you enqueue a block in a GCD dispatch queue, it will definitely be executed at some point.

1. **What is cherry-picking in Git**

To choose a specific commit from one branch and apply it to another branch

Make sure that you are on the branch where you want to apply the commit then use

git cherry-pick <commit-hash/commit id>

1. **Difference between merge and rebase in Git**

**Merge**

When you create a feature branch(f1) from master(m1) then you work on then f2 then f3 also same time m1 updated to m2 them m3

Now if you use Merge at f3 with master then it will be

m1->f1->m2->m3->f2->f3

in same order in which they got updated. so, it will maintain the commit history of each feature and master branch.

• Let's say you have created a branch for the purpose of developing a single feature. When you want to bring those changes back to master, you probably want merge (you don't care about maintaining all of the interim commits).

**Rebase**

If you rebase at f3 then first it will remove all commits of feature branch and rewrite the commit after rebase

so, after rebase it will be like

m1->m2->m3->f1>f2->f3

it will maintain commit history in linear order. Basically, master branch tail will be connected to top of the feature branch

Disadvantages is it rewrite the history.

So, it is very risky when multiple people are working on a single branch because it rewrites the history

* A second scenario would be if you started doing some development and then another developer made an unrelated change. You probably want to pull and then rebase to base your changes from the current version from the repository.

1. **What is GIT stash**

GIT stash takes the current state of the working directory and index and puts in on the stack for later and gives you back a clean working directory.  So, in case if you are in the middle of something and need to jump over to the other job, and at the same time you don’t want to lose your current edits then you can use GIT stash.

1. **How to secure data like username password or any confidential data (Keychain)**

Keychain is great because data encryption automatically is taken care of before it is stored in the file system so there is no need to waste time building encryption algorithms.

A keychain in both OS and iOS can be configured to lock. When locked it is impossible to access and decrypt stored keychain items. For iOS the keychain is locked when the device is locked and unlocked when the device is unlocked. Even when it is unlocked, only apps that have created an item can access it, unless configured otherwise.

Keychain also offers other features like accessing keychain items across apps. Normally, an app only has access to items it created but configuration can be made to let it access data within a group of designated apps.

1. **How to detect memory management**

**Memory Graph Debugging**

Go to the schema editor instead and make sure to enable the ‘Malloc Stack’ for live allocations only and ‘Malloc Scribble’ in Diagnostics section.

‘Malloc Stack’ will give you the stack for your memory leak.

‘Malloc Scribble’ will help XCode give you more accurate results by filling freed memory with a predefined value, so it will be more obvious which memory blocks are not being released.

1. **What are silent push notifications**

They can be used to inform the application of new content without having the user informed. Instead of displaying a notification alert, the application will be awakened in background (iOS does not automatically launch your app if the user has force-quit it) and [application:didReceiveRemoteNotification:fetchCompletionHandler:](https://developer.apple.com/documentation/uikit/uiapplicationdelegate/1623013-application) will be called. You then have the opportunity to process any information transparently for the user :

* Download some content
* Synchronize some elements,
* Inform the user directly within the application when he opens it back

Note that your time is limited to 30s.

1. **What are rich push notifications**

Rich push notifications are short pop-up messages sent to a user’s device with a rich media attachment such as an animated GIF, video, or audio. They allow you to communicate with your customers in an inviting way even when they’re not actively using your app or visiting your website.

Example we see some images on WhatsApp notifications.

1. **What are protocol extensions**

Protocols let us define contracts that conforming types must adhere to, and extensions let us add functionality to existing types.

We can extend a whole protocol to add method implementations, meaning that any types conforming to that protocol get those methods.

So whatever methods we add in extensions will be optional in protocols.

1. **What is CaseIterable Enum**

Swift has a CaseIterable protocol that automatically generates an array property of all cases in an enum. To enable it, all you need to do is make your enum conform to the CaseIterable protocol and at compile time Swift will automatically generate an allCases property that is an array of all your enum’s cases, in the order you defined them.

You must add CaseIterable to the original declaration of your enum rather than an extension for the allCases array to be synthesized – you can’t use extensions to retroactively make existing enums conform to the protocol.

1. **What is Associated value in Enum**

You can set a constant or variable to Enum, and check for this value later. However, it’s sometimes useful to be able to store values of other types alongside these case values. This additional information is called an associated value, and it varies each time you use that case as a value in your code.

You can define Swift enumerations to store associated values of any given type, and the value types can be different for each case of the enumeration if needed.

1. **How many access modifiers are there**

Private -

Can be accessed to same class and in extension but extension must be in the same file.

Like if there is a file MyClass.Swift and class like class Home and extension Home and you make a function or property private then it will be only accessible to only MyClass.Swift file. You can use in extension and class both as well

FilePrivate -

Can be accessed to different class and in extension.

Internal -

It is default and can be accessed within the module only like my Xcode project. If I add another module, then it will not be accessible to that module.

Open -

It is accessible to all the modules, and you can sub class outside the module also.

Public -

It is accessible to all the modules and, but you can sub class within the module only.

1. **What are Actors and how they are different from Class**

Actors

Is used to remove data race.

Data race means when multiple threads are trying to access data at a same time. We can achieve this with dispatch barrier and semaphore but this in swift 5.5 actor has been introduced for the same thing.

Swift’s actors are conceptually like classes that are safe to use in concurrent environments.

Swift automatically ensures no two pieces of code attempt to access an actor’s data at the same time – it is made impossible by the compiler, rather than requiring developers to write boilerplate code using systems such as locks.

* Like classes, actors are reference types. This makes them useful for sharing state in your program.
* They have many of the same features as classes: you can give them properties, methods (async or otherwise), initializers, and subscripts, they can conform to protocols, and they can be generic.
* Actors do not support inheritance, so they cannot have convenience initializers, and do not support either final or override.
* All actors automatically conform to the Actor protocol, which no other type can use. This allows you to write code restricted to work only with actors.

 if you’re attempting to read a variable property or call a method on an actor, and you’re doing it from outside the actor itself, you must do so asynchronously using await.

1. **What is Main Actor**

@MainActor

 represents your main thread.

The main actor will perform all its synchronization on the main dispatch queue.

To use the main actor, you need to add the @MainActor attribute to a definition. It can be either a method or a class. When adding @MainActor to a function, the function will always execute on the main thread.

Adding the @MainActor attribute to a bigger definition such as a class, will make all properties and methods be MainActor. Individual methods can choose to not be part of the main actor by adopting the nonisolated keyword.

1. **What are different app states**

Suspended:

The app is still in memory, so it makes it easier to bring back to the Active State, but it is not running code, so it does not affect battery life.  When the system is low on memory, the app can be terminated from this execution state and moved to the Not Running state.

Suspended → Background → Inactive → Active.

Background:

The app is running in the foreground, but not receiving events. This mode is where your app has been moved to the background but is still able to run code.  Your app can be launched directly to this mode if you ask the system to.

The main difference between Inactive and Background state is that you can update UI in Inactive state, but you cannot update UI in Background state.

Inactive:

It can run some code but does not respond to touch or other events yet.

Active:

When you use the app that is called active state.

1. **What is associated type in protocol**

Associated types are a powerful way of making protocols generic or opaque

WebServiceProtocol protocol that can store any type of those items depend on whatever conforms to the protocol, but we can still use them inside the protocol and any extensions.

protocol WebServiceProtocol: class {

associatedtype type

func getAll(inputType: type)

}

1. **What is opaque type in protocol**

When you declare protocol with associated type and you want to return protocol type in class or struct then you cannot expose return type in function so you need to make generic return type that is why we need to write some keyword before return type in function

1. **What is property wrapper**

When you write extension than it is available for in entire applications. Like if want to validate an email address then you will make one extension on string and write valid email function in extension but in this case this valid function will be available in entire app with all type of string variable. Even name parameter can also use that but if we want to restrict that only email field can access then we will use property wrapper.

1. **What are Equitable and Comparable protocol**

Equitable

Equitable protocol introduced by apple to provide us with the functionality to check the equality between two instances of a class, with the help of equitable protocol we can use “==” and “!=” operators. The class which is needed to be compared should conform this protocol

Comparable

This is an another protocol provided by iOS, which provide us the functionality to use <, <=, >=, and > operators on the custom class objects, it is similar to equitable protocol i.e. we need to conform it and implement its methods, one more important thing about comparable protocol is that is it inherited from equitable so it must override “==” operator as well, so we need to override “==” and “<” operators. Below is the example for it.

1. **What is Hashable protocol**

As Hashable inherits from the Equitable protocol, any type that conforms to Hashable must also conform to Equitable.

Many types in the Swift standard library already conform to the Hashable protocol such as String, Int, Bool, and Double.

Important: Hash values are not guaranteed to be equal across different executions of your program. Do not save hash values to use during a future execution.

Hashable protocol need to provide as far as you can a unique integer value hashValue which need to be the same within two objects when they are equals.

1. **How to handle errors in Combine**

To handle errors in Combine, you can use the catch() operator, which allows you to specify a closure that will be called when an error occurs. You can also use tryCatch operator to handle errors in a more elegant way.

1. **What is the difference between Sink and Assign in Combine**

sink is used to subscribe to a publisher and receive its emitted values, while assign is used to bind the value emitted by a publisher to a property on an object. sink also allows you to handle completion and error events while assign does not.

1. **What is the Cancellable in Combine**

A Cancellable is an object that represents an active subscription to a publisher. It allows you to cancel the subscription by calling its cancel() method.

1. **Why does a struct need mutation?**

You need to use the mutating function if you will change any state contained within the struct. Calling a mutating function returns a new struct in place as Struct is immutable. It works the same as passing an inout parameter to a function. The mutating keyword lets callers know that the method is going to make the value change.

1. **What is codable protocol**

The Codable protocol is used to encode and decode custom data types in Swift. It allows you to easily convert data between different formats, such as JSON and property lists.

The Codable protocol in Swift is really a union of two protocols: Encodable and Decodable. These two protocols are used to indicate whether a certain struct, enum, or class, can be encoded into JSON data, or materialized from JSON data.

When you only want to convert JSON data into a struct, you can conform your object to Decodable. If you only want to transform instances of your struct into Data, you can conform your object to Encodable, and if you want to do both you can conform to Codable.

1. **Difference between array and set**

An array is an ordered collection of values, whereas a set is an unordered collection of unique values. Sets are useful for checking for membership or removing duplicates.

Elements in the array are arranged in order. Or we can say that arrays are ordered. Elements in the set are not arranged in any specified order or we can say that sets are unordered.

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 Hashable protocol which makes Set optimized for performance. It takes the same amount of time to look up an element in a small collection vs a large collection.

1. **What are the higher order functions**

A higher-order function is a function that takes one or more functions as arguments or returns a function as its result. Below are the higher-order functions provided by the Swift language −

* forEach()
* map()
* compactMap()
* flatMap()
* filter()
* reduce()
* sort()
* sorted()

1. **What is difference between Self and self**

When you’re writing protocols and protocol extensions, there’s a difference between **Self** (capital S) and **self** (lowercase S). When used with a capital S, **Self** refers to the type that conform to the protocol, e.g. String or Int. When used with a lowercase S, self refers to the value inside that type, e.g. **“hello Swift”** or **786**.

1. **What is difference between static and final**

Both the static and class keywords allow us to attach variables to a class rather than to instances of a class. For example, you might create a Student class with properties such as name and age, then create a static numberOfStudents property that is owned by the Student class itself rather than individual instances.

Where static and class differ is how they support inheritance: When you make a static property it becomes owned by the class and cannot be changed by subclasses, whereas when you use class it may be overridden if needed.

**55)  What is the difference between the == operator and the === operator in Swift?**

The main difference between the double = operator and the triple = operator is that the equal to “==” operator compares value types to check if the values are the same, and the triple = “===” operator compares reference types to check if the references point to the same instance.

* 1. **What is difference between computed property and stored property**

The main difference is that you cannot assign something to the computed property since it has no setter. In this case the closure only gets called once and the return value gets stored in the variable so if the outcome doesn't change over time it is more efficient to use the stored variable rather than the computed one.

In general: computed properties should only be used if the value can be retrieved quickly.

Sidenote: If you don't change/reassign the stored variable you should consider making it a constant (let)

* 1. **What is difference between guard and if let**

if let and guard let both unwrap optional if they contain a value. guard let is normally used for an early exit from the current scope, so any values you unwrap using it will stay around after the check. While using if let, the value needs to use inside the if scope.

* 1. **What is copy on write**

Copy-on-write is a strategy used in Swift for optimising memory usage. The main idea of COW is that when multiple callers want to access the resources which are same, you can put them pointing to the same resource. The state of the resource will be maintained until a caller tries to modify its “copy” of the resource.

* 1. **What is intrinsic content size**

All views have an intrinsic content size which refers to the amount of space the view needs for its content to appear in an ideal state. For example, the intrinsic content size of a UILabel will be the size of the text it contains using whatever font you have configured it to use.

* 1. **What is GCD’s(Grand central dispatch) Quality of Service**

Quality of Service is the priority to perform task in GCD. If the task has higher quality of service than other, it will be handled before lower quality of service.

Quality of Services are ranked from highest to lowest –

User Interactive: Work that happens on the main thread, such as animations or drawing operations. User Initiated: Work that the user kicks off and should yield immediate results. This work must be completed for the user to continue. Utility: Work that may take a bit and doesn’t need to finish right away. Analogous to progress bars and importing data. Background: This work isn’t visible to the user. Backups, syncs, indexing, etc.

* 1. **What is retain cycle**

Retain Cycle is the situation when two objects keep a reference to each other and are retained, it creates a retain cycle since both objects try to retain each other, making it impossible to release. This mostly happens in classes and closures. Closures live in memory, so when you use “self” which is a reference, you need to make sure that you solve the retain cycle.

* 1. **What is MVVM design pattern**

MVVM/Model-View-ViewModel is an architectural pattern that enables you to separate the business and presentation logic from its UI. Creating the clean separation will help you to find bugs easily and to test effectively. MVVM will be a good choice if you want to shrink the size of your view controller and make the business logic more readable.

The View Model invokes changes in the Model and updates itself with the updated Model, and since we have a binding between the View and the View Model, the first is updated accordingly.

* 1. **What is the difference between an @ObservableObject and @EnvironmentObject in SwiftUI**

An @ObservableObject is a class that conforms to the ObservableObject protocol. This protocol has one requirement: a property called objectWillChange. This property is used to notify SwiftUI of any changes to the object so that the UI can be updated. An @EnvironmentObject is a property wrapper that injects an object into the environment of a view. This means that the object will be available to all views that are descendants of the view that defines the @EnvironmentObject.

* 1. **What is the use of @State property in SwiftUI**

The @State property wrapper can be used to create user input forms in SwiftUI by binding the value of a text field to a state variable. This will allow the text field to update its value as the user types and will also save the value of the text field when the user finishes editing it.

* 1. **What is the use of @Binding property in SwiftUI**

The @Binding property wrapper is used to create a two-way binding between a view and its underlying model. This means that when the model changes, the view will update to reflect the change, and vice versa. This is useful for keeping the UI in sync with the underlying data.

* 1. **What does SwiftUI use Struct for views?**

SwiftUI uses structs for views due to the benefits they offer in terms of performance and thread-safety. Structs are preferred over classes when they are small and can be easily copied, as this eliminates the risk of multiple references to the same instance and potential memory leaks or conflicts when multiple threads try to access or modify a shared instance.

In a multithreaded environment or when passing a variable between many classes, using structs ensures that a copy of the variable is always sent, rather than a reference to the original instance, which can prevent unintended changes to the variable’s value.

* 1. **Why do we need UIViewRepresentable?**

UIViewRepresentable acts as a wrapper to use UIKit’s UIView in SwiftUI View. To add your view into your SwiftUI interface, create your UIViewRepresentable instance and add it to your SwiftUI interface. The system calls the methods of your representable instance at appropriate times to create and update the view.

* 1. **What is ViewModifier?**

ViewModifier is the protocol that helps to customise another modifier or views. You can chain multiple view modifiers and at the end, it will return another view. To enhance a UIView in a UIKit, you use the view properties.

In UIKit, you probably use YourView.backgroudColor = .red to customise the UIView. It is an imperative approach to enhance your views. To customise a view in a SwiftUI declarative approach, developer just define what the view should look like using ViewModifier.

* 1. **Why ZStack is necessary?**

When we are required to put one view on top of another view like Text on Image at this time ZStack plays its role and it allows us to overlay its child views on top of each other. Stacks can further be customized with alignment and spacing to modify their appearance.

* 1. **Why does SwiftUI use ‘some View’ for its view type?**

Some keyword is used to describe an opaque type. The opaque type is useful to return a type without providing the exact details on the concrete type. Opaque types as a kind of generic function where the placeholder types are filled by the implementation return type. The opaque type limits what callers need to know about the returned type, only exposing information about its protocol compliance.

Using an opaque type is a way to let the compiler decide what would be the concrete type of a function return, based on the actual returned value, limiting the options to the types that comply to a given protocol. Returning some View means even though we don’t know what view type is going back, the compiler does.

* 1. **Why does SwiftUI render the Views?**

SwiftUI is a declarative framework for building user interfaces on Apple platforms. In a declarative framework, you describe the desired end state of your user interface, and the framework is responsible for rendering the views and handling user interactions.

In SwiftUI, you create views by combining simple, reusable building blocks called “view components.” These view components can be combined in a variety of ways to create more complex layouts and interactions.

When you build a SwiftUI app, you create a hierarchy of views by composing these view components and arranging them within a parent-child hierarchy.

Each view in the hierarchy is responsible for rendering its own content and layout, as well as handling user interactions and responding to changes in state.

SwiftUI uses a “diffing” algorithm to determine the minimum set of changes needed to update the user interface whenever the state of your app changes. When the state of a view changes, SwiftUI compares the new and old versions of the view and determines the minimum set of changes needed to update the view. This allows SwiftUI to update the user interface efficiently, without having to rebuild the entire view hierarchy from scratch.

SwiftUI also uses a “lazy” rendering strategy, which means that it only renders views that are visible on the screen. This helps to improve performance by reducing the number of views that need to be rendered and updated at any given time.

In summary, SwiftUI renders views by building a hierarchy of view components, using a diffing algorithm to determine the minimum set of changes needed to update the user interface, and using a lazy rendering strategy to improve performance. This allows SwiftUI to provide a fast and efficient way to build and update the user interface of your app.

* 1. **Why opaque types are so important in SwiftUI?**

Opaque return types are a feature of the Swift programming language that allow you to hide the underlying type of a value behind a “type-erased” wrapper. In SwiftUI, opaque return types are important because they allow you to create reusable view components that can be used in a variety of contexts without exposing their internal implementation.

For example, imagine that you have a view component that displays a list of items, and you want to allow users to tap on an item to select it. You could create a view component that returns a Binding<Int> to represent the selected item index, but this would limit the reuse of the view component to only situations where the selected item index is needed.

To make the view component more reusable, you can use an opaque return type to hide the underlying type of the selection. This allows you to use the view component in a variety of contexts, without exposing the internal implementation of the selection.

* 1. **Why is the role of the Combine Framework in SwiftUI?**

The Combine framework is a powerful tool for handling asynchronous events and managing data flow in SwiftUI. The framework provides a set of operators and publishers that you can use to process and transform streams of data, as well as a set of subscribers that you can use to respond to changes in the data. The Combine framework is also used to handle different events like UI or Network related events in SwiftUI.

In conclusion, SwiftUI is a powerful framework that allows you too easily create beautiful and responsive user interfaces, while providing powerful tools for data flow and state management. Whether you’re building a simple app or a complex one, understanding the basics of data flow, view communication, state management and Combine is essential to creating great SwiftUI apps.

* 1. **What are Publishers in the Combine framework?**

Publishers are the main building blocks of the Combine framework. They are objects that emit a stream of values over time, such as user input or network data. They can be transformed and combined in various ways to create complex data flows. A publisher can be thought of as an observable object that can be subscribed to by a subscriber.

* 1. **What are Subscribers in the Combine framework?**

Subscribers are objects that listen to the events emitted by publishers and respond to them. Subscribers can receive events, errors, or completion from a publisher. Subscribing to a publisher creates a subscription that can be used to control the flow of events and handle errors or completions.

* 1. **What is core data stack?**

A Core Data stack consist of following objects:

* a managed object models
* a managed object context
* a persistent store coordinator
* a persistent store (storage)
* NSManagedObjectModel
* An object that is an instance of NSManagedObjectModel. An NSManagedObjectModelobject describes a schema, a collection of
* entities (data models) that you use in your application. It uses a private internal store to maintain its properties and implements all
* the basic behaviour required of a managed object.

Managed Object Context

A managed object should always be associated with a managed object context. There are no exceptions to this rule. Remember that a managed object context manages a number of records or managed objects.

Persistent Store Coordinator

The persistent store coordinator is the glue of the Core Data stack. It keeps a reference to the managed object model and the managed object context. And, as the name implies, the persistent store coordinator oversees the persistent store of the application.

A coordinator that associates persistent stores with a model (or a configuration of a model) and that mediates between the persistent stores and the managed object contexts.

Persistent store

A persistent store is a repository in which managed objects may be stored. You can think of a persistent store as a database data file where individual records each hold the last-saved values of a managed object. Core Data offers three native file types for a persistent store: binary, XML, and SQLite. From the above screen shot you can see persistent store coordinates with Persistent store coordinator.

* 1. **How to migrate older data model to newer one?**

Yes, it is possible to migrate an older data model to a newer one. The process is known as “data migration.” To migrate data from an older model to a newer one, you first need to create a new data model that is compatible with the older model.

Next, you need to write code that will map the data from the older model to the new model. Finally, you need to run a migration process that will transfer the data from the old model to the new model.

* 1. **What is Object graph in Core data**

The object graph is the set of all objects managed by Core Data. This includes all the objects you have fetched, as well as any new objects you have created but not yet saved. The object graph is used by Core Data to keep track of all of the objects it is managing, and to ensure that they are all properly linked together.

* 1. **What is NSFetchedResultsController in Core data**

NSFetchedResultsController is a class that is used to manage the results of a Core Data fetch request. It is typically used in conjunction with a table view to display the results of the fetch request in the table view.

The NSFetchedResultsController class also provides support for managing the results of a fetch request in relation to changes that are made to the underlying data store.

* 1. **Why should we use lightweight migration instead of heavyweight ones?**

Lightweight migrations are generally preferred over heavyweight migrations because they are less disruptive and can be performed more quickly. Heavyweight migrations can take a long time to complete and can cause data loss if not performed correctly.

* 1. **What is GCD**

GCD(Grand central dispatch) is a library that provides a low level and object based API to run tasks concurrently while managing threads behind the scenes. GCD abstracts away thread management code and moves it down to the system level, exposing a light API to define tasks and execute them on an appropriate dispatch queue.

* 1. **Explain three GCD queues**

GCD provides three type of queues –

Main queue: Serial queue – It runs on the main thread. Global queue: Concurrent queue – It runs with different priorities and shared by the entire system. Custom queue: Serial/ Concurrent queue.

* 1. **Explain is GCD’s Quality of Service?**

Quality of Service is the priority to perform task in GCD.

If the task has higher quality of service than other, it will be handled before lower quality of service.

Quality of Services are ranked from highest to lowest –

User Interactive: Work that happens on the main thread, such as animations or drawing operations.

 User Initiated: Work that the user kicks off and should yield immediate results. This work must be completed for the user to continue.

 Utility: Work that may take a bit and doesn’t need to finish right away. Analogous to progress bars and importing data.

Background: This work isn’t visible to the user. Backups, syncs, indexing, etc.

* 1. **What is NSOperation?**

NSOperation is built on top of GCD. While using NSOperation developer can add dependency among various operations and re-use, cancel and suspend operations. Of course, the same thing can also be archived through GCD but it will add extra overhead.

* 1. **What is Semaphore?**

Semaphores gives us the ability to control access to a shared resource by multiple threads. A semaphore consists of a threads queue and a counter value. Counter value is used by the semaphore to decide if a thread should get access to a shared resource or not. The counter value changes when we call signal() or wait() functions.

* 1. **What is DispatchGroup?**

DispatchGroup allows for aggregate synchronization of work. You can use them to submit multiple different work items and track when they all complete, even though they might run on different queues. This behaviour can be helpful when progress can’t be made until all the specified tasks are complete.

* 1. **What is difference between aspect fit and aspect fill?**

Aspect fill and aspect fit are two different ways to display an image in an iOS app. Aspect fill will scale the image to fill the entire space, potentially cropping parts of the image. Aspect fit will scale the image to fit the entire space, potentially leaving some empty space.

* 1. **What is anchors in auto layout?**

Anchors in Auto Layout are used to specify the position and size of a view relative to its parent view or other layout guides. This makes it easy to create responsive user interfaces that adapt to different screen sizes and orientations. It will adjust the UI automatically according to device size.

* 1. **What is the use reuse identifiers in tableview cell in auto layout?**

Reuse identifiers for table view cells are used to improve performance in iOS apps. When a table view needs to display many cells, it will dequeue cells that have already been created but are no longer visible, instead of creating new ones. This can significantly improve the performance of the app.

So only visible cell will take the memory, and when scrolling is done then invisible cell’s memory will be reused to visible cells.

* 1. **When would you use a collection view rather than a table view?**

A collection view is typically used when displaying many items that are organized into multiple sections, or when displaying items with a more complex layout than a table view can support.

So, when you want to display multiple cells in a single row then you will use collection view rather tableview.

* 1. **What is the difference between @IBOutlet and @IBAction**

@IBOutlet is used to create a reference to a user interface element, such as a label or button, to manipulate its properties in code. @IBAction is used to create an action method, which is a method that will be called in response to an event, such as a button tap.

* 1. **How to do respond user interaction in SwiftUI?**

You can use the onTapGesture() modifier to respond to tap gestures, and the onLongPressGesture() modifier to respond to long press gestures. You can also use the .onReceive() method to respond to external events, such as notifications or changes in the environment.

* 1. **Difference between a view’s initializer and the onAppear() method in SwiftUI?**

The view’s initializer is used to set up the initial state of a view when it’s first created. On the other hand, the onAppear() method is called every time the view appears on the screen, whether it’s the first time or not. It’s useful to perform actions that need to be done every time the view appears, such as fetching data or starting animations.

* 1. **How to implement programmatic navigation in a SwiftUI app?**

In SwiftUI, programmatic navigation can be achieved by using the NavigationLink view, which allows you to navigate to a new view when a user interacts with it. You can also use the NavigationView to wrap your views and provide a navigation bar.

* 1. **How does Combine differ from RxSwift?**

Combine is a native framework for iOS, whereas ReactiveCocoa and RxSwift are third-party libraries. Combine also has a simpler API and a more Swift-like syntax, making it easier for developers to learn and use. It only imports necessary files needed for the projects

* 1. **What are the ways to cancel a subscription in Combine?**

The two ways to cancel a subscription in Combine are to call the **cancel()** method on the subscriber or to call the **subscription.cancel()**method on the returned Subscription object.

* 1. **How can you use the reduce operator in Combine?**

The reduce operator allows you to accumulate the values emitted by a publisher and return the result. Here is an example of using the reduce operator to find the sum of an array of integers:

let numbers = [1, 2, 3, 4, 5]

let publisher = numbers.publisher

publisher

.reduce(0, +)

.sink(receiveValue: { sum in

print("The sum is: \(sum)")

})

.store(in: &cancellables)

* 1. **How can you use the scan operator in Combine?**

The scan operator is like reduce, but it emits the intermediate results as well as the final result. Here is an example of using the scan operator to find the running total of an array of integers:

let numbers = [1, 2, 3, 4, 5]

let publisher = numbers.publisher

publisher

.scan(0, +)

.sink(receiveValue: { total in

print("The running total is: \(total)")

})

.store(in: &cancellables)

* 1. **How can you use the debounce operator in Combine?**

The debounce operator allows you to delay the emission of values by a certain amount of time. Here is an example of using the debounce operator to delay the emission of button presses by 0.5 seconds:

let button = UIButton()

let publisher = button.publisher(for: .touchUpInside)

publisher

.debounce(for: .milliseconds(500), scheduler: DispatchQueue.main)

.sink(receiveValue: { \_ in

print("Button pressed")

})

* 1. **How can you use the map operator in Combine?**

The map operator allows you to transform the values emitted by a publisher. Here is an example of using the map operator to square the values emitted by an array of integers:

let numbers = [1, 2, 3, 4, 5]

let publisher = numbers.publisher

publisher

.map { $0 \* $0 }

.sink(receiveValue: { value in

print("The squared value is: \(value)")

})

.store(in: &cancellables)

* 1. **How can you use the map operator in Combine?**

The map operator allows you to transform the values emitted by a publisher. Here is an example of using the map operator to square the values emitted by an array of integers:

let numbers = [1, 2, 3, 4, 5]

let publisher = numbers.publisher

publisher

.map { $0 \* $0 }

.sink(receiveValue: { value in

print("The squared value is: \(value)")

})

.store(in: &cancellables)

* 1. **How can you use the zip operator in Combine?**

The zip operator allows you to combine the values emitted by two or more publishers. Here is an example of using the zip operator to combine the name and age of a user:

let namePublisher = user.namePublisher

let agePublisher = user.agePublisher

Publishers.Zip(namePublisher, agePublisher)

.sink(receiveValue: { name, age in

print("Name: \(name), Age: \(age)")

})

.store(in: &cancellables)

* 1. **How can you use the throttle operator in Combine?**

The throttle operator allows you to limit the frequency of the values emitted by a publisher. Here is an example of using the throttle operator to limit the rate of API calls:

let publisher = searchTextField.publisher(for: .editingChanged)

.map { $0.text }

.throttle(for: .seconds(1), scheduler: DispatchQueue.main, latest: true)

* 1. **What is the difference between a managed object and a managed object context in core data?**

A managed object is an instance of an object that is managed by the Core Data framework. It represents the data in an application. A managed object context is an object that provides a way to interact with the data represented by managed objects. It is responsible for creating, retrieving, updating, and deleting managed objects.

* 1. **What is the function of NotificationCenter?**

NotificationCenter is a mechanism for broadcasting notifications across different parts of an app. It allows objects to communicate with each other without needing to know about each other’s existence.

It is a one-to-many object relationship model. If one object sends a message, then there can be multiple receivers.

* 1. **What is the iBeacon?**

iBeacons are a technology used for location-based services that allow apps to detect and range iBeacons in the physical world. iBeacon is dependent on location, so user has to on location. iBeacon tracking is done using the CLLocationManager class, which is also responsible for requesting location permission from users.

The value of iBeacon that we need to know first.

- UUID is a set of numbers that are created for Beacon

- Major is a set of numbers indicating that beacon is in which group can be referenced from the same UUID

- Minor is a set of numbers indicating that beacon is in which sub-group that divided from major.

* 1. **How can you create a custom layout in SwiftUI?**

To create a custom layout in SwiftUI, you can use the GeometryReader view, which provides access to the size and position of its parent view, and the position and size properties of its children. You can also use the frame and offset modifiers to adjust the position and size of views.

struct MyView: View {

var body: some View {

GeometryReader { geometry in

// Here goes your view content,

// and you can use the geometry variable

// which contains geometry.size of the parent

// You also have function to get the bounds

// of the parent: geometry.frame(in: .global)

}

}

}

* 1. **What is receive(on:) operator in Combine?**

The receive(on:) operator is used to specify the scheduler on which the subscriber will receive events. It takes a parameter of type Scheduler, which can be used to schedule the reception of events on a specific thread or queue. This is useful for ensuring that updates to the user interface are made on the main thread, for example.

import SwiftUI

struct ContentView: View {

@StateObject private var state: ContentViewState = .init()

var body: some View {

VStack {

Text(state.count.description)

Button("Count Up") { state.countUp() }

Button("Reset") { state.reset() }

}

.onReceive(state.$count) { count in

print(count)

}

}

}

final class ContentViewState: ObservableObject {

@Published private(set) var count: Int = 0

func countUp() {

count += 1

}

func reset() {

count = 0

}

}

* 1. **What is assign(to:on:) operator in Combine?**

The assign(to:on:) operator is used to bind the output of a publisher to a property on an object. It takes two parameters: the property to bind to and the object to bind to.

class MyClass {

var anInt: Int = 0 {

didSet {

print("anInt was set to: \(anInt)", terminator: "; ")

}

}

}

var myObject = MyClass()

let myRange = (0...2)

cancellable = myRange.publisher

.assign(to: \.anInt, on: myObject)

* 1. **What is difference between a static method and an instance method in Swift?**

A static method is a method that is called on the class itself, while an instance method is a method that is called on an instance of the class.

Instance methods are functions that belong to instances of a particular class, structure, or enumeration. They support the functionality of those instances, either by providing ways to access and modify instance properties, or by providing functionality related to the instance’s purpose.

Instance methods, as described above, are methods that are called on an instance of a particular type. You can also define methods that are called on the type itself. These kinds of methods are called type methods. You indicate type methods for classes by writing the keyword class before the method’s func keyword, and type methods for structures and enumerations by writing the keyword static before the method’s func keyword.

struct User {

// MARK: - Properties

let name: String

// MARK: - Instance Methods

func sayHello() -> String {

"Hello, \(name)"

}

// MARK: - Type Methods

static func createUser(with name: String) -> User {

User(name: name)

}

}

let john = User.createUser(with: "John")

class Book {

// MARK: - Properties

let title: String

let author: String

// MARK: - Initialization

init(title: String, author: String) {

self.title = title

self.author = author

}

// MARK: - Type Methods

static func favorites() -> [Book] {

[

Book(title: "LOTR", author: "Tolkien")

]

}

class func moreFavorites() -> [Book] {

[

Book(title: "The Hobbit", author: "Tolkien")

]

}

}

Book.favorites()

Book.moreFavorites()

* 1. **What is difference between a** **== and ===?**

The main difference between the double = operator and the triple = operator is that the equal to “==” operator compares value types to check if the values are the same, and the triple = “===” operator compares reference types to check if the references point to the same instance.

class MyClass: Equatable {

let myProperty: String

init(s: String) {

myProperty = s

}

}

func ==(lhs: MyClass, rhs: MyClass) -> Bool {

return lhs.myProperty == rhs.myProperty

}

let myClass1 = MyClass(s: "Hello")

let myClass2 = MyClass(s: "Hello")

myClass1 == myClass2 // true

myClass1 != myClass2 // false

myClass1 === myClass2 // false

myClass1 !== myClass2 // true

* 1. **Why can’t we declare struct as open?**

Because it is not possible inheritance between structs. “Open” means that you can override a parent method/property of a given class outside of module.

* 1. **What is the difference between lazy and eager loading in Swift?**

Lazy loading is when the initialization of a property is delayed until it is accessed for the first time. Eager loading is when the property is initialized as soon as the object is created.

* 1. **What is the difference between a singleton and a global variable in Swift?**

A singleton is a design pattern that ensures that a class has only one instance, whereas a global variable is a variable that is accessible from anywhere in your code. Singletons are useful for creating shared resources, such as a shared network manager, whereas global variables should generally be avoided in favour of dependency injection.

* 1. **What is the difference between Delegates and Callbacks?**

Delegation simply means that you pass yourself to someone else, when you want that someone to notify you of changes, so you can react to them. For instance, if a ViewController talks to a network service, and wants to get notified when that service is done with some request, it would make itself the network service’s delegate. The network service would then call the delegate methods when it’s done.

Callbacks are similar in function to the delegate pattern. They do the same thing: letting other objects know when something happened and passing data around. What differentiates them from the delegate pattern, is that instead of passing a reference to yourself, you are passing a function.

* 1. **What is trailing closure syntax?**

In swift functions allow multiple closures as parameter. Trailing closure syntax is a little piece of syntactic sugar that makes the code pleasant to read and write. Trailing closure syntax is used as the final parameter to a function.

// Without trailing closure syntax

func greeting(title: String, greetingCallBack: () -> ()) {

print("Hello world, \(title)")

greetingCallBack()

}

// With trailing closure syntax

func greeting(title: String) {

print("Hello world, \(title)")

}

* 1. **What is inout parameter?**

All parameters passed into a Swift function are constants, so you can’t change them. If you want, you can pass in one or more parameters as inout, which means they can be changed inside your function, and those changes reflect in the original value outside the function.

For example, if you want to double a number in place – i.e., change the value directly rather than returning a new one – you might write a function like this:

func doubleInPlace(number: inout Int) {

number \*= 2

}

To use that, you first need to make a variable integer – you can’t use constant integers with inout, because they might be changed. You also need to pass the parameter to doubleInPlace using an ampersand, &, before its name, which is an explicit recognition that you’re aware it is being used as inout.

In code, you’d write this:

var myNum = 10

doubleInPlace(number: &myNum)

* 1. **What is difference between upcast and downcast ?**

In swift, upcast will be checked on compile time and it goes from a derived class to a base class. Upcasting is casting to a super-type, while down casting is casting to a subtype. Upcasting is always allowed, but down-casting involves a type check. We can downcast using optional with as? or forced failable with as!.

Type casting in Swift is implemented with the is and as operators. is is used to check the type of a value whereas as is used to cast a value to a different type.

* 1. **What is difference between as?, as! And as in Swift?**
* “as” is used for upcasting.
* as? produces an optional value. It will produce nil when the casting cannot be done successfully.
* as! doesn’t produce an optional value, it will create a value of the specified type. The program will crash if the casting cannot be done successfully.

* 1. **What is race condition?**

A race condition is a situation in which the behavior of a software system is dependent on the relative timing of events, such as the order in which threads or tasks are executed. In iOS, a race condition can occur when multiple threads or tasks attempt to access or modify the same data simultaneously, leading to unexpected or incorrect results.

For example, imagine two threads both attempting to increment the same variable. If both threads access the variable at the same time, increment its value, and then store the result, the variable may end up with an incorrect value, because the order in which the increments are executed is not guaranteed. This is an example of a race condition.

Race conditions can be difficult to detect and debug, as they may not occur consistently or predictably. To avoid race conditions in iOS, it’s important to use synchronization techniques such as locks, semaphores, or atomic operations to ensure that data is accessed and modified in a controlled manner.

* 1. **What is TaskGroup?**

The task group in swift is a collection of tasks that work together to produce a result. Each task inside the task group can generate the partial result data which will be used to generate an expected result. A very common example of task group usage is creating a collage by downloading multiple images from the network.

func printMessage() async {

let string = await withTaskGroup(of: String.self) { group -> String in

group.addTask { "Hello" }

group.addTask { "From" }

group.addTask { "A" }

group.addTask { "Task" }

group.addTask { "Group" }

var collected = [String]()

for await value in group {

collected.append(value)

}

return collected.joined(separator: " ")

}

print(string)

}

await printMessage()

* 1. **What is strong reference cycle in closures?**

For a closure to execute later, it needs to retain any variables that it needs for it to run. A closure captures references as strong by default like a class. If you assign a closure to a property of a class instance, and the body of that closure captures the instance then strong reference cycle can occur.

The strong reference can occur because the closure’s body accesses a property/method of the instance(for example self.testProperty, self.testMethod(), In both case, these kind of accesses cause the closure to capture self and create a strong reference cycle.)

* 1. **What is circular dependencies?**

You can't have two classes where the initializer of both requires a reference to the other. It's simply not possible.

You need to change at least one of the classes to have an initializer that does not need the other.

Then you also have a problem of both having a strong reference to the other. This will cause a reference cycle (a form of memory leak). So in addition to changing one of the initializers, you should make one of the references weak.

So in the end, you need to make this much like you would any other parent-child class reference.

Without knowing more, I would suggest updating UserInteractor to not require the LoginInteractor. You might have a user that does several things, login being just one possible case.

class UserInteractor {

weak var loginInteractor: LoginInteractor?

init() {

}

}

class LoginInteractor {

let userInteractor: UserInteractor {

didSet {

userInterator.loginInteractor = self

}

}

init(userInteractor: UserInteractor) {

self.userInteractor = userInteractor

}

}

With this setup you can create an instance of UserInteractor. You can then create an instance of LoginInteractor using your instance of UserInteractor.

let userInteractor = UserInteractor()

let loginIteractor = LoginInteractor(userInteractor: userInteractor)

* 1. **What is the problem with MVC architecture?**

MVC/Model-View-Controller is an architectural pattern used to decouple user-interface (view), data (model), and application logic (controller). The main concern of MVC is that it does not enforce developers to decouple responsibilities. For example: network related codes are often found in ViewController. ViewController is treated as dumping station for all kind of logics sometimes. For this reason, MVC is often called massive view controller.

MVC roles are described here –

• Model will only encapsulates the data, it must not contain any business logic. • View will encapsulate what user can see in the UI and it also should not have business logic. • Controller should maintain all kinds of business logic that goes between the View and the Model.