Why Swift? Advantages?

Swift is a compiled programming language for iOS, macOs, watchOs, tvOS, and Linux applications. Created by Apple in 2014. We use Swift because it has open source meaning creators acknowledged the fact that in order to build a defining programming language, the technology needs to be open for all. So, within its seven years of existence, Swift acquired a large supportive community and an abundance of third-party tools. It is safe; Its syntax encourages you to write clean and consistent code which may even feel strict at times. Swift provides safeguards to prevent errors and improve readability. It is fast; Swift was built with performance in mind. Not only does its simple syntax and hand-holding help you develop faster, but it also lives up to its name: As stated on apple.com, Swift is up to 2.6x faster than Objective-C and 8.4x faster than Python. Lastly it is in demand; Remaining supreme to Objective C, Swift was ranked 20th among the most popular programming languages of 2021 (while Objective C is ranked 25th) and 8th among the most loved languages.

Advantages:

- **Rapid development process** - A clean and expressive language with a simplified syntax and grammar, Swift is easier to read and write. It is very concise, which means less code is required to perform the same task, as compared to Objective-C.
- **Easier to scale the product and the team** - In addition to faster development time, you get a product that is future-proof and can be extended with new features as needed. Thus, Swift projects are typically easier to scale.
- **Improved performance, speed of development, and safety** - With a focus on performance and speed, the language was initially designed to outperform its predecessor. Namely, the initial release claimed a 40 percent increase in performance, as compared to Objective-C.
- **Decreased memory footprint** - When you build an app, you use a lot of third-party code – reusable and often open source frameworks or libraries compiled into your app's code. These libraries can be static and dynamic (or shared).
- **Interoperability with Objective-C** - Swift language is perfectly compatible with Objective-C and can be used interchangeably within the same project. This is especially useful for large projects that are being extended or updated: You can still add more features with Swift, taking advantage of the existing Objective-C codebase.

- **Automatic memory management with ARC** - Swift's ARC determines which instances are no longer in use and gets rid of them on your behalf. It allows you to increase your app's performance without lagging your memory or CPU.

What is type Safety in swift

Swift is a type-safe language. A type safe language encourages you to be clear about the types of values your code can work with. If part of your code requires a String, you can't pass it an Int by mistake.

Because Swift is type safe, it performs *type checks* when compiling your code and flags any mismatched types as errors. This enables you to catch and fix errors as early as possible in the development process.

What is type Inference swift

If you don't specify the type of value you need, Swift uses type inference to work out the appropriate type. Type inference enables a compiler to deduce the type of a particular expression automatically when it compiles your code, simply by examining the values you provide.

Because of type inference, Swift requires far fewer type declarations than languages such as C or Objective-C. Constants and variables are still explicitly typed, but much of the work of specifying their type is done for you.

What is optionals

An optional in Swift is basically a constant or variable that can hold a value OR no value. The value can or cannot be nil.

How to create optional variables

It is denoted by appending a "?" after the type declaration. For example, var someValue:Int?

Diff b/w ? And ! , when to use !

- Use ? if the value can become nil in the future, so that you test for this.
- Use ! if it really shouldn't become nil in the future, but it needs to be nil initially.

if You are using **attachment.image!.size** any variable with ! means, the compiler doesn't bother about whether the variable has any value or nil. It goes to take action further, it will here try to get the size of the image. If attachment.image is nil then the app will crash here.
While, **attachment.image?.size**, this will make sure that if attachment.image is not nil then further action will take place, otherwise it confirms the application will not crash in case of a nil value of the image.

Diff b/w let vs Var

The difference between var and let in terms of Swift programming language is that var creates mutable variables, whereas let allows you to create immutable variables. This means that the variables that have been created by var and let can either hold a value or reference. The let keyword is used in order to declare a constant whilst the var keyword is used to declare a variable. Another dissimilarity is that the value of a constant is not possible to change once it is set but is the opposite when it comes to a variable as it can be set to a different value.

What is optionals Binding

Optional binding helps you to find out whether an optional contains a value or not. If an optional contains a value, that value is available as a temporary constant or variable. Therefore, optional binding can be used with an if-let or guard statement to check for a value inside an optional, and to extract that value into a constant or variable in a single action.

What is Optional Chaining

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 .

What is diff b/w if let and guard let

if let and guard let serve similar, but distinct purposes. The "else" case of guard must exit the current scope. Generally that means it must call, return or abort the program. guard is used to provide early return without requiring nesting of the rest of the function.
If let nests its scope, and does not require anything special of it. It can return or not.

In general, if the if-let block was going to be the rest of the function, or its else clause would have a return or abort in it, then you should be using guard instead. This often means (at least in my experience), when in doubt, guard is usually the better answer. But there are plenty of situations where if let still is appropriate.

What is nill colensansing operator

In Swift, you can also use nil-coalescing operator to check whether a optional contains a value or not. It is defined as (a ?? b). It unwraps an optional a and returns it if it contains a value, or returns a default value b if a is nil.

What is tuple

In Swift, a tuple is a group of different values. And, each value inside a tuple can be of different data types. Suppose we need to store information about the name and price of a product, we can create a tuple with a value to store name (string) and another value to store price (float).

What is optionals behind the scene

Declared as an enumeration.

Struct vs Class

Structures and classes are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.

STRUCTS	**CLASSES**
Value Types	Reference Types
Not inheritable	Inheritable
Structures use stack allocation	Classes use heap allocation
All structure elements are public by default	class variables and constants are private by default, while other class members are public by default

Does not require a constructor	Does require a constructor
Cannot be declared as protected	Can be declared as protected

What situation you will use Structs and Classes

It's recommended to use struct in Swift by default. Structs are helpful in these scenarios, too:

- **Simple Data Types**
- **Thread Safety**
- **Mostly Structs Scenario**
- **Doesn't need inheritance**

It's recommended to use a class if you need the specific features of a class. This is why working with structs is the default, and classes are a deliberate choice.

- Classes have a few extra characteristics that structs don't have:
    - Classes can inherit from another class, which you can't do with structs. With classes, you can write the class MyViewController : UIViewController to create a subclass of UIViewController. Structs can implement protocols, can be extended, and can work with generics, though!
    - Classes can be deinitialized, i.e. they can implement a deinit function. Also, you can make one or more references to the same class (i.e., classes are a reference type).
    - Classes come with the built-in notion of identity, because they're reference types. With the identity operator === you can check if two references (variables, constants, properties, etc.) refer to the same object.

What is closures

Closures are self contained blocks of functionality that can be passed around and used in your code. These closures can capture and store references to any constants and variables from the context in which they are defined. This is known as closing over those constants and variables.

Different type of Closures

Trailing Closure -

If you need to pass a closure expression to a function as the function's final argument and the closure expression is long, it can be useful to write it as a *trailing closure* instead. You write a trailing closure after the function call's parentheses, even though the trailing closure is still an argument to the function.

Escaping Closures -

A closure is said to escape a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write @escaping before the parameter's type to indicate that the closure is allowed to escape.

Auto Closures -

An autoclosure is a closure that's automatically created to wrap an expression that's being passed as an argument to a function. It doesn't take any arguments, and when it's called, it returns the value of the expression that's wrapped inside of it. This syntactic convenience lets you omit braces around a function's parameter by writing a normal expression instead of an explicit closure.

Is Closure a reference type or value type

Reference types

Difference b/w escaping and non escaping closures

Non escaping - 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.

Life Cycle of the non escaping closure:

- Pass the closure as function argument, during the function call
- Do some additional work with function
- Function runs the closure
- Function returns the compiler back

Escaping - When passing a closure as the function argument, the closure is being preserved to be executed later and the function's body gets executed, returning the compiler back. As the execution ends, the scope of the passed closure exists and remains in memory until the closure is executed.

Diff ways to escape the closure : asynchronous execution - on dispatch queue, the queue will hold the closure in memory for you to be used in the future. In this case you have no idea when the closure will be executed. Storage - when you need to preserve the closure in storage that exists in the memory, the past of the calling function gets executed and returns the compiler back. (Like waiting for the API response)

Life Cycle of the escaping closure:

- Pass the closure as function argument, during the function call
- Do some additional work with function
- Function execute the closure asynchronously or stored
- Function returns the compiler back

Set vs Array vs Dictionaries

Sets are unordered collections of unique values. Arrays are ordered collections of values. Dictionaries are unordered collections of key-value associations.

What is higher order functions

Higher order functions are simply functions that operate on other functions by either taking a function as an argument, or returning a function. Swift's Array type has a few methods that are higher order functions: sorted, map, filter, and reduce. These methods use closures to allow us to pass in functionality that can then determine how we want the method to sort, map, filter, or reduce an array of objects.

What is diff b/w map and compactMap

If you need to simply transform a value to another value, then use map . If you need to remove nil values, then use compactMap.

What is reduce higher order function

The reduce function allows you to combine all the elements in an array and return an object of any type (generics). Reduce has two parameters — initialResult and nextPartialResult. We need the initial result to tell us where to start, and the method then operates on that result based on the logic in the closure.

Map vs CompactMap vs FlatMap

Map - Returns an array containing the results of mapping the given closure over the sequence's elements.

CompactMap - Returns an array containing the non-nil results of calling the given transformation with each element of this sequence.

FlatMap - Flatmap is used to flatten a collection of collections

What is Protocol Oriented programming (POP)

Protocol Oriented Programming is a new approach for programming in which you decorate your classes, structs or enums using protocols. Swift does not support multiple inheritance, therefore a problem pops up when you want to add multiple abilities to your

class. Protocol OP lets you add abilities to a class or struct or enum with protocols which support multiple implementations.

Advantage of protocol programming

Testability

Protocols come to the rescue when you want to test something that requires network or requires storing in a database or on disk. One can easily create a mock class implementing the protocol and testing the functionalities that the protocol exposes.

Achieve multiple inheritance

Swift does not allow multiple class inheritance. That also does not mean we put multiple functionalities in the same class, violating the Single Responsibility Principle. With Protocols, we can segregate multiple functionalities into different classes and conform to them.

Inheritance in value types

Structs do not allow inheritance. With Protocols, one can achieve the desired inheritance in structures.

Dependency Injection

By making the parameters conform to a protocol, brings many more benefits like

- achieving a generic behavior where one can inject any type that conforms to a given protocol
- ability to test the class using mocked injected parameters

Reusability

With protocol extensions, we can reuse a piece of code in multiple classes that implement the same functionality without overriding and creating an inheritance chain.

What is protocol

Protocols are a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. It's an interface. A powerful feature of Swift language. Swift checks whether the requirements of the protocol are met for the classes it implements at compile-time. So this allows developers to find out if there are any issues or bugs in code even before running the program. Also, protocols bring more abstraction than classes do in swift.

Can we add variables in protocols

Yes. Variables inside a protocol must be declared as a variable and not a constant(let isn't allowed). Properties inside a protocol must contain the keyword var . We need to explicitly specify the type of property({get} or {get set}).

What is protocol extension

Protocols let you describe what methods something should have, but don't provide the code inside. Extensions let you provide the code inside your methods, but only affect one data type – you can't add the method to lots of types at the same time.

Protocol extensions solve both those problems: they are like regular extensions, except rather than extending a specific type like Int you extend a whole protocol so that all conforming types get your changes.

What is protocol Inheritance

One protocol can inherit from another in a process known as protocol inheritance. Unlike with classes, you can inherit from multiple protocols at the same time before you add your own customizations on top.

What is Associated Types

An associated type can be seen as a replacement of a specific type within a protocol definition. In other words: it's a placeholder name of a type to use until the protocol is adopted and the exact type is specified. This is best explained by a simple code example.

How to create optional methods in Protocols

By default, all methods listed in a Swift protocol must be implemented in a conforming type. However, there are two ways you can work around this restriction depending on your need.

The first option is to mark your protocol using the **@objc** attribute. While this means it can be adopted only by classes, it does mean you mark individual methods as being **optional.**

If possible, the second option is usually better: write default implementations of the optional methods that do nothing