**Q.1 what are the features of swift programming**

**What is Swift Programming Language?**

Swift is a **powerful** and **inherent programming language** for macOS, iOS and watchOS and tvOS. Writing Swift code is **interactive** and **fun,** the syntax is compact yet expressive and includes the modern features that the developers love the most. The code of swift is safe and also produces the software that runs lightning fast.

Also, know **What WIKI says about Swift programming language?**

Swift is a **general-purpose,multi-paradigm, compiled programming language** developed by **Apple Inc** for iOS family. It is **intended to work with Apple’s Cocoa and Cocoa Touch frameworks** and the large body of existing **Objective-C code** written for Apple products. It is developed with an **open source LLVM compiler framework** and has been included in **Xcode**. On platforms other than Linux, it utilises the **objective-c runtime library** which allows C, Objective-C, C++ and swift code to run within one program.

### **What is the role of Swift in developing an iPhone app?**

As swift is a new programming language developed for **iOS** and **OS X mobile app development** by the Apple. This program truly utilises the **best C** and **Objective C** without leaving the **constraints of C-language** that is used to write the application.

###### **Features of Swift Programming Language**

* Swift follows safe and easy programming patterns
* Provides latest programming features
* Syntax will be same as Objective C
* Best way to write iOS and OS X programs
* Quick access to existing cocoa Frameworks
* No need of separate library import to support functionalities like input/output or even the string handling
* Unites the procedural and object-oriented portions of the language
* Runtime is the same for both Objective -C and Swift on Mac OS and iOS

The swift language consists a **safe** and **secure programming features** to make it simpler, easier and fun to work with. Being simple to use, it is the first industrial quality program that is powerful and enjoyable.

**Q.2 What are the collection types available in swift?**

Swift provides three primary collection types for storing collection of values. They are:

* **Arrays →**orderedcollections of values
* **Dictionaries →**unordered collections of key-value pairs/associations
* **Sets →** unordered collections of unique values
* in swift, all these collection types are strict about their types, means we can not insert a value of other type into these which are defined with a particular type. So because of strongly type system, we are sure in retrieving, the element is of the same type of that collection is defined with.
* ***Note*** : *These collections can be declared and defined with any type as long as that type is defined in the project. in Foundation Framework, These collections implemented as generic collections, so they can be defined with any type.*
* *Note*: *The collections which are defined or declared with user defined types are not compile when you write KVO and KVC compile code in swift. Because The Objective -C Run time doesn’t know those user defined types which are declared in swift.*
* **Mutability Of Collections :**
* When you declaring a collection type i.e array, dictionary or set, if you assign it to a variable then that collection will be mutable(can be changed). This means we can change the collection by adding elements, deleting elements, or modifying them. While defining the collection, if you assign it to a constant then it is immutable(its size and content can not be changed).
* **Note**: *it is a good practice to use immutable collections where you do not need to change the collections. Because for the immutable types the compile time performance and optimisation are better than that of mutable counter parts.*

**→ Arrays :**store the values of the same type in ordered list. Same values can appear multiple times at different positions in the list.

Syntax : -

1. **Array<ElementType>**
2. [**ElementType**] //This is shorthand syntax, which we use usually

In the above syntax structure, **ElementType**is the type of values array is allowed to store.

Swift’s Array type also provides an initialiser for creating an array with a certain size, with all of its values set to the same value you pass with initialiser. We pass the value of type we want with the **repeating** parameter and the number of times the value need to repeat in the array with parameter **count**.

**→ Sets :**store the distinct values of the same type in un-ordered list. we prefer these to use when we want to store the unique values and the order is not needed.

A ***type*** must be ***hash-able***in order to be stored in a set. i.e the type must provide a way to compute a hash value for itself. A hash value is an ***Int*** value, that is the same for identical or same objects of the confirmed type. All of Swift’s basic types such as String, Int, Double, and Bool are hashable by default, and can be used as set value types or dictionary key types.

**→ Dictionaries :**A dictionary stores associations between keys of the same type and values of the same type in a collection with no defined ordering. Each value is associated with a unique key, which acts as an identifier for that value within the dictionary. Unlike items in an array, items in a dictionary do not have a specified order.

Syntax : -

1. **Dictionary<key, value>**where Key is the type of value that can be used as a dictionary key, and Value is the type of value that the dictionary stores for those keys.
2. [**key: Value**] //This is shorthand syntax, which we use usually

**Q.3 what are the control transfer statements used in swift?**

## **Loop Statements**

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. Swift has three loop statements: a for-in statement, a while statement, and a repeat-

while statement.

Control flow in a loop statement can be changed by a break statement and a continue statement and is discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) and [Continue Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID442) below.

### **For-In Statement**

A for-in statement allows a block of code to be executed once for each item in a collection (or any type) that conforms to the [Sequence](https://developer.apple.com/documentation/swift/sequence) protocol.

A for-in statement has the following form:

1. for item in collection {
2. statements
3. }

The makeIterator() method is called on the collection expression to obtain a value of an iterator type—that is, a type that conforms to the [IteratorProtocol](https://developer.apple.com/documentation/swift/iteratorprotocol) protocol. The program begins executing a loop by calling the next() method on the iterator. If the value returned isn’t nil, it’s assigned to the item pattern, the program executes the statements, and then continues execution at the beginning of the loop. Otherwise, the program doesn’t perform assignment or execute the statements, and it’s finished executing the for-in statement.

### **While Statement**

A while statement allows a block of code to be executed repeatedly, as long as a condition remains true.

A while statement has the following form:

1. while condition {
2. statements
3. }

A while statement is executed as follows:

1. The condition is evaluated.

If true, execution continues to step 2. If false, the program is finished executing the while statement.

1. The program executes the statements, and execution returns to step 1.

Because the value of the condition is evaluated before the statements are executed, the statements in a while statement can be executed zero or more times.

The value of the condition must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

### **Repeat-While Statement**

A repeat-while statement allows a block of code to be executed one or more times, as long as a condition remains true.

A repeat-while statement has the following form:

1. repeat {
2. statements
3. } while condition

A repeat-while statement is executed as follows:

1. The program executes the statements, and execution continues to step 2.
2. The condition is evaluated.

If true, execution returns to step 1. If false, the program is finished executing the repeat-while statement.

Because the value of the condition is evaluated after the statements are executed, the statements in a repeat-while statement are executed at least once.

The value of the condition must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

## **Branch Statements**

Branch statements allow the program to execute certain parts of code depending on the value of one or more conditions. The values of the conditions specified in a branch statement control how the program branches and, therefore, what block of code is executed. Swift has three branch statements: an if statement, a guard statement, and a switch statement.

Control flow in an if statement or a switch statement can be changed by a break statement and is discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) below.

### **If Statement**

An if statement is used for executing code based on the evaluation of one or more conditions.

There are two basic forms of an if statement. In each form, the opening and closing braces are required.

The first form allows code to be executed only when a condition is true and has the following form:

1. if condition {
2. statements
3. }

The second form of an if statement provides an additional *else clause* (introduced by the else keyword) and is used for executing one part of code when the condition is true and another part of code when the same condition is false. When a single else clause is present, an if statement has the following form:

1. if condition {
2. statements to execute if condition is true
3. } else {
4. statements to execute if condition is false
5. }

The else clause of an if statement can contain another if statement to test more than one condition. An if statement chained together in this way has the following form:

1. if condition 1 {
2. statements to execute if condition 1 is true
3. } else if condition 2 {
4. statements to execute if condition 2 is true
5. } else {
6. statements to execute if both conditions are false
7. }

The value of any condition in an if statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

### **Guard Statement**

A guard statement is used to transfer program control out of a scope if one or more conditions aren’t met.

A guard statement has the following form:

1. guard condition else {
2. statements
3. }

The value of any condition in a guard statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

Any constants or variables assigned a value from an optional binding declaration in a guard statement condition can be used for the rest of the guard statement’s enclosing scope.

The else clause of a guard statement is required, and must either call a function with the Never return type or transfer program control outside the guard statement’s enclosing scope using one of the following statements:

* return
* break
* continue
* throw

Control transfer statements are discussed in [Control Transfer Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID440) below. For more information on functions with the Never return type, see [Functions that Never Return](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID551).

### **Switch Statement**

A switch statement allows certain blocks of code to be executed depending on the value of a control expression.

A switch statement has the following form:

1. switch control expression {
2. case pattern 1:
3. statements
4. case pattern 2 where condition:
5. statements
6. case pattern 3 where condition,
7. pattern 4 where condition:
8. statements
9. default:
10. statements
11. }

**Q. 4 what is optional?**

Swift 4 also introduces **Optionals** type, which handles the absence of a value. Optionals say either "there is a value, and it equals x" or "there isn't a value at all".

An Optional is a type on its own, actually one of Swift 4’s new super-powered enums. It has two possible values, **None** and **Some(T)**, where **T** is an associated value of the correct data type available in Swift 4.

Here’s an optional Integer declaration −

var perhapsInt: Int?

Here’s an optional String declaration −

var perhapsStr: String?

The above declaration is equivalent to explicitly initializing it to **nil** which means no value −

var perhapsStr: String? = nil

Let's take the following example to understand how optionals work in Swift 4

var myString:String? = nil

if myString != nil {

print(myString)

} else {

print("myString has nil value")

}

When we run the above program using playground, we get the following result −

myString has nil value

Optionals are similar to using **nil** with pointers in Objective-C, but they work for any type, not just classes.

## **Forced Unwrapping**

If you defined a variable as **optional**, then to get the value from this variable, you will have to **unwrap** it. This just means putting an exclamation mark at the end of the variable.

Let's take a simple example −

var myString:String?

myString = "Hello, Swift 4!"

if myString != nil {

print(myString)

} else {

print("myString has nil value")

}

When we run the above program using playground, we get the following result −

Optional("Hello, Swift 4!")

Now let's apply unwrapping to get the correct value of the variable

var myString:String?

myString = "Hello, Swift 4!"

if myString != nil {

print( myString! )

} else {

print("myString has nil value")

}

When we run the above program using playground, we get the following result.

Hello, Swift 4!

## **Automatic Unwrapping**

You can declare optional variables using exclamation mark instead of a question mark. Such optional variables will unwrap automatically and you do not need to use any further exclamation mark at the end of the variable to get the assigned value. Let's take a simple example

var myString:String!

myString = "Hello, Swift 4!"

if myString != nil {

print(myString)

} else {

print("myString has nil value")

}

When we run the above program using playground, we get the following result −

Hello, Swift 4!

## **Optional Binding**

Use optional binding to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable.

An optional binding for the **if** statement is as follows −

if let constantName = someOptional {

statements

}

Let's take a simple example to understand the usage of optional binding −

var myString:String?

myString = "Hello, Swift 4!"

if let yourString = myString {

print("Your string has - \(yourString)")

} else {

print("Your string does not have a value")

}

When we run the above program using playground, we get the following result −

Your string has - Hello, Swift

**Q.5 how to define variable and constant in Swift language?n**

## Variables

### Declaring Variables

In Swift, we use the var keyword to declare a variable. While this is similar to how variables are declared in other programming languages, I strongly advise you not to think about other programming languages when using the var keyword in Swift. There are a few important differences.

The var keyword is the only way to declare a variable in Swift. The most common and concise use of the var keyword is to declare a variable and assign a value to it.

|  |  |
| --- | --- |
| 1 | var street = "5th Avenue" |

Remember that we don't end this line of code with a semicolon. While a semicolon is optional in Swift, the best practice is not to use a semicolon if it isn't required.

You also may have noticed that we didn't specify a type when declaring the variable street. This brings us to one of Swift's key features, *type inference*.

### Type Inference

The above statement declares a variable street and assigns the value 5th Avenue to it. If you're new to Swift or you're used to JavaScript or PHP, then you may be thinking that Swift is a typeless or loosely typed language, but nothing could be further from the truth. Let me reiterate that Swift is a strongly typed language. Type safety is one of the cornerstones of the language.

We're just getting started, and Swift already shows us a bit of its magic. Even though the above statement doesn't explicitly specify a type, the variable street is of type String. This is Swift's type inference in action. The value we assign to street is a string. Swift is smart enough to see that and implicitly sets the type of street to String.

The following statement gives us the same result. The difference is that we explicitly set the type of the variable. This statement literally says that street is of type String.

## Constants

Constants are similar to variables in terms of typing. The only difference is that the value of a constant cannot be changed once it has a value. The value of a constant is, well... constant.

### Declaring Constants

To declare a constant, you use the let keyword. Take a look at the following example in which we declare street as a constant instead of a variable.

|  |  |
| --- | --- |
| 1  2  3  4  5 | let street: String = "5th Avenue"  var number: Int    street = "Main Street"  number = 10 |

If we only update the first line, replacing var with let, Xcode throws an error for obvious reasons. Trying to change the value of a constant is not allowed in Swift. Remove or comment out the line in which we try to assign a new value to street to get rid of the error.

|  |  |
| --- | --- |
| 1  2  3  4  5 | let street: String = "5th Avenue"  var number: Int    // street = "Main Street"  number = 10 |

**Q.6 how to connect UI in Swift Language?**

1. Open your storyboard, Main.storyboard.
2. Click the Assistant button in the Xcode toolbar near the top right corner of Xcode to open the [assistant edito](https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/GlossaryDefinitions.html#//apple_ref/doc/uid/TP40015214-CH12-SW76)
3. If you want more space to work, collapse the [project navigator](https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/GlossaryDefinitions.html#//apple_ref/doc/uid/TP40015214-CH12-SW57) and [utility area](https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/GlossaryDefinitions.html#//apple_ref/doc/uid/TP40015214-CH12-SW72) by clicking the Navigator and Utilities buttons in the Xcode toolbar.
4. In the editor selector bar, which appears at the top of the assistant editor, change the assistant editor from Preview to Automatic > ViewController.swift.

5. In ViewController.swift, find the class line, which should look like this:

1. class ViewController: UIViewController{
2. }
3. Below the class line, add the following:
4. //MARK: Properties

You just added a comment to your source code. Recall that a [comment](https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/GlossaryDefinitions.html#//apple_ref/doc/uid/TP40015214-CH12-SW31) is a piece of text in a source code file that doesn’t get compiled as part of the program but provides context or useful information about individual pieces of code.

A comment that begins with the characters //MARK: is a special type of comment that’s used to organize your code and to help you (and anybody else who reads your code) navigate through it. You’ll see this in action later. Specifically, the comment you added indicates that this is the section of your code that lists properties.

1. In your storyboard, select the text field.
2. Control-drag from the text field on your [canvas](https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/GlossaryDefinitions.html#//apple_ref/doc/uid/TP40015214-CH12-SW6) to the code display in the editor on the right, stopping the drag at the line below the comment you just added in ViewController.swift.
3. In the dialog that appears, for Name, type nameTextField. Leave the rest of the options as they are.

Click Connect.

Xcode adds the necessary code to ViewController.swift to store a reference to the text field and configures the storyboard to set up that connection.

1. @IBOutlet weak var nameTextField: UITextField!

**Q. What are different iOS app states?**

[iOS apps](https://itunes.apple.com/us/genre/ios/id36?mt=8) developed to run on version 4.0 or newer has the ability to process in the background. This means, of course, that a developer has many more options for managing memory and streamlining their iOS apps. The operating system keeps constant communication with each iOS app, and passes information to allow the app an opportunity to make the necessary adjustments. It is important for every developer to understand the concept of application state, and the five most important application methods to implement for handling transitions.

## States

Apps developed for early iOS versions (before iOS 4.0) supported three states: non-running, inactive, and active. An application delegate for pre-iOS 4.0 apps received two important method calls: applicationDidFinishLaunching and applicationWillTerminate. When an app received an applicationDidFinishLaunching message, it was an opportunity for information to be retrieved from the previous launch to restore the app to its last used state. The status, applicationWillTerminate, was used to notify the app when the app was preparing to shut down. This gave the developer an opportunity to save any unsaved data or specific state information.

Currently, there are five possible application states that would be cause for the app to prepare for a transition - such as a shutdown or moving to the background. In certain cases, an app might need to continue processing in the background. However, there is certainly no reason for the app to process any graphics, animations, or display-specific routines. The five states of an iOS app - as listed in the [iOS App Programming Guide](http://developer.apple.com/library/ios/#DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html) - include the following:

1. ***Non-running*** - The app is not running.
2. ***Inactive*** - The app is running in the foreground, but not receiving events. An iOS app can be placed into an inactive state, for example, when a call or SMS message is received.
3. ***Active*** - The app is running in the foreground, and receiving events.
4. ***Background*** - The app is running in the background, and executing code.
5. ***Suspended*** - The app is in the background, but no code is being executed.

**Q. 8 Explain difference between class and struct?**

First, let’s take a look at what features [classes](https://learnappmaking.com/swift-class-how-to/) and [structs](https://learnappmaking.com/structs-swift-how-to/) have in common:

* Both structs and classes can define [properties](https://learnappmaking.com/properties-swift-how-to/) to store values, and they can define functions
* They can define subscripts to provide access to values with [subscript syntax](https://learnappmaking.com/subscript-syntax-swift/)
* They can define [initializers](https://learnappmaking.com/initializers-init-swift-how-to/) to set up their initial state, with init()
* They can be [extended](https://learnappmaking.com/swift-extensions-how-to/) with extension (this is important!)
* They can [conform to protocols](https://learnappmaking.com/protocols-swift-how-to/), for example to support Protocol Oriented Programming
* They can [work with generics](https://learnappmaking.com/generics-swift-how-to/) to provide flexible and reusable types

Classes support a few more capabilities that structs don’t have:

* Classes can [inherit](https://learnappmaking.com/swift-inheritance-subclassing-how-to/) from another class, like you inherit from [UIViewController](https://learnappmaking.com/view-controller-uiviewcontroller-ios-swift/) to create your own view controller subclass
* Classes can be deinitialized, i.e. you can invoke a deinit() function before the class is destroyed
* [Classes are reference types and structs are value types](https://learnappmaking.com/swift-value-types-vs-reference-types/)

That last point is important: classes are *reference types*, and structs are *value types*. Here’s what’s what:

* **Value Type:** When you copy a value type (i.e., when it’s assigned, initialized or passed into a function), each instance keeps a unique copy of the data. If you change one instance, the other doesn’t change too.
* **Reference Type:** When you copy a reference type, each instance shares the data. The *reference* itself is copied, but not the data it references. When you change one, the other changes too.

In Swift, structs are value types whereas classes are reference types. When you copy a struct, you end up with two unique copies of the data. When you copy a class, you end up with two references to one instance of the data. It’s a crucial difference, and it affects your choice between classes or structs.

**Q. what are protocols in Swift?**

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

## **Protocol Syntax**

You define protocols in a very similar way to classes, structures, and enumerations:

1. protocol SomeProtocol {
2. // protocol definition goes here
3. }

Custom types state that they adopt a particular protocol by placing the protocol’s name after the type’s name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

1. struct SomeStructure: FirstProtocol, AnotherProtocol {
2. // structure definition goes here
3. }

If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

1. class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
2. // class definition goes here
3. }

## **Property Requirements**

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn’t specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable and settable.

If a protocol requires a property to be gettable and settable, that property requirement can’t be fulfilled by a constant stored property or a read-only computed property. If the protocol only requires a property to be gettable, the requirement can be satisfied by any kind of property, and it’s valid for the property to be also settable if this is useful for your own code.

Property requirements are always declared as variable properties, prefixed with the var keyword. Gettable and settable properties are indicated by writing { get set } after their type declaration, and gettable properties are indicated by writing { get }.

1. protocol SomeProtocol {
2. var mustBeSettable: Int { get set }
3. var doesNotNeedToBeSettable: Int { get }
4. }

Always prefix type property requirements with the static keyword when you define them in a protocol. This rule pertains even though type property requirements can be prefixed with the class or static keyword when implemented by a class:

1. protocol AnotherProtocol {
2. static var someTypeProperty: Int { get set }
3. }

Here’s an example of a protocol with a single instance property requirement:

1. protocol FullyNamed {
2. var fullName: String { get }
3. }

The FullyNamed protocol requires a conforming type to provide a fully qualified name. The protocol doesn’t specify anything else about the nature of the conforming type—it only specifies that the type must be able to provide a full name for itself. The protocol states that any FullyNamed type must have a gettable instance property called fullName, which is of type String.

Here’s an example of a simple structure that adopts and conforms to the FullyNamed protocol:

1. struct Person: FullyNamed {
2. var fullName: String
3. }
4. let john = Person(fullName: "John Appleseed")
5. // john.fullName is "John Appleseed"

This example defines a structure called Person, which represents a specific named person. It states that it adopts the FullyNamed protocol as part of the first line of its definition.

Each instance of Person has a single stored property called fullName, which is of type String. This matches the single requirement of the FullyNamed protocol, and means that Person has correctly conformed to the protocol. (Swift reports an error at compile time if a protocol requirement isn’t fulfilled.)

Here’s a more complex class, which also adopts and conforms to the FullyNamed protocol:

1. class Starship: FullyNamed {
2. var prefix: String?
3. var name: String
4. init(name: String, prefix: String? = nil) {
5. self.name = name
6. self.prefix = prefix
7. }
8. var fullName: String {
9. return (prefix != nil ? prefix! + " " : "") + name
10. }
11. }
12. var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
13. // ncc1701.fullName is "USS Enterprise"

This class implements the fullName property requirement as a computed read-only property for a starship. Each Starship class instance stores a mandatory name and an optional prefix. The fullName property uses the prefix value if it exists, and prepends it to the beginning of name to create a full name for the starship.

**Q. Difference between Optional Chaining and Optional Binding?**

**Optional :**Optional is swift's powerful feature which come to solve problem of  non-existing value. It is just a type . There are two types as Int (which must has a value) and Int?(which may contain a Int value or may have nil value). It is declared as T? i.e Int?, String?  
  
**Forced Unwrapping :**Exclamation mark ( ! ) is used to unwrap value.  
i.e let optionalInt : Int? = 5  
let intVal : Int = 2  
optionalInt! + intVal  
  
So we hvae forecfully unwrap optionaInt. That means we tell compiler that optionalInt has a value and extract and use it.  
  
But this is not good practise. If sometimes optionaInt has not value and we try to unwrap, then app will be crashed. A good practise is to check with nil before unwrapping or use optional binding. It checks it has value or not and if and only if it has value extract it and use it.

**Optional Binding :** You use optional binding to check if the optional contains a value or not. If it does contain a value, unwrap it and put it into a temporary constant or variable.  
  
Example :  
  
var stockCode:String? = findStockCode("Facebook")  
  
let text = "Stock Code - "  
  
if let tempStockCode = stockCode {  
  
    let message = text + tempStockCode  
  
    println(message)  
}  
  
The “if let” (or “if var”) are the two keywords of optional binding. In plain English, the code says “If stockCode contains a value, unwrap it, set its value to tempStockCode and execute the conditional block. Otherwise, just skip it the block”. As the tempStockCode is a new constant, you no longer need to use the ! suffix to access its value.

**Implicitly Unwrapped Optional :**When we are very very sure about it has value after first time it is set, then we need not unwrap every time. So for this type of scenario, we have to use it with ! mark in their type.  
  
// forced unwrapping  
let optionalInt: Int? = 123  
let forcedInt: Int = optionalInt!  
  
// implicitly unwrapped optional  
let assumedInt: Int! = 123  
let implicitInt: Int = assumedInt  
  
It may has nil value.

**Optional Chaining :**  
  
The feature allows us to chain multiple optionals together with the “?.” operator.  
  
  
if let sharePrice = findStockCode("Apple")?.price {  
  
    let totalCost = sharePrice \* 100  
  
    println(totalCost)  
  
}  
  
FindstockCode method returns optional value. We can access multiple optional together using Optional chaining feature.

**Q. What is interpolation?**

This is a fancy name for what is actually a very simple thing: combining variables and constants inside a string.

Clear out all the code you just wrote and leave only this:

**var** name = "Tim McGraw"

If we wanted to print out a message to the user that included their name, string interpolation is what makes that easy: you just write a backslash, then an open parenthesis, then your code, then a close parenthesis, like this:

**var** name = "Tim McGraw"

"Your name is \(name)"

The results pane will now show "Your name is Tim McGraw" all as one string, because string interpolation combined the two for us.

Now, we could have written that using the **+** operator, like this:

**var** name = "Tim McGraw"

"Your name is " + name

…but that's not as efficient, particularly if you're combining multiple variables together. In addition, string interpolation in Swift is smart enough to be able to handle a variety of different data types automatically. For example:

**var** name = "Tim McGraw"

**var** age = 25

**var** latitude = 36.166667

"Your name is \(name), your age is \(age), and your latitude is \(latitude)"

Doing that using **+** is much more difficult, because Swift doesn't let you add integers and doubles to a string.

At this point your result may no longer fit in the results pane, so either resize your window or hover over the result and click the + button that appears to have it shown inline.

One of the powerful features of string interpolation is that everything between **\(** and **)** can actually be a full Swift expression. For example, you can do mathematics in there using operators, like this:

**var** age = 25

"You are \(age) years old. In another \(age) years you will be \(age \* 2)."

**Q. 12 Explain type inference in Swift?**

Swift is a *statically typed language*, meaning that the type of every property, constant and variable that we declare needs to be specified at compile time. However, that’s often not something that needs to be done manually, instead the compiler is able to figure out a wide range of type information on its own — thanks to the fact that Swift supports *type inference*.

So for example, here we’ve declared a few constants — all without specifying any types at all, since the compiler is able to infer that information based on the values that are being assigned:

let number = 42

let string = "Hello, world!"

let array = [1, 1, 2, 3, 5, 8]

let dictionary = ["key": "value"]

For comparison, here’s what the above assignments would look like if we instead were to manually specify the types of each of our constants:

let number: Int = 42

let string: String = "Hello, world!"

let array: [Int] = [1, 1, 2, 3, 5, 8]

let dictionary: [String: String] = ["key": "value"]

So type inference plays a major part in making Swift’s syntax as lightweight as possible, not only when it comes to variable declarations and other kinds of assignments, but in many other kinds of situations as well.

For example, here we’ve defined an enum that describes various kinds of contacts, and a function that lets us load an array of Contact values that belong to a certain kind:

enum ContactKind {

case family

case friend

case coworker

case acquaintance

}

func loadContacts(ofKind kind: ContactKind) -> [Contact] {

...

}

While we would normally refer to a member of the above enum by specifying both the type and the case (such as ContactKind.friend), thanks to type inference, we can completely omit the name of the type when referring to a case within a context in which the type is known — like when calling our above function:

let friends = loadContacts(ofKind: .friend)

What’s really cool is that the above “dot syntax” doesn’t *just* work for enum cases, it works when referencing any *static property or method as well*. For example, here we’ve extended Foundation’s URL type with a static property that creates a URL that points to this very website:

extension URL {

static var swiftBySundell: URL {

URL(string: "https://swiftbysundell.com")!

}

}

Now, when calling any method that accepts a URL parameter (for example the new [Combine](https://www.swiftbysundell.com/basics/combine)-powered [URLSession](https://www.swiftbysundell.com/basics/networking) API), we can simply refer to the above property like this:

let publisher = URLSession.shared.dataTaskPublisher(for: .swiftBySundell)

Really neat! However, while type inference is an incredibly useful feature, there are still situations in which we might need to specify a bit of extra type information in order to achieve our desired result.

A very common example of such a situation is when dealing with numeric types. When a numeric literal is assigned to a variable or constant, it’ll by default be inferred to have the type Int — which is a completely reasonable default — but if we wish to use another numeric type, such as Double or Float, we’ll need to specify those types manually. Here are a few ways to do that:

let int = 42

let double = 42 as Double

let float: Float = 42

let cgFloat = CGFloat(42)

Another type of situation in which we might need to give the compiler some additional type information is when calling a function that has a [generic return type](https://www.swiftbysundell.com/basics/generics).

For example, here we’ve extended the built-in Bundle type with a generic method that lets us easily load and decode any JSON file that we’ve bundled within our app:

extension Bundle {

struct MissingFileError: Error {

var name: String

}

func decodeJSONFile<T: Decodable>(named name: String) throws -> T {

guard let url = self.url(forResource: name, withExtension: "json") else {

throw MissingFileError(name: name)

}

let data = try Data(contentsOf: url)

let decoder = JSONDecoder()

return try decoder.decode(T.self, from: data)

}

}

To learn more about Swift’s built-in error handling mechanism (that we use above through the throws and try keywords), check out [the Basics article about error handling](https://www.swiftbysundell.com/basics/error-handling).

Now let’s say that during the development of our app, until our real server and networking code is up and running, we wish to decode an instance of the following User type from a bundled JSON file:

struct User: Codable {

var name: String

var email: String

var lastLoginDate: Date

}

However, if we were to call our decodeJSONFile method like this, we’d end up with a compiler error:

// Error: Generic parameter 'T' could not be inferred

let user = try Bundle.main.decodeJSONFile(named: "user-mock")

That’s because the exact type that we’ll decode any given JSON file into depends on what the generic type T will actually refer to at each call site — and since we’re not giving the compiler any such information above, we’ll end up with an error. There’s simply no way for the compiler to know that we wish to decode a User instance in this case.

To fix that problem, we can use the same sort of techniques that we used above to specify different kinds of numeric values, and either give our user constant an explicit type, or use the as keyword — like this:

let user: User = try Bundle.main.decodeJSONFile(named: "user-mock")

let user = try Bundle.main.decodeJSONFile(named: "user-mock") as User

However, if we were to call our decodeJSONFile method within a context in which our desired return type *is known*, then we could once again let Swift’s type inference mechanism figure that information out for us — like in this case, in which we’ve defined a wrapper struct called MockData that has a User-typed property that we’re assigning our result to:

struct MockData {

var user: User

}

let mockData = try MockData(

user: Bundle.main.decodeJSONFile(named: "user-mock")

)

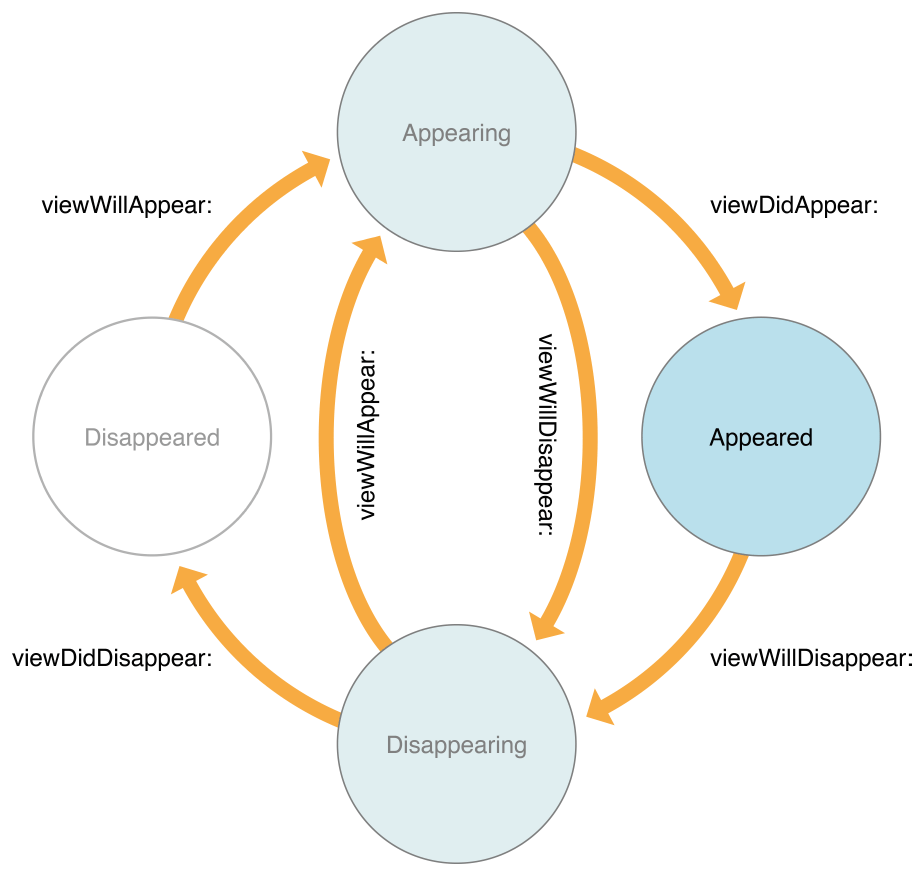
So that’s a quick intro to Swift’s type inference capabilities. It’s also important to point out that type inference does have a computation cost associated with it, which thankfully takes place entirely at compile-time (so it won’t effect the runtime performance of our apps), but it’s still something that can be good to keep in mind when dealing with more complex expressions. However, if we do encounter an expression that takes the compiler a long time to figure out, then we could always use one of the above techniques for specifying those types manually.

**Q. Explain the different life cycle methods of uiviewcontroller.**

There is three way to create UI (Which attached with a viewController class ).

1. From the .xib
2. From Code
3. By Storyboard

View controller Life Cycle :



## **loadView**

This method use when view Controller create from code .Its good not to do anything on this method .If view Controller made from .xib or storyboard.

What Do in View Load : loadView( ) is a method managed by the viewController. The viewController calls it when its current view is nil. loadView( ) basically takes a view (that you create) and sets it to the viewController’s view (superview).

## **viewDidLoad:**

This Method is loaded once in view controller life cycle .Its Called When all the view are loaded .You Can do Some common task in this method :

1.Network call which need Once.

2.User Interface

3.Others Task Those are Need to do Once

Note: This Method Call before the bound are defined and rotation happen.So its Risky to work view size in this method.

## **viewWillAppear:**

This Method is called every time before the view are visible to and before any animation are configured .In this method view has bound but orientation not set yet.You can override this method to perform custom tasks associated with displaying the view such as to hide fields or disable actions before the view becomes visible.

## **viewWillLayoutSubviews:**

It don’t do Nothing by default. When a view’™s bounds change, the view adjusts the position of its subviews. View controller can override this method to make changes before the view lays out its subviews.

## **viewDidLayoutSubviews:**

This method is called after the viewController has been adjust to its subview following a change on its bound.Add code here if you want to make change to subviews after they have been set.

## **viewDidAppear:**

This Method is called after the view present on the screen. Usually save data to core data or start animation or start playing a video or a sound, or to start collecting data from the network This type of task good for this method.

## **viewWillDisappear:**

This method called before the view are remove from the view hierarchy.The View are Still on view hierarchy but not removed yet . any unload animations haven’t been configured yet. Add code here to handle timers, hide the keyboard, cancel network requests, revert any changes to the parent UI. Also, this is an ideal place to save state.

## **viewDidDisappear:**

This method is called after the VC’s view has been removed from the view hierarchy. Use this method to stop listening for notifications or device sensors.

## **deinit:**

Before a view controller is removed from memory, it gets deinitialized. You usually override deinit() to clean resources that the view controller has allocated that are not freed by ARC. Keep in mind that just because a VC is no longer visible, doesn’t mean that it has been deallocated. Container view controllers such as NavigationController can keep their VCs available in memory. Keep in mind that even though a VC is off screen, if it is still in memory, it still works normally and can receive notifications.

## **didReceiveMemoryWarning()**

When memory starts to fill up, iOS does not automatically move data from memory to its limited hard disk space. It does, however, issue this warning and YOU (I mean YOU) are responsible for clearing some objects out of memory. Be aware that if the memory of your app goes over a certain threshold, iOS will shutdown your app. Unfortunately, this will look like a crash to the end user.

## **viewWillTransition(to:with:)**

When the interface orientation changes, UIKit calls this method on the window’s root view controller before the size changes are about to be made. The root view controller then notifies its child view controllers, propagating the message throughout the view controller hierarchy. The parameter to contains the new CGSize size of the container’s view and the parameter with contains a UIViewControllerTransitionCoordinator coordinator, an enum that describes the new orientation.

**Q. 14 What is an app bundle?**

## **Application Bundles**

Application bundles are one of the most common types of bundle created by developers. The application bundle stores everything that the application requires for successful operation. Although the specific structure of an application bundle depends on the platform for which you are developing, the way you use the bundle is the same on both platforms.

### **What Files Go Into an Application Bundle?**

Table 2-1 summarizes the types of files you are likely to find inside an application bundle. The exact location of these files varies from platform to platform and some resources may not be supported at all. For examples and more detailed information, see the platform-specific bundle sections in this chapter.

Top of Form

Bottom of Form

### **What Files Go Into an Application Bundle?**

Table 2-1 summarizes the types of files you are likely to find inside an application bundle. The exact location of these files varies from platform to platform and some resources may not be supported at all. For examples and more detailed information, see the platform-specific bundle sections in this chapter.

|  |  |
| --- | --- |
| **Table 2-1**  Types of files in an application bundle | |
| **File** | **Description** |
| Info.plist file | (Required) The information property list file is a structured file that contains configuration information for the application. The system relies on the presence of this file to identify relevant information about your application and any related files. |
| Executable | (Required) Every application must have an executable file. This file contains the application’s main entry point and any code that was statically linked to the application target. |
| Resource files | Resources are data files that live outside your application’s executable file. Resources typically consist of things like images, icons, sounds, nib files, strings files, configuration files, and data files (among others). Most resource files can be localized for a particular language or region or shared by all localizations.  The placement of resource files in the bundle directory structure depends on whether you are developing an iOS or Mac app. |
| Other support files | Mac apps can embed additional high-level resources such as private frameworks, plug-ins, document templates, and other custom data resources that are integral to the application. Although you can include custom data resources in your iOS application bundles, you cannot include custom frameworks or plug-ins. |

Although most of the resources in an application bundle are optional, this may not always be the case. For example, iOS applications typically require additional image resources for the application’s icon and default screen. And although not explicitly required, most Mac apps include a custom icon instead of the default one provided by the system.

### **Anatomy of an iOS Application Bundle**

The project templates provided by Xcode do most of the work necessary for setting up the bundle for your iPhone or iPad application. However, understanding the bundle structure can help you decide where you should place your own custom files. The bundle structure of iOS applications is geared more toward the needs of a mobile device. It uses a relatively flat structure with few extraneous directories in an effort to save disk space and simplify access to the files.

**Q. 15 What are generics?**

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

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you’ve been using generics throughout the *Language Guide*, even if you didn’t realize it. For example, Swift’s Array and Dictionary types are both generic collections. You can create an array that holds Int values, or an array that holds String values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

## **The Problem That Generics Solve**

Here’s a standard, nongeneric function called swapTwoInts(\_:\_:), which swaps two Int values:

1. func swapTwoInts(\_ a: inout Int, \_ b: inout Int) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }

This function makes use of in-out parameters to swap the values of a and b, as described in [In-Out Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID173).

The swapTwoInts(\_:\_:) function swaps the original value of b into a, and the original value of a into b. You can call this function to swap the values in two Int variables:

1. var someInt = 3
2. var anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5. // Prints "someInt is now 107, and anotherInt is now 3"

The swapTwoInts(\_:\_:) function is useful, but it can only be used with Int values. If you want to swap two String values, or two Double values, you have to write more functions, such as the swapTwoStrings(\_:\_:) and swapTwoDoubles(\_:\_:) functions shown below:

1. func swapTwoStrings(\_ a: inout String, \_ b: inout String) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }
6. func swapTwoDoubles(\_ a: inout Double, \_ b: inout Double) {
7. let temporaryA = a
8. a = b
9. b = temporaryA
10. }

You may have noticed that the bodies of the swapTwoInts(\_:\_:), swapTwoStrings(\_:\_:), and swapTwoDoubles(\_:\_:) functions are identical. The only difference is the type of the values that they accept (Int, String, and Double).

It’s more useful, and considerably more flexible, to write a single function that swaps two values of *any* type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

NOTE

In all three functions, the types of a and b must be the same. If a and b aren’t of the same type, it isn’t possible to swap their values. Swift is a type-safe language, and doesn’t allow (for example) a variable of type String and a variable of type Double to swap values with each other. Attempting to do so results in a compile-time error.

**Q.16 Difference between designated and convenience initialiser?**

## **Designated Initializers**

Like most constructs in Swift, designated initializers are aptly named and do exactly what they say they do.  They are the main initializers to be used for a class.  A class must have one designated initializer, but it is not limited to one.  It can have multiple if necessary, but most classes only have one.  
  
Let’s take a look at a designated initializer:

init(sender: String, recipient: String) {

self.sender = sender

self.recipient = recipient

timeStamp = Date()

}

Does that look a bit familiar?  If you’ve been following the blog for a while it should, that is the initializer for our Message class used in the previous articles [Classes In Swift — An Introduction](http://www.codingexplorer.com/classes-in-swift-an-introduction/) and [Using a Nested Type in Swift](http://www.codingexplorer.com/using-nested-type-swift/).  That syntax is to create a designated initializer.  Our Message class only had one initializer, so it had to be the designated one.  To create a designated initializer, it is just the init keyword, with the parameters afterwards, and then the code inside the curly braces.

## **Convenience Initializers**

Convenience initializers are also aptly named, and they are initializers used to make initialization a bit easier.  Designated initializers tend to set all of the properties up and let the user send in values for each.  A convenience initializer often has some of those hard coded, and thus can take less parameters.  The developer usually write’s a convenience initializer to set some defaults that are appropriate to a special use case.

The initializer above probably should have been a convenience initializer, since it sets timeStamp without giving the user any input to set one themselves.  Like I said above, I think that designated initializers tend to have parameters for all properties they are setting (unless derived from some of the other parameters).  This is just personal preference though, the above one is perfectly valid as a designated initializer, since we said in those posts that the timeStamp is set when the Message is created.

Nonetheless, let’s add a convenience initializer to our Swift app.  Let’s say that this app could also send a message to oneself, to use it as a note perhaps.  In this case, we wouldn’t need to set the sender and the recipient to different values.  We would only be repeating ourself in the initializer when doing that, so lets write a convenience initializer that takes only one parameter:

convenience init(sender: String) {

self.init(sender: sender, recipient: sender)

}

The main difference in their syntax, is that convenience initializers have the convenience keyword before the init.  Otherwise they are exactly the same.  Once inside our convenience initializer that only takes the sender as a parameter, we just give that parameter to the designated initializer as both the sender and the recipient, since this Message is being used more like a note.

**Q.17 What is the reuseidentifier used for?**

# reuseIdentifier

A string used to identify a cell that is reusable.

## Declaration

var reuseIdentifier: [String](https://developer.apple.com/documentation/swift/string)? { get }

## Discussion

The reuse identifier is associated with a UITableViewCell object that the table-view’s delegate creates with the intent to reuse it as the basis (for performance reasons) for multiple rows of a table view. It is assigned to the cell object in [init(frame:reuseIdentifier:)](https://developer.apple.com/documentation/uikit/uitableviewcell/1623218-initwithframe) and cannot be changed thereafter. A [UITableView](https://developer.apple.com/documentation/uikit/uitableview) object maintains a queue (or list) of the currently reusable cells, each with its own reuse identifier, and makes them available to the delegate in the [dequeueReusableCell(withIdentifier:)](https://developer.apple.com/documentation/uikit/uitableview/1614891-dequeuereusablecell) method.

**Q.18 What is a category and when is it used?**

**Categories** and extensions allow you to extend the **functionality** of existing classes without subclassing (inherit nothing) adding functionality to an existing class, even to one for which you do not have the source.

A category allow you to add (only) methods to a class by declaring them in an interface file (.h) and defining them in an implementation file (.m), like in a basic Objective-C class. Sadly a category can’t declare additional instance variable for a class.

Now this declared methods become part of the categorized-class!!!

There’s no limit to the number of categories that you can add to a categorized-class, but each category name must be different should declare and define a different set of methods.

 In objective c, when you want to add some more functionality to a class without inheritance, you simply use category for it.

-> Category use to add new method not properties

**Q.19 Explain Closures?**

*Closures* are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Closures can capture and store references to any constants and variables from the context in which they’re defined. This is known as *closing over* those constants and variables. Swift handles all of the memory management of capturing for you.

NOTE

Don’t worry if you aren’t familiar with the concept of capturing. It’s explained in detail below in [Capturing Values](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID103).

Global and nested functions, as introduced in [Functions](https://docs.swift.org/swift-book/LanguageGuide/Functions.html), are actually special cases of closures. Closures take one of three forms:

* Global functions are closures that have a name and don’t capture any values.
* Nested functions are closures that have a name and can capture values from their enclosing function.
* Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

Swift’s closure expressions have a clean, clear style, with optimizations that encourage brief, clutter-free syntax in common scenarios. These optimizations include:

* Inferring parameter and return value types from context
* Implicit returns from single-expression closures
* Shorthand argument names
* Trailing closure syntax

**Q.20 What is the Responder Chain?**

## **The Responder Chain**

As mentioned in the beginning, UIKit handles this by dynamically managing a linked list of **UIResponders**. The so called **first responder** is simply the root element of the list, and if a responder can't handle a specific action/event, the action is recursively sent to the next responder of the list until someone can handle the action or the list ends.

Although inspecting the actual first responder is protected by a private **firstResponder** property in **UIWindow**, you can check the Responder Chain for any given responder by checking the **next** property:

**extension UIResponder {**

**func responderChain() -> String {**

**guard let next = next else {**

**return String(describing: self)**

**}**

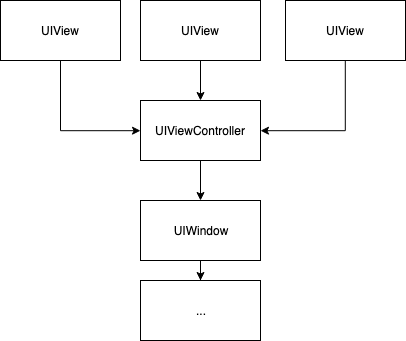
**return String(describing: self) + " -> " + next.responderChain()**

**}**

**}**

**myViewController.view.responderChain()**

**// MyView -> MyViewController -> UIWindow -> UIApplication -> AppDelegate**



In the previous example where the action was handled by the **UIViewController**, UIKit first sent the action to the **UIView** first responder - but since it doesn't implement **myCustomMethod** the view forwarded the action to the next responder - the **UIViewController** which happened to have that method in its implementation.

While in most cases the Responder Chain is simply be the order of the subviews, you can customize it to change the general flow order. Besides being able to override the **next** property to return something else, you can force an **UIResponder** to become the first responder by calling **becomeFirstResponder()** and have it go back to its position by calling **resignFirstResponder()**. This is commonly used in conjunction with **UITextField** to display a keyboard - **UIResponders** can define an optional **inputView** property that only shows up when the responder is the first responder, which is the keyboard in this case.

* 1. 62.21
  2. 58.63
  3. 59.44
  4. 68.31

**Q. 21 Explain difference between sqlite and Core Data?**

## **What Is SQLite**

SQLite is a library that implements a lightweight database engine that is incredibly performant and, therefore, a great fit for embedded systems, such as mobile devices. Even though SQLite is advertised as a relational database, it is important to realize that the developer is in charge of maintaining the relationships between records stored in the database.

Like many other databases, SQLite stores data and that is what it is good at. The SQLite [website](http://sqlite.org/mostdeployed.html) claims its the most used database in the world.

## **What Is Core Data**

The most important difference between Core Data and SQLite is that SQLite is a database while Core Data is not. That is the most important difference because there is very little to compare. Core Data and SQLite are solutions to different problems.

Core Data can use SQLite as its persistent store, but the framework itself is not a database.

Core Data is not a database.

Core Data is a framework for managing an object graph. An object graph is nothing more than a collection of interconnected objects. The framework excels at managing complex object graphs.

Core Data manages an object graph.

The framework is responsible for managing the life cycle of the objects in the object graph. It can optionally persist the object graph to disk and it also offers a powerful interface for searching the object graph it manages.

But Core Data is much more than that. The framework adds a number of other compelling features, such as input validation, data model versioning, and change tracking.

## **Persistent Store Types**

I already mentioned that Core Data can use a SQLite database as its persistent store, but it also has support for other persistent store types, including a binary store and an in-memory store.

Even though Core Data knows how to use a SQLite database as its persistent store, that doesn't mean you can hand it any SQLite database. The database schema of the SQLite database used by Core Data is an implementation detail of the framework. It isn't publicly documented and liable to change.

## **Core Data or SQLite**

What should you use? Core Data or SQLite? Once again, that is the wrong question to ask. If you need a lightweight solution and don't need Core Data's feature set, then SQLite may fit your needs. If you don't like C, then you may want to consider [Gus Mueller's FMDB library](https://github.com/ccgus/fmdb). It provides an object-oriented wrapper written in Objective-C.

But if you are managing a complex object graph with many entities, attributes, and relationships, then Core Data is definitely worth considering. Getting up to speed with Core Data is easier than you might think.

If you decide to opt for Core Data, make sure you don't skip the basics of the framework. Core Data has a learning curve, but it isn't as bad as many make you believe it is.

**Q.22 What are “strong” and “weak” references?**

The key difference between a strong and a weak or unowned reference is that a strong reference prevents the class instance it points to from being deallocated. That is very important to understand and remember.

**ARC** keeps track of the number of strong references to a class instance. This is also known as the **retain count** of a class instance. The class instance cannot be deallocated as long as at least one strong reference points to the class instance. That is, in a nutshell, what **ARC** is about.

Weak and unowned references also point to class instances. The difference is that **ARC** doesn't take these references into account to determine when it is safe to deallocate a class instance. In other words, weak and unowned references cannot prevent a class instance from being deallocated.

### **What Is a Weak Reference?**

In the [previous episode](https://cocoacasts.com/how-to-break-a-strong-reference-cycle/), we discovered that the delegate property of the UITableView class is defined as weak. You already know why that is and why it isn't defined as strong. But why is the delegate property of the UITableView class defined as weak? Why isn't it defined as unowned?

A reference should be marked as weak if it is possible that the class instance the reference points to is deallocated at some point in the future. This means that a weak reference can become nil during its lifetime. This has several consequences that are important to keep in mind.

First, **ARC** automatically sets a weak reference to nil when the class instance it points to is deallocated. That is a good thing. You don't want to have a reference pointing to a class instance that no longer exists.

Second, because **ARC** sets a weak reference to nil when the class instance is deallocated, weak references always need to be declared as optionals.

Third, this implies that a weak reference should always be declared as a variable. We discussed this in the [previous episode](https://cocoacasts.com/how-to-break-a-strong-reference-cycle/).

### **What Is an Unowned Reference?**

Unowned references differ in several aspects from weak references and they shouldn't be used interchangeably. An unowned reference is always expected to point to a class instance. That is essential to understand and remember. Because an unowned reference should always point to a class instance, it isn't necessary to declare an unowned reference as an optional. This makes unowned references more convenient than weak references, which are required to be of an optional type.

In the previous episode, we declared the account property of the Plan class as unowned. The account property doesn't need to be declared as an optional. Remember that the reason for defining the account property of the Plan class as unowned is to avoid a strong reference cycle.

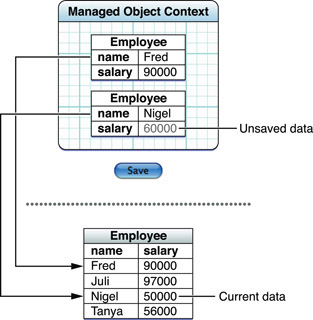
Unowned references come with a big warning, though. You are responsible for making sure an unowned reference always points to a class instance. If you attempt to access an unowned reference that no longer points to a class instance, a runtime error is thrown and your application is terminated.

**Q.23 Describe managed object context and is function.**

# Managed object context

A managed object context represents a single object space, or scratch pad, in a Core Data application. A managed object context is an instance of NSManagedObjectContext. Its primary responsibility is to manage a collection of managed objects. These managed objects represent an internally consistent view of one or more persistent stores. The context is a powerful object with a central role in the life-cycle of managed objects, with responsibilities from life-cycle management (including faulting) to validation, inverse relationship handling, and undo/redo.

From your perspective, the context is the central object in the Core Data stack. It’s the object you use to create and fetch managed objects, and to manage undo and redo operations. Within a given context, there is at most one managed object to represent any given record in a persistent store.



A context is connected to a parent object store. This is usually a persistent store coordinator, but may be another managed object context. When you fetch objects, the context asks its parent object store to return those objects that match the fetch request. Changes that you make to managed objects are not committed to the parent store until you save the context.

In some applications, you may have just a single context. In others, however, you might have more than one. You might want to maintain discrete sets of managed objects and edits to those objects; or you might want to perform a background operation using one context while allowing the user to interact with objects in another.

**Q.24 what is plist?**

An information property list file is a structured text file that contains essential configuration information for a bundled executable. The file itself is typically encoded using the Unicode UTF-8 encoding and the contents are structured using XML. The root XML node is a dictionary, whose contents are a set of keys and values describing different aspects of the bundle. The system uses these keys and values to obtain information about your app and how it is configured. As a result, all bundled executables (plug-ins, frameworks, and apps) are expected to have an information property list file.

By convention, the name of an information property list file is Info.plist. This name of this file is case sensitive and must have an initial capital letter I. In iOS apps, this file resides in the top-level of the bundle directory. In macOS bundles, this file resides in the bundle’s Contents directory. Xcode typically creates this file for you automatically when you create a project of an appropriate type.

**Important:** In the sections that follow, pay attention to the capitalization of files and directories that reside inside a bundle. The [NSBundle](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Classes/NSBundle/Description.html" \l "//apple_ref/occ/cl/NSBundle" \t "_self) class and Core Foundation bundle functions consider case when searching for resources inside a bundle directory. Case mismatches could prevent you from finding your resources at runtime.

**Q.25 What is the difference between an “app ID” and a “bundle ID” and what is each used for?**

You don't find the App ID in Xcode. It is an object (not just a string) in Member Center. Xcode is where the Bundle ID is found. It is specified when creating a new project. (screenshot below) Each Target in a project will have a unique Bundle ID. It can be found/changed by selecting the project in the Project Navigator the choosing the Target and either General or Info from the Jump Bar.

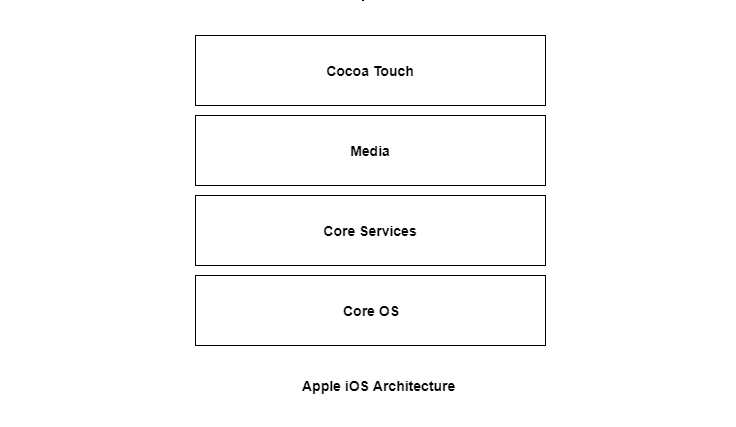
It uniquely defines each App. It is specified in Xcode. A single Xcode project can have multiple Targets and therefore output multiple apps. A common use case for this is an app that has both lite/free and pro/full versions or is branded multiple ways.

**Q.26 Explain iOS architecture?**

The iOS is the operating system created by Apple Inc. for mobile devices. The iOS is used in many of the mobile devices for apple such as iPhone, iPod, iPad etc. The iOS is used a lot and only lags behind Android in terms of popularity.

The iOS architecture is layered. It contains an intermediate layer between the applications and the hardware so they do not communicate directly. The lower layers in iOS provide the basic services and the higher layers provide the user interface and sophisticated graphics.

The layered architecture of iOS is given as follows −



## Layers in iOS Architecture

The different layers as shown in the above diagram are given as follows −

### Core OS

All the iOS technologies are build on the low level features provided by the Core OS layer. These technologies include Core Bluetooth Framework, External Accessory Framework, Accelerate Framework, Security Services Framework, Local Authorisation Framework etc.

### Core Services

There are many frameworks available in the cure services layer. Details about some of these are given as follows −

### Cloudkit Framework

The data can be moved between the app the iCloud using the Cloudkit Framework.

### Core Foundation Framework

This provides the data management and service features for the iOS apps.

### Core Data Framework

The data model of the model view controller app is handled using the Core Data Framework.

### Address Book Framework

The address book framework provides access to the contacts database of the user.

### Core Motion Framework

All the motion based data on the device is accessed using core motion framework.

### Healthkit Framework

The health related information of the user can be handled by this new framework.

### Core Location Framework

This framework provides the location and heading information to the various apps.

## Media

The media layer enables all the graphics, audio and video technology of the system. The different frameworks are:

### UIKit Graphics

This provides support for designing images and animating the view content.

### Core Graphics Framework

This provides support for 2-D vector and image based rendering and is the native drawing engine for iOS apps.

### Core Animation

The Core Animation technology optimizes the animation experience of the apps.

### Media Player Framework

This framework provides support for playing playlists and enables the user to use their iTunes library.

### AV Kit

This provides various easy to use interfaces for video presentation.

## Cocoa Touch

The cocoa touch layer provides the following frameworks −

### EventKit Framework

This shows the standard system interfaces using view controllers for viewing and changing calendar related events.

### GameKit Framework

This provides support for users to share their game related data online using Game center.

### MapKit Framework

This provides a scrollable map which can be included into the app user interface.

**Q.27 which are the ways of achieving concurrency in ios?**

There are 3 ways of achieving concurrency in iOS

* Dispatch queues.
* Threads.
* Operation queues.

**Q. 28 explain the purpose of interface builder what is NIB file.**

A nib file is a special type of resource file that you use to store the user interfaces of iOS and Mac apps. A nib file is an Interface Builder document. You use Interface Builder to design the visual parts of your app—such as windows and views—and sometimes to configure nonvisual objects, such as the controller objects that your app uses to manage its windows and views. In effect, as you edit an Interface Builder document, you create an object graph that is then archived when you save the file. When you load the file, the object graph is unarchived.

The nib file—and hence the object graph—may contain placeholder objects that are used to refer to objects that live outside of the document but that may have references to objects in the document, or to which objects in the document may have references. A special placeholder is the File’s Owner.

At runtime, you load a nib file using the method loadNibNamed:owner: or a variant thereof. The File’s Owner is a placeholder in the nib file for the object that you pass as the owner parameter of that method. Whatever connections you establish to and from the File’s Owner in the nib file in Interface Builder are reestablished when you load the file at runtime.

iOS uses nibs as an implementation detail that supports storyboards, the iOS user interface design layout format. Storyboards allow you to design and visualize the entire user interface of your app on one canvas. For iOS developers, using storyboards is the recommended way to design user interfaces.

**Q.29 What kind of notifications does iOS offer?**

# **View and respond to notifications on iPhone**

Notifications help you keep track of what’s new—they let you know if you missed a call, if the date of an event moved, and more. You can customize your notification settings so you see only what’s important to you. View and respond to notifications on the iPhone Lock Screen or in Notification Center.

## **Find all your notifications in one place**

iPhone displays notifications as they arrive, but if you don’t read one right away, it’s saved in Notification Center so you can check it later.

To see your notifications in Notification Center, do any of the following:

* On the Lock Screen: Swipe up from the middle of the screen.
* On other screens: Swipe down from the top center. Then you can scroll up to see older notifications, if there are any.

To close Notification Center, swipe up from the bottom with one finger or press the Home button (on an iPhone with a Home button).

## **Respond to notifications**

When you have multiple notifications in Notification Center or on the Lock Screen, they’re grouped by app, which makes them easier to view and manage. Notifications from some apps may also be grouped by organizing features within the app, such as by topic or thread. Grouped notifications appear as small stacks, with the most recent notification on top.

Do any of the following:

* To expand a group of notifications to see them individually, tap the group. To close the group again, tap Show Less.
* Touch and hold a notification to view it and perform quick actions if the app offers them (on supported models).
* Tap a notification to open the app that it’s from.

## **Dismiss, clear, and manage notifications**

Do any of the following:

* Handle a notification you receive while using another app: Pull it down to view it, then swipe up to dismiss it.
* Clear notifications: Swipe left over the notification or group of notifications, then tap Clear or Clear All.
* Send notifications directly to Notification Center: Swipe left over the notification or group of notifications, tap Manage, then tap Deliver Quietly. This prevents notifications from this app or group from appearing on the Lock Screen, playing a sound, lighting up the screen, or presenting a banner.

To see and hear these notifications again, swipe left on a notification in Notification Center, tap Manage, then tap Deliver Prominently.

* Turn off notifications for an app or notification group: Swipe left on a notification or group of notifications, tap Manage, then tap Turn Off.
* Change how an app displays notifications: Swipe left on a notification, tap Manage, tap Settings, then choose an option. You can choose whether to allow notifications from the app, where the notifications appear (in Notification Center, for example), whether to play an alert sound, and so on.
* Clear all your notifications in Notification Center: Tap the Clear Notifications button, then tap Clear.

When you haven’t used an app for a while, Siri suggests that you turn off notifications for that app.

**Q. 30 What is provisioning profile?**

Now to tie it all together and enable your app to run, you need to use Provisioning Profiles✉️. A provisioning profile specifies a Bundle Identifier, so the system knows which app the permission is for, a certificate, with the information who created the app, and it’s defined in which ways the app can be distributed. The provisioning profile is therefore a combination of the App ID, the Bundle ID, the device ID or list of device IDs and the certificates (Development or Distribution).

**Q. 31 what are the differences between development and production ios signing certificates?**

If you are testing a push notifications in your app that time you easily check through debugging mode for this purpose you need Development SSL certificate. If you uploaded the app into App Store that time you need to add Production SSL certificate (for live app push notifications purpose you want to add production SSl).

**Q. 32 Explain retain counts.**

Retain Count represents number of owners for a particular object. It is zero till object does not have any owners.

Increase in one ownership claim will cause **retain count**to **increase by 1** and decrease will cause it to **decrement by 1**.

**Q.33 What is difference between frame and bound?**

All **UIView** subclasses have two properties that at first glance seem similar: **frame** and **bounds**. Both return a **CGRect** – a rectangle containing their X and Y position, plus their width and height – but that doesn’t mean they are the *same*.

At its simplest, a view’s bounds refers to its coordinates relative to its own space (as if the rest of your view hierarchy didn’t exist), whereas its frame refers to its coordinates relative to its parent’s space.

This means a few things:

1. If you create a view at X:0, Y:0, width:100, height:100, its **frame** and **bounds** are the same.
2. If you move that view to X:100, its frame will reflect that change but its bounds will not. Remember, the bounds is relative to the view’s own space, and internally to the view nothing has changed.
3. If you transform the view, e.g. rotating it or scaling it up, the frame will change to reflect that, but the bounds still won’t – as far as the view is concerned internally, it hasn’t changed.

When you change the width or height of either **frame** or **bounds**, the other value is updated to match. Generally it’s better to modify **bounds** plus **center** and **transform**, and let UIKit calculate the **frame** for you.

**Q.34 Explain how the push notifications works.**

**Apple Push Notification service (APNs) propagates push notifications to devices having applications registered to receive those notifications. Each device establishes an accredited and encrypted IP connection with the service and receives notifications over this persistent connection. Providers connect with APNs through a persistent and secure channel while monitoring incoming data intended for their client applications. When new data for an application arrives, the provider prepares and sends a notification through the channel to APNs, which pushes the notification to the target device.**