**Chap -1**

**1] Explain different collections available in swift(array, set, dictionary)**

**1. Swift Dictionary**

A Swift dictionary is a simple container that can contain multiple data as key-value pairs in an unordered way.

A Swift dictionary is used instead of the array when you want to look up value with some identifier in the collection. Suppose, we have to search for the capital city of the country. In this case, we will create a dictionary with key countries and value capital city. Now, you get the capital city from the collection by searching with the key country. Here, we have paired a country with its capital city.

Declaring an empty dictionary

To create an empty dictionary, we specify the key: value Data type inside square brackets [ ].

Example:

let emptyDictionary:[Int: String] = [:]

print(emptyDictionary)

Or

let emptyDictionary: Dictionary<Int, String> = [:]

print(emptyDictionary)

Output:

[:]

Declaring a dictionary with some values

let valDictionary = ["a":10, "b":20, "c":30, "d":40, "e":50, "f":60, "g":70, "h":80, "i":90]

print(valDictionary)

Output:

["c": 30, "d": 40, "g": 70, "b": 20, "a": 10, "f": 60, "h": 80, "i": 90, "e": 50]

In the above program, we have not declared the type explicitly but we initialize with some default elements. The element is in key: value pair where the key is of type String and the value is of Int type.

Creating Dictionary from two arrays

Example:

let customKeys = ["India", "Pakistan", "United Kingdom"]

let customValues = ["New Delhi", "Islamabad", "London"]

Let newDictionary = Dictionary(uniqueKeysWithValues: zip(customKeys,customValues))

print(newDictionary)

Output:

["India": "New Delhi", "United Kingdom": "London", "Pakistan": "Islamabad"]

Access Dictionary elements in Swift

We can access elements of a Swift dictionary by including key of the value we want to access within square brackets immediately after the name of the dictionary.

Example:

let valDictionary = ["a":10, "b":20, "c":30, "d":40, "e":50, "f":60, "g":70, "h":80, "i":90]

print(valDictionary["c"])

print(valDictionary["f"])

Output:

Optional(30)

Optional(60)

Access Dictionary elements in Swift using for-in loop

Example:

let valDictionary = ["a":10, "b":20, "c":30, "d":40, "e":50, "f":60, "g":70, "h":80, "i":90]

for (key, value) in valDictionary {

print("key:\(key) value:\(value)")

}

Output:

key:h value:80

key:d value:40

key:a value:10

key:g value:70

key:e value:50

key:b value:20

key:c value:30

key:f value:60

key:i value:90

Modify dictionary elements in Swift

We can add a new element in a Swift dictionary by using a new key as an index and assign to a new value.

Example:

var valDictionary = ["India":"New Delhi", "China":"Beijing", "USA":"Washington D.C."]

valDictionary["Germany"] = "Berlin"

print(valDictionary)

Output:

["Germany": "Berlin", "China": "Beijing", "USA": "Washington D.C.", "India": "New Delhi"]

Changing elements in Dictionary

Example:

var valDictionary = ["India": "New Delhi", "China": "Beijing", "USA": "Washington D.C."]

valDictionary["India"] = "NEW DELHI"

print(valDictionary)

Output:

["USA": "Washington D.C.", "China": "Beijing", "India": "NEW DELHI"]

**2. Swift Arrays**

Arrays are used to store multiple values of the same data type. An Array is just like a container which holds multiple data/ values of a data type in an ordered list. When you fetch the data, you will get the elements in the same order as you defined in the array.

An array can store values of any data type e.g. Int, String, class etc.

Specify the data type inside the square brackets [] to create an empty array. You have to specify the type inside the square bracket [], otherwise Swift will treat it as a normal data type and you can store only a single value in it.

Declare an empty array

Syntax:

let emptyIntArr:[Int] = []

print(emptyIntArr)

Or

let emptyIntArr:Array<Int> = Array()

print(emptyIntArr)

Output:

[ ]

Here, we have declared a constant emptyIntArr that can store an array of integer and initialized with 0 values.

Create an array directly

Swift is a type inference language so, we can also create an array directly without specifying the data type but we have to initialize with some values so that compiler can find out its type.

Example:

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

print(someIntArr)

Output:

[1, 2, 3, 4, 5]

Here, we have declared a constant someIntArr that can store an array of Integer without specifying the type explicitly. Also, we have initialized an array which gives the output as 1, 2, 3, 4, 5.

Declare an array with a specified number of single repeated value

We can repeat a value to a specific number of times to make an array in Swift. It is done by using the array initializer with repeating and counting.

Example:

let arrWithRepeatingValues = Array(repeating: "JavaTpoint", count: 5)

print(arrWithRepeatingValues)

Output:

["JavaTpoint", "JavaTpoint", "JavaTpoint", "JavaTpoint", "JavaTpoint"]

Here, we have defined a constant arrWithRepeatingValues that stores an array of string JavaTpoint and repeats the same value for 5 times as specified in the count.

Let's take a constant IntArray to store some strings:

let intArray = [10, 11, 12, 13, 14, 15]

The index of an array starts with 0 means the first element is stored in index 0, the second element in the index (1) and so on.

You have to use the index of the value of that array you want to access within square brackets immediately after the name of the array.

Suppose, we declared an array intArray as above. The first element is intArray[0], the second element is intArray[1], and so on. We use them to access the array's values.

Example:

let intArr = [10, 11, 12, 13, 14]

print(intArr[0])

print(intArr[1])

print(intArr[2])

print(intArr[3])

print(intArr[4])

Output:

10

11

12

13

14

We can modify elements of the array by using the assignment operator. We have to add the index of the value we want to update within square brackets after the name of the array followed by the assignment operator and new value.

Example:

var intArray = [10, 11, 12, 13, 14]

intArray[0] = 12

intArray[1] = 42

intArray[2] = 45

intArray[3] = 21

intArray[3] = 36

print(intArray)

Output:

[12, 42, 45, 36, 14]

You can also modify the whole array completely.

Example:

var intArr = [1, 2, 3, 4]

intArr = [10,20,30, 40]

print(intArr)

Output:

[10, 20, 30, 40]

A list of built-in Array functions and their properties:

| Index | Array function | Explanation |
| --- | --- | --- |
| 1. | isEmpty | The isEmpty property is used to determine whether an array is empty or not. It returns true if the array does not contain any value otherwise, returns false. |
| 2. | first | This property is used to access the first element of an array. |
| 3. | append | The append function is used to insert/append elements at the end of the array. |
| 4. | insert | This function is used to insert/append elements at the specific indexes of the array. |
| 5. | remove | This function removes and returns the value specified at the specific position from the array. |
| 6. | reversed | This function returns the elements of an array in reverse order. |
| 7. | count | This property returns the total number of elements in an array. |

**3. Swift Sets**

A Swift set is a simple container that can hold multiple values of a data type in an unordered list. It can't store the same value twice. Here, the unordered list means you won't get the items in the same manner in which you have entered them in the set.

Values stored in a Swift set must be hashable. Hashable means, it must provide a hashValue property. HashValue is used to access the elements of the sets because sets are unordered.

All Swift basic data types (such as String, Int, Double, and Bool) are hashable by default and can be used as set value types. We can also create our Hashable Type in Swift that can be stored in a set.

We can declare a set that can store some values.

Example:

let someIntSet:Set = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(someIntSet)

Output:

[2, 4, 9, 5, 6, 7, 3, 1, 8]

Declare a set containing duplicate value

let someIntSet:Set = [1, 2, 3, 4, 8, 8, 8, 5, 5]

print(someIntSet)

Output:

[2, 4, 5, 3, 1, 8]

In the above program, we have entered the repeated value of 5 and 8. But in the output, we can see that the repeated values have been removed.

**Access set elements in Swift**

We cannot access elements of the set in the same way we access elements of an array in Swift because set elements are unordered.

So, we have to access set elements using their methods and properties or using for-in loops.

**Access all set elements in Swift**

Example:

var someIntSet:Set = [1, 2, 3, 4, 5, 6]

for val in someIntSet {

print(val)

}

Output:

5

6

2

3

1

4

Using remove() method to access selective set elements

We can access the selective elements from the set by using remove() method.

Example:

var someIntSet:Set = [1, 2, 3, 4, 5, 6]

let someVal = someIntSet.remove(5)

print(someVal)

print(someIntSet)

Output:

Optional(5)

[6, 2, 3, 1, 4]

In the above program, you can see that the remove() method returns an optional string. It is recommended to do optional handling as follows.

Optional handling for using remove() method

Example:

var someIntSet:Set = [1, 2, 3, 4, 5, 6]

if let someVal = someIntSet.remove(5) {

print(someVal)

print(someIntSet)

} else {

print("cannot find element to remove")

}

Output:

5

[6, 2, 3, 1, 4]

Add new elements in a set in Swift

The insert() method is used to add new elements in set in Swift.

To add single element

Example:

var someIntSet:Set = [1, 2, 3, 4, 5, 6]

someIntSet.insert(7)

print(someIntSet)

Output:

[2, 4, 5, 6, 7, 3, 1]

In the above example, we have added a new element 7 in a predefined set.

To add multiple elements

Example:

var someIntSet:Set = [1, 2, 3, 4, 5, 6]

someIntSet.insert(7)

someIntSet.insert(8)

someIntSet.insert(9)

someIntSet.insert(10)

print(someIntSet)

Output:

[10, 2, 4, 9, 5, 6, 7, 3, 1, 8]

Set operations in Swift

Another advantage of using set is that it facilitates you to perform set operations, i.e. union, intersection etc. It is same as set operation in Mathematics.

Union Set Operation

The union of two sets a and b is the set of all elements combined in set a and set b. It is accessed by using a.union(b).

Example:

let a: Set = [1, 3, 5, 7, 9]

let b: Set = [0, 2, 4, 6, 8]

print(a.union(b))

Output:

[8, 2, 9, 4, 5, 7, 6, 3, 1, 0]

Intersection Set Operation

The intersection of two sets a and b is the set of elements which are common in both sets. It is accessed by using a.intercection(b).

Example:

let a: Set = [1, 3, 5, 7, 9]

let b: Set = [0, 3, 4, 7, 8]

print(a.intersection(b))

Output:

[7, 3]

Subtracting Set Operation

The subtraction of two sets a and b is the set which contains all elements of a but excludes the elements that also belongs to b. It is accessed by using a.subtracting(b).

Example:

let a: Set = [1, 3, 5, 7, 9]

let b: Set = [0, 3, 4, 7, 8]

print(a.subtracting(b))

Output:

[5, 9, 1]

Some Built-in functions and properties

| Index | Built-in Set functions | Explanation |
| --- | --- | --- |
| 1. | isEmpty | The isEmpty property is used to determine if a set is empty or not. It returns true if a set is empty otherwise, returns false. |
| 2. | first | The first property is used to access the first element of a set. |
| 3. | insert | The insert function is used to insert/append elements in the set. |
| 4. | reversed | This function returns the elements of a set in reverse order. |
| 5. | count | This property returns the total number of elements in a set. |
| 6. | removeFirst | This function removes and returns the first value from the set. |

**2] Inheritance in swift**

Inheritance and polymorphism are fundamental concepts in Object-Oriented Programming, including Objective-C and Swift on iOS.

Inheritance is a mechanism that allows you to create a new class based on an existing class, inheriting the properties and behavior of the parent class. In Swift, yo

class Subclass: Superclass {

}

The subclass inherits all the properties and methods of the superclass and can add new properties and methods or override existing ones.

Polymorphism is the ability of objects to take on multiple forms, depending on the context in which they are used. In Swift, you can achieve polymorphism through the use of dynamic dispatch, where the actual method implementation is determined at runtime based on the object's class. For example:

let object: Superclass = Subclass()

object.doSomething()

In this example, the object variable is of type Superclass, but it is pointing to an instance of Subclass. The doSomething method will be dynamically dispatched to the implementation in the Subclass, even though the object variable is declared as a Superclass type.

In summary, inheritance allows you to create new classes based on existing classes, while polymorphism allows objects to take on multiple forms based on their context. These concepts are fundamental to object-oriented programming in both Objective-C and Swift on iOS.

Polymorphism is a fundamental concept in Object-Oriented Programming that allows objects to take on multiple forms, depending on the context in which they are used. In Objective-C and Swift on iOS, polymorphism is achieved through the use of dynamic dispatch, which means that the actual method implementation is determined at runtime based on the object's class.

For example, suppose we have a superclass called Animal, and two subclasses called Cat and Dog, both of which inherit from Animal. Each subclass can have its own implementation of a method called makeSound, which is defined in the Animal superclass. Here is an example of how polymorphism works in Objective-C:

Animal \*myPet;

if (isCat) {

myPet = [[Cat alloc] init];

} else {

myPet = [[Dog alloc] init];

}

[myPet makeSound];

In this example, the type of the myPet variable is Animal, but it is assigned either a Cat or Dog object depending on the value of the isCat variable. When the makeSound method is called on myPet, the implementation that is executed is determined at runtime based on the actual class of myPet. If myPet is a Cat object, the implementation of makeSound in the Cat subclass will be executed. If myPet is a Dog object, the implementation of makeSound in the Dog subclass will be executed.

Similarly, here's an example of polymorphism in Swift:

var myPet: Animal

if isCat {

myPet = Cat()

} else {

myPet = Dog()

}

myPet.makeSound()

This code is similar to the Objective-C example, but uses Swift syntax. Again, the type of the myPet variable is Animal, but it is assigned either a Cat or Dog object depending on the value of the isCat variable. When the makeSound method is called on myPet, the implementation that is executed is determined at runtime based on the actual class of myPet.

In summary, polymorphism allows objects to take on multiple forms and is achieved through the use of dynamic dispatch in both Objective-C and Swift on iOS.

**3] run time polymorphism**

Run-time polymorphism is a powerful feature of Object-Oriented Programming that allows objects to take on multiple forms at run-time. The main advantage of run-time polymorphism in Objective-C is that it enables the creation of more flexible and extensible code. By using run-time polymorphism, it becomes easier to write code that can work with a wide range of objects, even if they have different types or are derived from different classes.

The implementation of run-time polymorphism in Objective-C is achieved through the use of dynamic dispatch. This means that the method implementation to be executed is determined at runtime, based on the actual class of the object. This allows for more flexibility, since the code does not need to know the exact class of an object at compile-time.

Here's an example of how run-time polymorphism works in Objective-C:

@interface Shape: NSObject

- (void)draw;

@end

@implementation Shape

- (void)draw {

NSLog(@"Drawing a generic shape");

}

@end

@interface Circle: Shape

@end

@implementation Circle

- (void)draw {

NSLog(@"Drawing a circle");

}

@end

@interface Square: Shape

@end

@implementation Square

- (void)draw {

NSLog(@"Drawing a square");

}

@end

int main(int argc, char \* argv[]) {

@autoreleasepool {

Shape \*shape;

shape = [[Circle alloc] init];

[shape draw]; // Drawing a circle

shape = [[Square alloc] init];

[shape draw]; // Drawing a square

shape = [[Shape alloc] init];

[shape draw]; // Drawing a generic shape

}

return 0;

}

In this example, we have a superclass called Shape, and two subclasses called Circle and Square, both of which inherit from Shape. Each subclass overrides the draw method of the Shape class with its own implementation.

In the main method, we create instances of Circle and Square and assign them to a Shape variable. When we call the draw method on the Shape object, the implementation that is executed is determined at runtime based on the actual class of the object. This means that the correct implementation of the draw method is selected automatically, based on the actual class of the object.

In summary, the advantages of run-time polymorphism in Objective-C are increased flexibility and extensibility, as well as the ability to write code that can work with a wide range of objects, even if they have different types or are derived from different classes. This is achieved through the use of dynamic dispatch, which determines the method implementation to be executed at runtime, based on the actual class of the object.

**4] Initializers in swift (constructor)**

In Swift, initializers are used to create and configure instances of a class, structure, or enumeration. There are different types of initializers available in Swift, and each has a specific purpose. Here are some important concepts related to initializers in Swift:

1. init: This keyword is used to define an initializer for a class, structure, or enumeration. The init method is called when a new instance of the object is created, and it is used to set up the initial state of the object.
2. self: This keyword refers to the current instance of a class, structure, or enumeration. It is used within instance methods to refer to the properties and methods of the current instance. For example, self.propertyName is used to access the value of a property within an instance method.
3. super: This keyword is used to call a method or initializer of the superclass. It is used within a subclass to call a method or initializer that is defined in the superclass. For example, super.init() is used to call the initializer of the superclass.

In addition to the above concepts, there are different types of initializers in Swift that can be used to create instances of an object. These include:

1. Designated Initializer: A designated initializer is the primary initializer for a class, structure, or enumeration. It is responsible for initializing all properties of the object, and it must call the designated initializer of its superclass to ensure that all properties are initialized properly.
2. Convenience Initializer: A convenience initializer is a secondary initializer for a class, structure, or enumeration. It is used to provide a shortcut for creating an instance of an object with default values. A convenience initializer must call a designated initializer of its own class to ensure that all properties are initialized properly.
3. Failable Initializer: A failable initializer is an initializer that can return nil if it fails to initialize the object. It is used to handle situations where the object cannot be initialized due to missing or invalid data. Failable initializers are defined with the init? keyword.
4. Required Initializer: A required initializer is an initializer that must be implemented by all subclasses of a class. It is used to ensure that all subclasses provide a specific initializer implementation. A required initializer is defined with the required init() keyword.

**5] protocol and extension eg**

Protocols and extensions are important concepts in iOS development that help in creating flexible and reusable code. Here is an explanation of each of them with examples:

1. Protocols: A protocol defines a blueprint of methods, properties, and other requirements that can be adopted by a class, structure, or enumeration. It is used to define a set of rules or standards that a class or other types must follow in order to conform to the protocol.

For example, let's say we want to create a protocol to represent a musical instrument. Here is how we can define the protocol:

protocol Instrument {

var name: String { get set }

var numberOfStrings: Int { get }

func play()

}

In the above example, we have defined a protocol named Instrument that requires any object that conforms to it to implement three things: name, numberOfStrings, and play(). The name and numberOfStrings properties are defined as read-only and read-write respectively using the get and get set keywords. The play() method is defined without an implementation, as it is up to the conforming class to provide its own implementation.

2. Extensions: An extension is a way to add new functionality to an existing class, structure, or enumeration, without the need to subclass it. It is used to extend the capabilities of an existing type, making it more versatile and reusable.

For example, let's say we want to add a new method to the Int type that returns the square of the number. Here is how we can define the extension:

extension Int {

func squared() -> Int {

return self \* self

}

}

In the above example, we have defined an extension to the Int type, which adds a new method named squared(). The method takes no parameters and returns the square of the number. The method can be called on any instance of the Int type, like this:

let number = 5

let squaredNumber = number.squared() // Returns 25

Extensions can be used to add new properties, methods, and initializers to an existing type. They can also be used to conform to a protocol or to provide default implementations for a protocol.

**6] What is an interface builder**

* Interface Builder is a graphical tool in Xcode, the integrated development environment (IDE) for iOS and macOS app development, that enables developers to design and lay out the user interface (UI) of an app visually, without having to write code.
* Interface Builder provides a drag-and-drop interface for adding UI elements, such as buttons, labels, text fields, and images, to a storyboard or a nib file.
* The developer can then customize the appearance and behaviour of these elements using a variety of options in the Attributes inspector.
* Interface Builder also enables developers to create connections between UI elements and code using IBOutlet and IBAction.
* By control-dragging from a UI element to the code file, the developer can create these connections and write code that responds to user input or updates the UI dynamically.
* Additionally, Interface Builder supports auto layout, a powerful system for laying out UI elements that enables them to adapt to different screen sizes and orientations.
* With the auto layout, developers can create a UI that looks great on any device, from the smallest iPhone to the largest iPad.
* Overall, Interface Builder is an essential tool for iOS development, enabling developers to create beautiful and intuitive user interfaces for their apps quickly and easily.

**Chap - 2**

**1] What is a storyboard? Advantages and Disadvantages**

A storyboard is a visual representation of the user interface (UI) of an iOS app, consisting of a sequence of screens, or view controllers, that the user navigates through to accomplish a task or access content. Storyboards are created and edited using Interface Builder, a graphical tool in Xcode.

**Advantages of Storyboards:**

1. Visual representation: Storyboards provide a visual representation of the app's UI, making it easier for developers to design, modify, and test the app's navigation flow and user experience.
2. Faster development: By providing a drag-and-drop interface for adding UI elements and creating connections between UI elements and code, storyboards can speed up the development process and reduce the amount of code needed.
3. Collaboration: Storyboards enable developers, designers, and stakeholders to collaborate more effectively by providing a shared understanding of the app's UI and behavior.
4. Segues: Storyboards simplify the creation of segues between view controllers, making it easy to create different types of transitions, such as push, modal, and popover segues.

**Disadvantages of Storyboards:**

1. Large files: Storyboards can become large and complex, making them more difficult to manage and maintain over time. This can result in slower performance and longer build times.
2. Merge conflicts: When multiple developers work on the same storyboard, merge conflicts can occur, making it challenging to resolve differences and merge changes.
3. Limited reuse: Storyboards are typically designed for a single app and cannot be reused across multiple apps or projects.
4. Steep learning curve: While storyboards can speed up the development process, they can also have a steep learning curve, requiring developers to learn new tools and techniques for designing and implementing UI elements.

Overall, storyboards can be a powerful tool for designing and implementing the UI of an iOS app. However, like any development tool, they have their advantages and disadvantages and should be used judiciously based on the needs and constraints of the project.

**2] what are IBAction and IBOutlet**

* IBAction and IBOutlet are two important concepts in iOS development that are used to connect user interface elements to code.
* An IBOutlet is a reference to an object in the user interface that can be accessed by code. In other words, an IBOutlet is a way for code to "see" and interact with an object in the user interface, such as a button or a label. IBOutlet is typically used for read and write operations.
* On the other hand, an IBAction is a method that gets triggered in response to a user action on a user interface element, such as tapping a button. An IBAction is essentially a way for code to respond to user input. IBAction is typically used for writing operations, such as changing the text of a label or updating a data model.
* To connect a user interface element to code, the developer can use Interface Builder, a graphical tool in Xcode. By control-dragging from a user interface element to the code file, the developer can create an IBOutlet or an IBAction, depending on the type of connection needed.
* Overall, IBOutlet and IBAction are two important concepts in iOS development that enable developers to create interactive and dynamic user interfaces in their apps.

**3] Write steps to implement animation using an image view**

Here are the steps to implement animation using an image view in iOS:

1. Create a new Xcode project and add an image to the project that you want to animate.
2. Drag an Image View from the Object Library into your view controller's view.
3. Select the Image View and in the Attributes Inspector, set the image property to the image you want to animate.
4. Create an IBOutlet for the Image View in your view controller to reference it in your code.
5. In your view controller's viewDidLoad method, set the Image View's animationImages property to an array of images that will be used in the animation.
6. Set the Image View's animationDuration property to the duration you want the animation to take.
7. Set the Image View's animationRepeatCount property to the number of times you want the animation to repeat (set it to 0 for infinite repetition).
8. To start the animation, call the startAnimating() method on the Image View.
9. To stop the animation, call the stopAnimating() method on the Image View.

Here is some sample code

class ViewController: UIViewController {

@IBOutlet weak var imageView: UIImageView!

override func viewDidLoad() {

super.viewDidLoad()

imageView.animationImages = [UIImage(named: "image1")!, UIImage(named: "image2")!, UIImage(named: "image3")!]

imageView.animationDuration = 1.0

imageView.animationRepeatCount = 0

}

@IBAction func startAnimation(\_ sender: Any) {

imageView.startAnimating()

}

@IBAction func stopAnimation(\_ sender: Any) {

imageView.stopAnimating()

}

}

In this code, the animationImages property is set to an array of three images named "image1", "image2", and "image3". The animationDuration property is set to 1 second and the animationRepeatCount is set to infinite. The startAnimation() and stopAnimation() methods are called from buttons in the view controller's view to start and stop the animation.

**4] Different types of gestures**

Working with different types of gestures in iOS involves detecting the gestures, recognizing them, and performing the desired action based on the type of gesture recognized. Here are some examples of working with different types of gestures in iOS:

1. **Tap Gesture**

* A tap gesture is a single touch on the screen that is recognized as a tap.
* To work with a tap gesture in iOS, you can create a UITapGestureRecognizer instance, add it to the view that you want to recognize the tap gesture on, and set up a handler method that is called when the tap is recognized.

let tapGesture = UITapGestureRecognizer(target: self, action: #selector(handleTap(\_:)))

view.addGestureRecognizer(tapGesture)

@objc func handleTap(\_ gestureRecognizer: UITapGestureRecognizer) {

// Handle tap gesture here

}

**2. Swipe Gesture**

* A swipe gesture is a quick horizontal or vertical movement of the finger on the screen.
* To work with a swipe gesture in iOS, you can create a UISwipeGestureRecognizer instance, add it to the view that you want to recognize the swipe gesture on, and set up a handler method that is called when the swipe is recognized.

let swipeGesture = UISwipeGestureRecognizer(target: self, action: #selector(handleSwipe(\_:)))

swipeGesture.direction = .right

view.addGestureRecognizer(swipeGesture)

@objc func handleSwipe(\_ gestureRecognizer: UISwipeGestureRecognizer) {

// Handle swipe gesture here

}

**3. Pinch Gesture**

* A pinch gesture is a two-finger gesture that is used to zoom in or out on the screen.
* To work with a pinch gesture in iOS, you can create a UIPinchGestureRecognizer instance, add it to the view that you want to recognize the pinch gesture on, and set up a handler method that is called when the pinch is recognized.

let pinchGesture = UIPinchGestureRecognizer(target: self, action: #selector(handlePinch(\_:)))

view.addGestureRecognizer(pinchGesture)

@objc func handlePinch(\_ gestureRecognizer: UIPinchGestureRecognizer) {

// Handle pinch gesture here

}

**4. Long Press Gesture**

* A long press gesture is a touch on the screen that is held for a certain period of time.
* To work with a long press gesture in iOS, you can create a UILongPressGestureRecognizer instance, add it to the view that you want to recognize the long press gesture on, and set up a handler method that is called when the long press is recognized.

let longPressGesture = UILongPressGestureRecognizer(target: self, action: #selector(handleLongPress(\_:)))

view.addGestureRecognizer(longPressGesture)

@objc func handleLongPress(\_ gestureRecognizer: UILongPressGestureRecognizer) {

// Handle long press gesture here

}

* By working with different types of gestures in iOS, you can create more engaging and interactive user interfaces that respond to user input in a more natural way.

**Chap - 3**

**1] what is Segue**

In iOS development, a segue is a transition between two view controllers in an app's user interface. A segue defines a transition from one view controller to another by creating a visual link between the two.

In other words, a segue is a way to move from one screen to another in an iOS app. Segues can be triggered by user actions, such as tapping a button, or they can be triggered programmatically by the app's code.

There are different types of segues available in iOS development, such as push segues, modal segues, and unwind segues, each with its own specific behaviour and purpose.

Overall, segues are an essential component of iOS app development, allowing developers to create smooth and intuitive user interfaces that enable users to navigate through an app's content easily.

**2] What is table view? Write steps to implement table view in iOS**

— A table view displays a list of data in our application.

— It is used to display a vertically scrollable view which consists of a number of cells that are reusable

Steps to implement tableview

1. Create a new iOS single-view application and set the required properties

2. Add a tableview from the object library to your main.storyboard

3. Set the data source and delegate for tableview

4. Inherit UITableViewDataSource and UITableViewDelegate protocol

5. Add the method Stubs of UITableViewDataSource and UITableViewDelegate

6. Create IBOutlet for tableview and add some data to display using a static array

7. Implement the tableview datasource and data delegate

7.1 Number of sections in the tableview method

7.2 Implement the number of rows in each section method

7.3 Implement the cell for a row at the index path method to fill up the table view with data source

7.4 Implement didSelectrow method to select an item from tableview and use it somewhere

**3] What is Picker view? Write steps to implement Picker view in iOS**

* A picker view is a user interface element in iOS that allows the user to select a value from a list of predefined options.
* The picker view consists of one or more columns, each containing a set of rows with values that the user can select by scrolling.
* UIPickerView is a view controller in iOS that displays a set of options in a spinning wheel or a list-like format.
* It is often used to allow users to select from a pre-defined set of options or values.

Here's an example of how to implement a UIPickerView in iOS using Swift:

1. Create a new iOS single-view application and set the required properties

2. Add a UIPickerView from the object library to your main.storyboard

3. Create an IBOutlet for the UIPickerView in the view controller file:

@IBOutlet weak var pickerView: UIPickerView!

4. Set the data source and delegate for PickerView

override func viewDidLoad() {

super.viewDidLoad()

pickerView.dataSource = self

pickerView.delegate = self

}

5. Inherit UIPickerViewDataSource and UIPickerViewDelegate protocol

6. Add the method Stubs of UIPickerViewDataSource and UIPickerViewDelegate (Implement the UIPickerViewDataSource and UIPickerViewDelegate protocols in the view controller file)

class ViewController: UIViewController, UIPickerViewDataSource, UIPickerViewDelegate {

let ~ = ["Option 1", "Option 2", "Option 3"]

// MARK: - UIPickerViewDataSource

func numberOfComponents(in pickerView: UIPickerView) -> Int {

return 1

}

func pickerView(\_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {

return options.count

}

// MARK: - UIPickerViewDelegate

func pickerView(\_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {

return options[row]

}

func pickerView(\_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {

print("Selected option: \(options[row])")

}

}

* In this example, we have created an array of options and set the numberOfRowsInComponent method to return the number of options in the array.
* We have also set the titleForRow method to return the title for each row, which is the value of each option in the array.
* Finally, we have implemented the didSelectRow method to print the selected option to the console.
* When you run the app, you should see a UIPickerView with three options. When you select an option, its title will be printed to the console.

**4] What is a View controller? write steps to share data between 2 screens**

A view controller in iOS is a fundamental building block of an app's user interface, responsible for managing the display and interaction of a single screen or view. Each view controller is associated with a specific view hierarchy and is responsible for updating the contents of the view and responding to user interaction events.

Here are the steps to share data between two screens using a view controller:

1. Create two view controllers in the storyboard, one for each screen.
2. Create a property in the first view controller to hold the data that needs to be shared, such as a string or an array.
3. In the first view controller, create a segue to the second view controller using Interface Builder.
4. Implement the prepare(for : sender:) method in the first view controller to pass the data to the second view controller before the segue is performed. For example:

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

if segue.identifier == "showSecondScreenSegue" {

let secondVC = segue.destination as! SecondViewController

secondVC.data = self.data

}

}

1. In the second view controller, create a property to hold the data received from the first view controller, and update the UI accordingly. For example

class SecondViewController: UIViewController {

var data: String?

@IBOutlet weak var dataLabel: UILabel!

override func viewDidLoad() {

super.viewDidLoad()

dataLabel.text = data

}

}

1. Run the app and test the data sharing by entering data in the first screen, navigating to the second screen, and verifying that the data is displayed correctly.

Overall, sharing data between two screens using a view controller involves creating properties to hold the data in both view controllers, passing the data using a segue and the prepare(for:sender:) method, and updating the UI in the second view controller based on the received data.

**ViewController**

**import** UIKit

**class** ViewController: UIViewController {

**@IBOutlet** **weak** **var** textbox: UITextField!

**override** **func** viewDidLoad() {

**super**.viewDidLoad()

}

**@IBAction** **func** button(\_ sender: **Any**) {

performSegue(withIdentifier: "s1", sender:**self**)

}

**override** **func** prepare(for segue: UIStoryboardSegue, sender: **Any**?) {

**if**(segue.identifier=="s1")

{

**let** dest = segue.destination **as**! ViewController1

dest.str=textbox.text!

}

}

}

ViewController 2 for display label

**import** UIKit

**class** ViewController1: UIViewController {

**@IBOutlet** **weak** **var** label: UILabel!

**var** str=String()

**override** **func** viewDidLoad() {

**super**.viewDidLoad()

label.text=str

// Do any additional setup after loading the view.

}

**5] What is tableview? how to Insert items into tableview**

UITableView is a UI component in iOS that displays a scrollable list of cells in a single column. Each cell typically contains text, images, or custom views.

To insert items into a UITableView, you need to follow these steps:

1. Implement the UITableViewDataSource protocol in your view controller class. This protocol provides methods that the table view uses to retrieve the data to be displayed.
2. In the viewDidLoad() method of your view controller, set the table view's dataSource property to self (i.e., the view controller instance).
3. Implement the tableView(\_:numberOfRowsInSection:) method to return the number of rows (i.e., items) that should be displayed in the table view.
4. Implement the tableView(\_:cellForRowAt:) method to configure and return each cell. This method is called once for each visible row in the table view.
5. When you want to add a new item to the table view, you can add it to your data source (e.g., an array) and then call the table view's insertRows(at:with:) method to insert a new row into the table view.

Here is some sample code that demonstrates how to insert a new item into a table view:

// Assume that self.items is an array of items to be displayed in the table view

// Add a new item to the array

let newItem = "New Item"

self.items.append(newItem)

// Insert a new row into the table view

let indexPath = IndexPath(row: self.items.count - 1, section: 0)

self.tableView.insertRows(at: [indexPath], with: .automatic)

* In this example, a new item is added to the items array and then a new row is inserted into the table view at the end of the section (section 0) using the insertRows(at:with:) method.
* The .automatic animation option specifies that the table view should animate the insertion of the new row.

**6] What is Date Picker view? Write steps to implement Date Picker view in iOS**

* A Date Picker view is a graphical user interface element that allows the user to select a date and/or time value.
* In iOS, a Date Picker view is implemented using the UIDatePicker class, which provides a variety of customization options for selecting date and time values.
* UIDatePicker is a view controller in iOS that allows users to select dates and times.
* It is often used in forms, scheduling apps, and other apps that require users to input dates or times.

Here's how to implement a UIDatePicker in iOS using Swift:

1. Create a new Xcode project and select "Single View App" as the template.
2. Open the Main.storyboard file and drag a UIDatePicker onto the view controller.
3. Create an IBOutlet for the UIDatePicker in the view controller file:

@IBOutlet weak var datePicker: UIDatePicker!

4. In the viewDidLoad() method, set the mode of the UIDatePicker to .date or .time depending on your requirements:

override func viewDidLoad() {

super.viewDidLoad()

datePicker.datePickerMode = .date // or .time

}

5. Optionally, you can set the minimum and maximum dates for the UIDatePicker:

let calendar = Calendar(identifier: .gregorian)

let minimumDate = calendar.date(byAdding: .year, value: -1, to: Date())

let maximumDate = calendar.date(byAdding: .year, value: 1, to: Date())

datePicker.minimumDate = minimumDate

datePicker.maximumDate = maximumDate

6. Implement the dateChanged method to handle changes in the selected date or time:

@IBAction func dateChanged(\_ sender: UIDatePicker) {

let formatter = DateFormatter()

formatter.dateStyle = .medium // or .short or .long

formatter.timeStyle = .none // or .short or .long

let dateString = formatter.string(from: sender.date)

print("Selected date: \(dateString)")

}

* In this example, we have implemented the dateChanged method to print the selected date to the console.
* We have also set the date style and time style of the DateFormatter to customize the date format displayed.
* When you run the app, you should see a UIDatePicker with either a date or time selection depending on your requirements.
* When you select a date or time, its formatted string will be printed to the console.

**Chap - 4**

**1] What is a core data stack(architecture)? Crud operation steps**

Core Data is a framework provided by Apple that helps developers manage the model layer objects in their iOS, macOS, watchOS, and tvOS apps. A Core Data stack is a set of objects and technologies that work together to provide access to a persistent data store.

The Core Data stack consists of three main objects:

1. Managed Object Model: A blueprint for your app's data model. It defines the types of objects that can be stored in the persistent store and their relationships.
2. Persistent Store Coordinator: Manages the persistent store that stores your app's data on disk. It provides a layer of abstraction between the managed object context and the persistent store.
3. Managed Object Context: Represents a scratch pad for objects in memory. It is responsible for tracking changes to objects, managing object relationships, and fetching objects from the persistent store.

The Core Data stack is initialized when your app launches, and it provides a convenient API for performing CRUD (Create, Read, Update, Delete) operations on your app's data.

Here are the steps to perform CRUD operations in Core Data:

Create:

1. Create a new instance of your managed object class.
2. Set the properties of the managed object to the desired values.
3. Add the managed object to the managed object context using the insert(\_:) method.
4. Save the changes to the persistent store using the save() method of the managed object context.

Read

1. Fetch the desired objects from the persistent store using a fetch request.
2. Process the fetched objects as needed.

Update:

1. Fetch the desired objects from the persistent store using a fetch request.
2. Update the properties of the fetched objects as needed.
3. Save the changes to the persistent store using the save() method of the managed object context.

Delete

1. Fetch the desired objects from the persistent store using a fetch request.
2. Delete the fetched objects from the managed object context using the delete(\_:) method.
3. Save the changes to the persistent store using the save() method of the managed object context.

Here is an example code snippet for creating and saving a new managed object in Core Data:

// Create a new instance of your managed object class

let newPerson = Person(context: managedObjectContext)

// Set the properties of the managed object to the desired values

newPerson.firstName = "John"

newPerson.lastName = "Doe"

newPerson.age = 30

// Add the managed object to the managed object context

managedObjectContext.insert(newPerson)

// Save the changes to the persistent store

do {

try managedObjectContext.save()

} catch let error {

print("Could not save: \(error)")

}

This is just a basic example, but the same principles apply for performing other CRUD operations in Core Data.

**2] What is SQLite? Crud operation steps**

* SQLite is a lightweight, file-based database management system that is widely used in various applications.
* It is easy to use, portable, and can handle a small to a medium-sized database. SQLite is available on different platforms, such as Windows, Mac OS, and Linux.
* To perform CRUD (Create, Read, Update, Delete) operations in SQLite, we need to use SQL statements.
* SQL is a standard language for interacting with relational databases.

import SQLite

// Database create

let dbPath = NSSearchPathForDirectoriesInDomains(.documentDirectory, .userDomainMask, true).first!

let db = try! Connection("\(dbPath)/student.sqlite3")

let students = Table("students")

let id = Expression<Int>("id")

let name = Expression<String>("name")

let age = Expression<Int>("age")

try! db.run(students.create { t in

t.column(id, primaryKey: true)

t.column(name)

t.column(age)

})

if let count = try? db.scalar(students.count) {

print("Table created with \(count) rows")

} else {

print("Table not created")

}

// Insert data

let students = Table("students")

let id = Expression<Int>("id")

let name = Expression<String>("name")

let age = Expression<Int>("age")

let insert = students.insert(name <- "John Doe", age <- 20)

do {

let rowId = try db.run(insert)

print("Inserted row id: \(rowId)")

} catch {

print("Insertion failed: \(error)")

}

for student in try! db.prepare(students) {

print("id: \(student[id]), name: \(student[name]), age: \(student[age])")

}

// Display data

do {

let query = students.select(id, name, age)

for student in try db.prepare(query) {

print("id: \(student[id]), name: \(student[name]), age: \(student[age])")

}

} catch {

print("Select failed: \(error)")

}

// update the record

let studentToUpdate = students.filter(id == 1)

let update = studentToUpdate.update(name <- "Jane Doe", age <- 21)

do {

let changes = try db.run(update)

print("Updated \(changes) row(s)")

} catch {

print("Update failed: \(error)")

}

for student in try! db.prepare(students) {

print("id: \(student[id]), name: \(student[name]), age: \(student[age])")

}

// delete record

let studentToDelete = students.filter(id == 1)

let delete = studentToDelete.delete()

do {

let changes = try db.run(delete)

print("Deleted \(changes) row(s)")

} catch {

print("Deletion failed: \(error)")

}

for student in try! db.prepare(students) {

print("id: \(student[id]), name: \(student[name]), age: \(student[age])")

}

**3] What is core data**

* CoreData is the framework provided by Apple to save, track, filter, and modify the data within the iOS applications.
* It is not the database, but it uses SQLite as its persistent store.
* It is used to manage the model layer object in our application.
* It manages the object graphs, tracks the changes in the data, and modifies the data on the user interactions.
* In the previous section of the tutorial, we have seen how we can use user defaults to store the short pieces of data.
* However, in this section of the tutorial, we will discuss how to use the actual database framework like CoreData to store, modify, and filter the user's data in the application's database.

**How to use CoreData in iOS application**

* Here, we will create an iOS application that includes CoreData as a framework to store and persist the data objects.
* Let's create a single-view iOS application to demonstrate the basics of CoreData. To enable the app to use CoreData, we must check the Use CoreData option displayed at the bottom.

**Adding the Entity to CoreData Model**

* We can add an entity into the xcdatamodeld file by selecting the option Add Entity given at the bottom of the file.
* In the right pane of the file, we can add attributes, relationships, and fetched properties to the Entity.
* Here, we have created a Student entity and added three attributes id, age, and name in the model, as shown in the below image.
* Now, we have created our model Student. Let's add some records to this model. The model will be saved to the CoreData.

To add records to the model, we need to follow the following steps.

* Instantiate the persistentContainer.
* Create the context object.
* Create an entity object.
* Create a new record object.
* Set values for the records for each key.

To refer to the persistentContainer, we need to instantiate the AppDelegate. The instance of the AppDelegate is formed using the following code.

let delegate = UIApplication.shared.delegate as! AppDelegate

To create the context, we can use the persistentContainer reference using the below code.

let context = delegate.persistentContainer.viewContenxt

Now, we need to create the entity using the context reference, we just created.

let entity = NSEntityDescription.entity(forEntityName: "Student", in: context)

Now, we need to create the new Student record as the entity object.

let newStudent = NSManagedObject(entity: entity!, insertInto: context)

Now, let's add some records to the newly created entity object.

newStudent.setValue("John", forKey: "name")

newStudent.setValue(23, forKey: "age")

newStudent.setValue(1, forKey: "id")

We have created an AppDelegate object, the context, entity, and entity object. We have also set the values for the newly created entity object.

Now, we need to save the data inside CoreData. To save the data, we use the context object to save the context. We have to wrap this code with the try-catch block.

**ViewController.swift**

1. **import** UIKit
2. **import** CoreData
4. **class** ViewController: UIViewController {
6. override func viewDidLoad() {
7. **super**.viewDidLoad()
8. // Do any additional setup after loading the view.
10. let appDelegate = UIApplication.shared.delegate as! AppDelegate
11. let context = appDelegate.persistentContainer.viewContext
12. let entity = NSEntityDescription.entity(forEntityName: "Student", in: context)
13. let newStudent = NSManagedObject(entity: entity!, insertInto: context)
14. newStudent.setValue("John", forKey: "name")
15. newStudent.setValue(23, forKey: "age")
16. newStudent.setValue(1, forKey: "id")
18. **do**{
19. **try** context.save()
20. }**catch**{
21. debugPrint("Can't save")
22. }
24. }
25. func fetchData(){
26. let request = NSFetchRequest<NSFetchRequestResult>(entityName: "Student")
27. request.returnsObjectsAsFaults = **false**
28. **do**{
29. let result = **try** context.fetch(request)
30. **for** data in result{
31. debugPrint((data as AnyObject).value(forKey: "name") as! String)
32. debugPrint((data as AnyObject).value(forKey: "id") as! Int16)
33. debugPrint((data as AnyObject).value(forKey: "age") as! Int32)
34. }
35. }**catch**{
36. }
37. }
38. }
39. }

**Fetching the records**

Fetching the records from the CoreData is very simple. We need to instantiate the NSFetchRequest class and create a request object. We can pass this request object into the fetch() method for NSManagedContext reference.

1. let result = **try** context.fetch(request)

The following code can be used to fetch the data from the CoreData.

1. let request = NSFetchRequest<NSFetchRequestResult>(entityName: "Student")
2. request.returnsObjectsAsFaults = **false**
3. **do**{
4. let result = **try** context.fetch(request)
5. **for** data in result{
6. debugPrint((data as AnyObject).value(forKey: "name") as! String)
7. debugPrint((data as AnyObject).value(forKey: "id") as! Int16)
8. debugPrint((data as AnyObject).value(forKey: "age") as! Int32)
9. }
10. }**catch**{
11. }

**Chap - 5**

**1] Write steps to implement a map view**

To implement a map view in iOS, follow these steps:

1. Open Xcode and create a new project.
2. Choose a Single View App template and enter a name for your project.
3. Go to the Main.storyboard and add a MapKit view to your app’s main view.
4. Select the MapKit view and open the Attributes Inspector.
5. Set the Map Type attribute to Standard, Satellite, or Hybrid.
6. Set the Region attribute to define the initial viewable region of the map.
7. Create an IBOutlet for the MapKit view in your view controller.
8. Add the MapKit framework to your project by selecting your project in the project navigator and selecting the General tab.
9. In the Linked Frameworks and Libraries section, click the + button and select the MapKit.framework.
10. In your view controller, import the MapKit framework by adding the following import statement at the top of your file:

import MapKit

11. In your view controller, set the MapView delegate to self.

12. Implement the MKMapViewDelegate protocol methods in your view controller to handle map events and add annotations to the map.

13. Set the map’s region to the user’s location by calling the setRegion method on the MapView object.

14. Optionally, add custom annotations to the map by creating a new class that conforms to the MKAnnotation protocol and adding instances of the annotation class to the map using the addAnnotation method.

15. Run your app and test the map view.

These are the basic steps required to implement a map view in iOS using MapKit. There are many other customization options available for MapKit, such as adding overlays, drawing routes, and using custom map tiles. You can refer to the MapKit documentation for more information on these topics.

**2] what are GeoCoding and reverse GeoCoding**

GeoCoding and reverse GeoCoding are features of the iOS MapKit framework that allow you to convert between geographic coordinates and physical addresses, and vice versa.

GeoCoding is the process of converting a physical address into geographic coordinates, such as latitude and longitude. This is useful for displaying the location of an address on a map or for finding nearby locations based on a user’s address. In iOS, you can use the CLGeocoder class to perform GeoCoding. You simply provide the address information to the CLGeocoder object and it returns the corresponding geographic coordinates in the form of a CLLocation object.

Reverse GeoCoding is the opposite process of GeoCoding. It involves converting geographic coordinates (such as latitude and longitude) into a physical address. This is useful for displaying the address of a location on a map or for obtaining information about a location based on its coordinates. In iOS, you can use the CLGeocoder class to perform reverse GeoCoding. You simply provide the CLLocation object containing the geographic coordinates to the CLGeocoder object, and it returns a CLPlacemark object containing the corresponding physical address information.

Both GeoCoding and reverse GeoCoding are useful features for location-based apps and are commonly used in various types of iOS apps such as maps, navigation, and social networking apps.

**Forward and Reverse Geocoding**

Geocoding is the process of going from a set of latitude and longitude coordinates to a human readable address and vice versa. Forward geocoding means you start with an address or location (such as Boston, MA) and end up with latitude and longitude coordinates. Reverse geocoding is the process of going from latitude and longitude coordinates back to a human-readable address.

Before iOS 5, developers only had access to reverse geocoding APIs available in Map Kit. With the introduction of iOS 5, however, the Map Kit APIs have been deprecated and Apple engineers added both forward and reverse geocoding to the Core Location framework. Not only does iOS 5 provide unique access to forward geocoding APIs, but there is no longer a dependency on Map Kit for these processes.

**Geocoding Benefits**

One of the major advantages of using the iOS 5 geocoding APIs is the fact that they are inherently locale based. For example, if my phone is set to Japanese as my native language and I’m visiting a friend in the United States, when I perform a geocoding operation to convert coordinates to a physical address, the result is returned in the native language of my phone (Japanese). This involves not only translating the language but also reformatting the order in which addresses are communicated.

Additionally, the forward geocoding APIs are form agnostic, meaning they really don’t care what language or format an address is entered in. The geocoding APIs will automatically handle any known format based on the language settings of the device and handle the conversion as necessary.

As a developer working with the geocoding APIs, you don’t have to do anything special to make your app work with different geocoding languages.

**Geocoding Drawbacks**

One of the biggest drawbacks to the geocoding API stems from one of its great advantages. All of the geocoding operations are handled in the cloud, meaning the conversions do not happen on the device. Now, this is undeniably an advantage because your device is not wasting precious power and resources to handle the conversion. Additionally, as new conversion information and techniques become more accurate, Apple can simply update their APIs in the cloud giving your app even better performance down the road. The drawback is your app must have an Internet connection to use the geocoding APIs. That means if your app is running in airplane mode or on a Wi-Fi-only device that’s not connected to a Wi-Fi hotspot, you won’t have access to geocoding services and should plan accordingly.

**Reverse Geocoding**

Reverse geocoding is the process of converting a CLLocation into a CLPlacemark. Remember that the CLPlacemark contains the CLLocation, CLRegion, and an NSDictionary for the address. So while both geocoding techniques create a CLPlacemark, the geocoding process CLGeocoder simply fills in the blanks.

The following example demonstrates how to convert a CLLocation into a CLPlacemark using reverse geocoding. Remember, because the monitoring services return CLLocation objects when a new update is performed, you can easily obtain an address for a user’s location by starting the standard location service, obtaining their current location, and then reverse geocoding that location with the CLGeocoder.

**import** UIKit

**import** CoreLocation

**class** ViewController: UIViewController,CLLocationManagerDelegate {

**var** clm : CLLocationManager?

**var** geo : CLGeocoder?

**var** places : CLPlacemark?

**override** **func** viewDidLoad() {

**super**.viewDidLoad()

**self**.geo = CLGeocoder()

**self**.clm = CLLocationManager()

**self**.clm?.delegate = **self**

**self**.clm?.desiredAccuracy = kCLLocationAccuracyBest

**self**.clm?.requestWhenInUseAuthorization()

**self**.clm?.startUpdatingLocation()

}

**func** locationManager(\_ manager: CLLocationManager, didUpdateLocations locations : [CLLocation]) {

**var** myloc = locations.first

//print(myloc?.coordinate.latitude)

//print(myloc?.coordinate.longitude)

/\* geo?.geocodeAddressString("Surat,India", completionHandler: { \_,\_ in

do

{

print(locations.last?.coordinate.latitude)

print(locations.last?.coordinate.longitude)

}

})\*/

geo?.reverseGeocodeLocation(CLLocation(latitude: 19.0176147,longitude: 72.8561644)){

(placemarks,error) **in**

**if** **let** placemarks = placemarks {

**self**.places = placemarks.first

print(**self**.places?.country! **as** **Any**)

print(**self**.places?.administrativeArea! **as** **Any**)

print(**self**.places?.postalCode! **as** **Any**)

}

}

}

}

**3] what are annotation and callouts**

In iOS, annotations and callouts are used to add markers and additional information to a map view. Here is an explanation of each:

1. Annotations: Annotations are graphical objects that represent a point of interest on a map. Annotations are typically used to mark locations, such as landmarks or businesses. An annotation is represented by an instance of the MKAnnotation protocol, which contains the latitude and longitude of the location, as well as a title and subtitle that describe the annotation. Annotations can also include custom images and callouts.
2. Callouts: A callout is a graphical object that is displayed when a user taps on an annotation. Callouts provide additional information about the location that the annotation represents. By default, a callout displays the title and subtitle of the annotation, but it can also include custom views and buttons. A callout is represented by an instance of the MKAnnotationView class, which is a subclass of UIView. Callouts can be customized by providing a custom view for the callout in the viewForAnnotation method of the map view's delegate.

Together, annotations and callouts provide a way to display and interact with information on a map in iOS. Annotations mark specific locations on the map, while callouts provide additional details about each location. With custom images and views, developers can create unique and engaging maps for their iOS apps.

**import** UIKit

**import** CoreLocation

**import** MapKit

**class** ViewController: UIViewController {

**@IBOutlet** **weak** **var** mapkitview: MKMapView!

**override** **func** viewDidLoad() {

**super**.viewDidLoad()

**let** location : CLLocationCoordinate2D = CLLocationCoordinate2D(latitude: 21.1702 , longitude: 72.8311)

**let** span: MKCoordinateSpan = MKCoordinateSpan(latitudeDelta: 0.05, longitudeDelta: 0.05)

**let** region: MKCoordinateRegion = MKCoordinateRegion(center: location, span: span)

mapkitview.setRegion(region, animated: **true**)

**self**.mapkitview.showsUserLocation = **true**

**let** ant : MKPointAnnotation = MKPointAnnotation()

ant.title = "Where am I?"

ant.subtitle = "I am at VNSGU"

ant.coordinate = CLLocationCoordinate2D(latitude: 21.1702, longitude: 72.8311)

mapkitview.addAnnotation(ant)

}

}