Q1:- Step by step explain how to use horizontal stack and vertical stack view.

Ans:- Horizontal Stack and Vertical Stack are also common layout views in iOS app development. Here's how you can use them in Xcode:

1. Open Xcode and create a new project.
2. Open the Storyboard and drag a Horizontal Stack or Vertical Stack view onto your view controller.
3. Drag and drop any views that you want to be a part of the stack onto the stack view. They will automatically align horizontally or vertically, depending on which stack view you are using.
4. To customize the spacing between views in the stack, select the stack view and adjust the Spacing property in the Attributes Inspector.
5. You can also adjust the alignment of the views in the stack using the Alignment property. This will determine how the views are aligned within the stack view.
6. You can add constraints to the stack view to ensure that it resizes correctly on different device sizes. Select the stack view and then click the "Add New Constraints" button at the bottom of the storyboard.
7. Once you've added constraints, you can preview how the stack view will look on different device sizes by selecting the View as: option at the bottom of the storyboard and choosing a different device.
8. Finally, you can add additional stack views within the original stack view to create nested layouts, allowing you to create more complex layouts with ease.

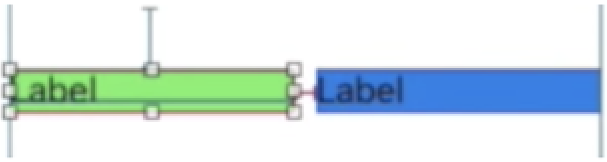
Overall, horizontal and vertical stack views are powerful layout tools that can simplify your app development process and make it easier to create dynamic and responsive user interfaces.

Q2:- With appropriate example and diagram explain “Content Hugging Priority” and “Content Compression Priority”.

Ans:- The priority really come in to play only if two different constraints conflict. The system will give importance to the one with higher priority. So, Priority is the tie-breaker in the autolayout world.

1. Content Hugging Priority :

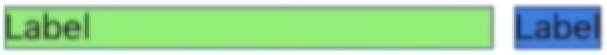
Larger the content hugging priority , the views bound will hug to the intrinsic content more tightly preventing the view to grow beyond its intrinsic content size. Setting a larger value to this priority indicates that we don’t want the view to grow larger than its content.



horizontal content hugging priority: green(251)- blue(251)

In above example, we set both label leading, trailing, top and bottom but not set width constraint. So here conflicts will occur. You can see red line between two label showing conflict.

Solution : If we have label's content hugging priority higher than there will be no conflicts as hight priority label will not grow its size more than its content size. See below image :



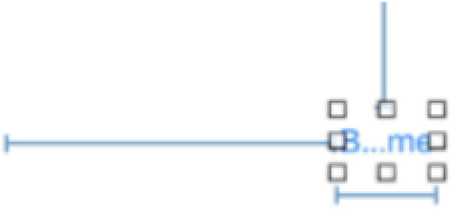
horizontal content hugging priority: green(250)- blue(251)

Blue label has more content hugging priority (251) than green label(250). Blue label's width will be set fixed as its content size.

2. Content Compression Resistance :

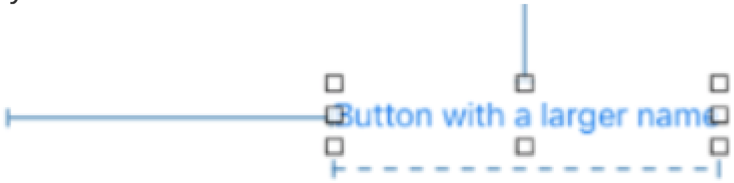
Setting a higher value means that we don’t want the view to shrink smaller than the intrinsic content size. It's simple : Higher the priority means larger the resistance to get shrunk.

Example . One button having large name and auto layout is set on button as its width is become 40. So, button's content is not readable. See image :



Button horizontal compression resistance is 750 and width constraint priority is 1000.

For Solution, let's change horizontal compression resistance to 1000 and width constraint priority less than 1000 i.e 999 . Now see effect on below image :



Button horizontal compression resistance is 1000 and width constraint priority is 999.

Q4:- Write and explain Swift code to pass data one view controller to another view controller using segue.

Ans:- In order to pass data from one view controller to another view controller using a segue, you can follow these steps:

1. Create the source view controller that will pass the data.
2. Create the destination view controller that will receive the data.
3. Create a segue between the two view controllers in the storyboard.
4. Add a unique identifier to the segue in the storyboard.
5. Override the prepare(for:sender:) method in the source view controller to pass the data.

Here is an example Swift code to pass data using a segue:

swift

Copy code

```
// Source View Controller

class SourceViewController: UIViewController {

    var data: String = "Hello, World!"

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "ShowDestination" {
            let destinationVC = segue.destination as! DestinationViewController
            destinationVC.dataReceived = self.data
        }
    }

    // Trigger the segue when a button is pressed
    @IBAction func buttonPressed(_ sender: Any) {
        performSegue(withIdentifier: "ShowDestination", sender: self)
    }
}
```

```
// Destination View Controller

class DestinationViewController: UIViewController {

    var dataReceived: String?

    override func viewDidLoad() {
        super.viewDidLoad()

        if let data = self.dataReceived {
            print("Data received: \(data)")
        }
    }
}
```

In the above code, we have a SourceViewController that contains a data property. When a button is pressed, it triggers the segue with the identifier "ShowDestination". In the prepare(for:sender:) method, we check that the segue has the correct identifier, then we cast the destination view controller to a DestinationViewController and set its dataReceived property to the data property of the source view controller.

In the DestinationViewController, we have a dataReceived property that will receive the data passed from the source view controller. We can access this data in the viewDidLoad() method and print it to the console.

Overall, this is a simple way to pass data between view controllers using segues in Swift.

Q6:- What is use of UINavigationController? Write steps to use UINavigationController in your application.

Ans:- UINavigationController is a container view controller that manages a stack of view controllers and provides a navigation interface for users to move between them. It is a fundamental building block for many iOS apps, and is commonly used in apps with a hierarchical structure of content.

The UINavigationController provides a navigation bar at the top of the screen, which displays the title of the current view controller and allows the user to navigate to other view controllers in the stack. It also provides a back button to allow the user to navigate back to the previous view controller.

Here are the steps to use UINavigationController in your application:

1. Create a new Xcode project and select the "Single View App" template.
2. In the storyboard, select the view controller and choose "Editor > Embed In > Navigation Controller" from the menu. This will embed the view controller in a new navigation controller.
3. Add additional view controllers to the storyboard and connect them to the navigation controller using segues. You can do this by control-dragging from a button or table view cell to the destination view controller, and choosing the appropriate segue type (e.g. "push" for a navigation stack).
4. Set the title of each view controller using the navigation item's title property. This will display the title in the navigation bar when the view controller is pushed onto the stack.
5. Optionally, customize the appearance of the navigation bar by setting properties such as the bar tint color, title text color, or back button appearance. This can be done in the storyboard or in code using the UINavigationController's navigationBar property.

Here is an example of how to set the title of a view controller in the navigation bar:

swift

Copy code

```
class MyViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        self.navigationItem.title = "My Title"
    }

}
```

In the above code, we override the viewDidLoad() method of our view controller and set the navigationItem's title property to "My Title". This will display the title in the navigation bar when the view controller is pushed onto the stack.

In summary, UINavigationController is a powerful tool for managing a stack of view controllers in your iOS app. By following these steps, you can easily add a navigation interface to your app and allow users to navigate between different screens of content.

Q7:- With appropriate example and diagram explain navigation controller.

Ans:- The Navigation Controller can be defined as the container view controller that maintains a stack of View Controllers for navigating hierarchical content. It is an instance of the UINavigationController class, which inherits UIViewController.

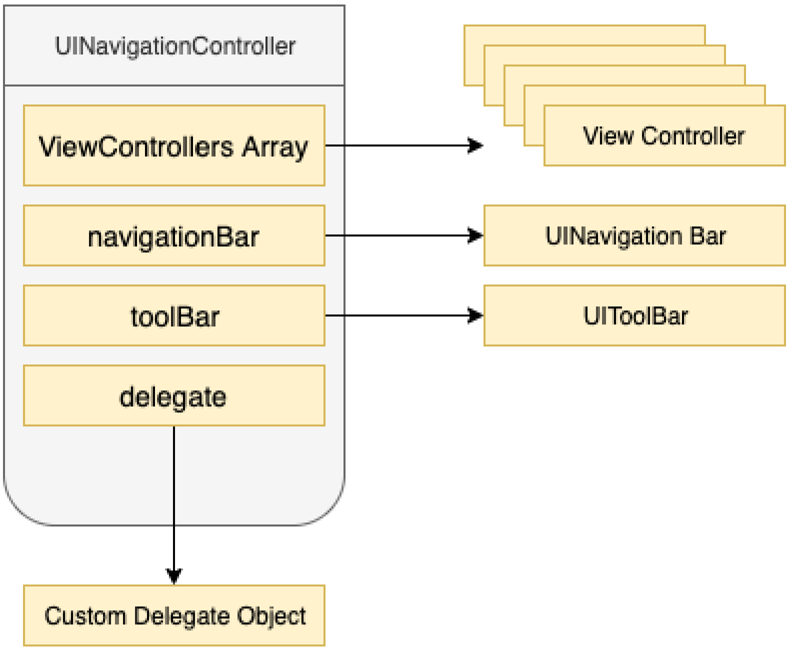
The Navigation Controller manages one or more child view controllers in the navigation interface. Navigation Controllers are being used in almost every iOS application. Even though one or more child view controllers are managed into navigation stack, only one view controller appears on-screen at an instance. Selecting an item in the view controller pushes a new view controller on the screen. This process is animated and therefore hides the previous view controller. Let's look at the navigation interface used in the settings app on iOS.

All the View Controllers embedded in the Navigation Controller contain a navigation bar that contains the title of the view controller and back button. The top view controller is removed from the navigation stack by tapping the back button. However, the back button is not provided for the root view of the stack. Navigation Controller manages the View Controllers in the ordered array where the first item is considered as the root view controller and also the bottom of the navigation stack. The last item in the array is the topmost view controller, which is currently being displayed. We can push or pop View Controllers into the stack using the methods of UINavigationController class.

The navigation controller also manages the navigation bar for all the view controllers of the navigation stack, which always appears on the top of the screen. The navigation controller also manages an optional toolbar on the bottom of the screen.

A delegate object maintains the behavior of a navigation controller. It can provide the custom animations to pushing and popping the view controllers. It can also specify the preferred orientation for the navigation interface.

The Following image shows the objects managed by a navigation controller.



A Navigation Controller adopts the view of the topmost view controller in the navigation stack. We can access this view by using the view property of the navigation controller. However, the view contains the navigation bar, the content view, and the optional toolbar. The content of the navigation bar and the toolbar changes, whereas the views do not change.

Q8:- Step by step explain how to dismiss the keyboard if user hits return key in TextField.

Ans:- Here are the steps to dismiss the keyboard if the user hits the return key in a TextField:

1. Adopt the UITextFieldDelegate protocol in your view controller.

```
class MyViewController: UIViewController, UITextFieldDelegate {

   // ...

}
```

2. Set the delegate property of your TextField to your view controller instance.

```
override func viewDidLoad() {

   super.viewDidLoad()
```

```
    myTextField.delegate = self
}
```

3. Implement the textFieldShouldReturn() method in your view controller. This method will be called when the user hits the return key in the TextField.

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {
    textField.resignFirstResponder()
    return true
}
```

4. In the textFieldShouldReturn() method, call the resignFirstResponder() method on the TextField to dismiss the keyboard.

Here's an example of how this would look in a view controller:

```
class MyViewController: UIViewController, UITextFieldDelegate {

    @IBOutlet weak var myTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        myTextField.delegate = self
    }

    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        textField.resignFirstResponder()
        return true
    }
}
```

In the above code, we set the delegate property of myTextField to self in viewDidLoad(). We then implement the textFieldShouldReturn() method to call resignFirstResponder() on the TextField when the user hits the return key. This will dismiss the keyboard.

By following these steps, you can easily dismiss the keyboard when the user hits the return key in a TextField.

Q9:- Which protocol consists of following function? With appropriate example, demonstrate its usage.func textFieldShouldReturn(_ textField: UITextField) -> Bool { textField.resignFirstResponder()

return true}

Ans:- The protocol that consists of the textFieldShouldReturn(_:) function is the UITextFieldDelegate protocol. This protocol defines methods that allow you to respond to events related to text input in a UITextField.

The textFieldShouldReturn(_:) function is called when the user hits the return key on the keyboard while editing a text field. It gives you the opportunity to perform any necessary actions in response to the return key press, such as validating input or dismissing the keyboard.

Here's an example of how you might use the textFieldShouldReturn(_:) function in a view controller:

```
class MyViewController: UIViewController, UITextFieldDelegate {

    @IBOutlet weak var myTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Set the delegate of the text field to the view controller
        myTextField.delegate = self
    }

    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        // Dismiss the keyboard when the user hits the return key
        textField.resignFirstResponder()

        // Perform any necessary actions in response to the return key press
        // ...

        // Return true to indicate that the text field should respond to the return key press
        return true
    }
}
```

In this example, we first set the delegate property of myTextField to self in viewDidLoad(). This tells the text field to send delegate messages to the view controller. We then implement the textFieldShouldReturn(_:) method to dismiss the keyboard when the user hits the return key, and perform any other necessary actions. Finally, we return true to indicate that the text field should respond to the return key press.

By implementing the textFieldShouldReturn(_:) function in your view controller and setting the delegate property of your text field to the view controller, you can customize the behavior of the text field in response to the return key press.

Q10:- Step by step explain how to dismiss the keyboard if user taps outside of the TextField.

Ans:- Here are the steps to dismiss the keyboard if the user taps outside of the TextField:

1. Create a UITapGestureRecognizer and add it to your view controller's view in viewDidLoad().

```
override func viewDidLoad() {
  super.viewDidLoad()
  let tapGesture = UITapGestureRecognizer(target: self, action: #selector(dismissKeyboard))
  view.addGestureRecognizer(tapGesture)
}
```

2. Implement the dismissKeyboard() method to dismiss the keyboard when the user taps outside of the TextField.

```
@objc func dismissKeyboard() {
  view.endEditing(true)
}
```

3. In the dismissKeyboard() method, call the endEditing(_:) method on the view to dismiss the keyboard.

Here's an example of how this would look in a view controller:

```
class MyViewController: UIViewController {
  @IBOutlet weak var myTextField: UITextField!
  override func viewDidLoad() {
    super.viewDidLoad()

    let tapGesture = UITapGestureRecognizer(target: self, action: #selector(dismissKeyboard))
    view.addGestureRecognizer(tapGesture)
  }

  @objc func dismissKeyboard() {
    view.endEditing(true)
```

```
 }
}
```

In the above code, we create a UITapGestureRecognizer and add it to the view in viewDidLoad(). We also implement the dismissKeyboard() method to dismiss the keyboard when the user taps outside of the TextField. This method calls endEditing(_:) on the view to dismiss the keyboard.

By following these steps, you can easily dismiss the keyboard when the user taps outside of a TextField.

Q11:- Write and explain Swift code to create property observer, which removes digit 9 if entered in TextField.

Ans:- Here's an example of how to create a property observer in Swift that removes the digit 9 if entered in a TextField:

```
class MyViewController: UIViewController {

    @IBOutlet weak var myTextField: UITextField! {
        didSet {
            // Add a target to the text field to monitor for changes
            myTextField.addTarget(self, action: #selector(textFieldDidChange), for: .editingChanged)
        }
    }

    @objc func textFieldDidChange(_ textField: UITextField) {
        // Remove the digit 9 if entered in the text field
        if textField.text?.contains("9") == true {
            textField.text = textField.text?.replacingOccurrences(of: "9", with: "")
        }
    }
}
```

In this code, we define a myTextField property with a didSet property observer. In the didSet block, we add a target to the text field to monitor for changes using the addTarget(_:action:for:) method.

We also define a textFieldDidChange(_:) method that is called whenever the text field's value changes. In this method, we check if the text field's value contains the digit 9 using the contains(_:) method. If it does, we remove the digit 9 using the replacingOccurrences(of:with:) method.

By using a property observer and a target action, we can monitor for changes to a TextField and modify its value in response. In this example, we remove the digit 9 if it is entered in the text field.

Q12:- What is use of UIImagePickerController? List and explain use of at least two different UIImagePickerControllerSourceType

Ans:- The UIImagePickerController is a pre-built user interface component provided by Apple in iOS that allows users to pick media from their photo library or take a new photo or video using the device's camera. It provides a standard user interface for choosing and editing images and videos.

The UIImagePickerControllerSourceType enumeration defines the types of sources that can be used with the image picker controller. Here are two examples of different UIImagePickerControllerSourceType values and their uses:

1. UIImagePickerControllerSourceType.photoLibrary: This source type allows the user to select an existing photo or video from their device's photo library. This is useful if your app needs to access media content that the user has already saved on their device.

2. UIImagePickerControllerSourceType.camera: This source type allows the user to take a new photo or video using the device's camera. This is useful if your app needs to capture new media content that the user wants to save or share.

Here's an example of how to use the UIImagePickerController to access the photo library:

```
class MyViewController: UIViewController, UIImagePickerControllerDelegate & UINavigationControllerDelegate {



  let imagePicker = UIImagePickerController()



  override func viewDidLoad() {

    super.viewDidLoad()
```

```
        imagePicker.delegate = self
        imagePicker.sourceType = .photoLibrary
        imagePicker.allowsEditing = true
    }

    @IBAction func choosePhotoTapped(_ sender: Any) {
        present(imagePicker, animated: true, completion: nil)
    }

    func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {
        // Do something with the selected photo or video
        dismiss(animated: true, completion: nil)
    }

    func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
        dismiss(animated: true, completion: nil)
    }

}
```

In this code, we define a MyViewController class that conforms to the UIImagePickerControllerDelegate and UINavigationControllerDelegate protocols. We create an instance of the UIImagePickerController and set its delegate to self.

In viewDidLoad(), we set the source type of the image picker to .photoLibrary and enable editing mode. We also define an IBAction method that presents the image picker when the user taps a button.

We implement the imagePickerController(_:didFinishPickingMediaWithInfo:) and imagePickerControllerDidCancel(_:) methods to handle the user's selection or cancellation of the image picker.

Here's an example of how to use the UIImagePickerController to take a new photo or video using the device's camera:

```
class MyViewController: UIViewController, UIImagePickerControllerDelegate &
UINavigationControllerDelegate {


  let imagePicker = UIImagePickerController()


  override func viewDidLoad() {

    super.viewDidLoad()


    imagePicker.delegate = self

    imagePicker.sourceType = .camera

    imagePicker.cameraDevice = .rear

    imagePicker.allowsEditing = true

  }


  @IBAction func takePhotoTapped(_ sender: Any) {

    present(imagePicker, animated: true, completion: nil)

  }


  func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info:
[UIImagePickerController.InfoKey : Any]) {

    // Do something with the captured photo or video

    dismiss(animated: true, completion: nil)

  }


  func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
        dismiss(animated: true, completion: nil)

    }



}
```

In this code, we use the same MyViewController class as before. However, in viewDidLoad(), we set the source type of the image picker to .camera and set the camera device to the rear-facing camera using the cameraDevice property.

We define an IBAction method that presents the image picker when the user taps a button

Q13:- How to use permissions in iOS application? Write steps for same.

Ans:- In iOS, you need to request user permissions before accessing certain features or data on the device, such as the camera, microphone, location, contacts, photos, and so on. Here are the steps to use permissions in an iOS application:

1. Identify which permissions your app requires: Before requesting permissions from the user, you should first identify which features or data your app requires access to. This will depend on the specific use case of your app.
2. Add the appropriate keys to your app's Info.plist file: In order to request permissions, you need to add the appropriate keys to your app's Info.plist file. For example, if your app requires access to the device's camera, you would add the NSCameraUsageDescription key and a corresponding message explaining why your app needs this permission.
3. Request permissions at the appropriate time: Once you have identified the permissions your app requires and added the necessary keys to your app's Info.plist file, you can request permissions at the appropriate time in your app's code. For example, if your app requires access to the device's location, you might request this permission when the user opens a particular view controller that requires location data.

Q14:- What is UIGestureRecognizer? List at least four sub classes of UIGestureRecognizer and explain use of any two.

Ans:- UIGestureRecognizer is a class in iOS that allows you to recognize gestures on a view or control. It provides a way to detect touch events, and then analyze those events to determine if they correspond to a predefined gesture. Here are four subclasses of UIGestureRecognizer:

1. UITapGestureRecognizer: This subclass recognizes single or multiple taps on a view or control. You can use it to detect when a user taps on a button, an image, or any other view.
2. UIPinchGestureRecognizer: This subclass recognizes pinch gestures on a view or control. You can use it to detect when a user is pinching to zoom in or out on an image or map.
3. UIPanGestureRecognizer: This subclass recognizes pan gestures on a view or control. You can use it to detect when a user is dragging a view or control across the screen.

4. UILongPressGestureRecognizer: This subclass recognizes long-press gestures on a view or control. You can use it to detect when a user presses and holds on a button, an image, or any other view.

Q15:- Write Swift code to detect pinch gesture and display scale, velocity. Also write Swift code to detect rotation gesture and display radians, velocity.

Ans:- Here's an example Swift code to detect pinch gesture and display scale and velocity:

```
class ViewController: UIViewController {

  override func viewDidLoad() {
    super.viewDidLoad()
    let pinchGesture = UIPinchGestureRecognizer(target: self, action:
#selector(handlePinchGesture(_:)))
    view.addGestureRecognizer(pinchGesture)
  }

  @objc func handlePinchGesture(_ gesture: UIPinchGestureRecognizer) {
    let scale = gesture.scale
    let velocity = gesture.velocity
    print("Pinch gesture detected. Scale: \(scale), Velocity: \(velocity)")
  }
}
```

Here's an example Swift code to detect rotation gesture and display radians and velocity:

```
class ViewController: UIViewController {

  override func viewDidLoad() {
    super.viewDidLoad()
    let rotationGesture = UIRotationGestureRecognizer(target: self, action:
#selector(handleRotationGesture(_:)))
    view.addGestureRecognizer(rotationGesture)
  }

  @objc func handleRotationGesture(_ gesture: UIRotationGestureRecognizer) {
    let radians = gesture.rotation
    let velocity = gesture.velocity
    print("Rotation gesture detected. Radians: \(radians), Velocity: \(velocity)")
  }
```

Q16:- With example, explain use of UISwipeGestureRecognizer and UITapGestureRecognizer.

Ans:- UISwipeGestureRecognizer and UITapGestureRecognizer are two subclasses of UIGestureRecognizer that can be used to detect different types of touch events on iOS devices.

Here's an example of how to use UISwipeGestureRecognizer:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        let swipeGesture = UISwipeGestureRecognizer(target: self, action: #selector(handleSwipeGesture(_:)))
        swipeGesture.direction = .right // Set the direction of the swipe gesture
        view.addGestureRecognizer(swipeGesture)
    }

    @objc func handleSwipeGesture(_ gesture: UISwipeGestureRecognizer) {
        print("Swipe gesture detected")
    }
}
```

In the example above, we create a UISwipeGestureRecognizer instance and add it to the view of the view controller. We also set the direction of the swipe gesture to .right. When the user performs a swipe gesture in the specified direction, the handleSwipeGesture method is called.

Here's an example of how to use UITapGestureRecognizer:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        let tapGesture = UITapGestureRecognizer(target: self, action: #selector(handleTapGesture(_:)))
        view.addGestureRecognizer(tapGesture)
    }

    @objc func handleTapGesture(_ gesture: UITapGestureRecognizer) {
        print("Tap gesture detected")
    }
}
```

In the example above, we create a UITapGestureRecognizer instance and add it to the view of the view controller. When the user performs a tap gesture on the view, the handleTapGesture method is called.

UITapGestureRecognizer can also be customized with properties such as numberOfTapsRequired and numberOfTouchesRequired to detect specific types of tap gestures, such as double taps or two-finger taps.

UNIT-5:-

Q1:- What is "Archiving" and "Unarchiving"? Classes whose instances need to be archived must conform to which protocol? Write Swift code snippet and explain "Archiving" and "Unarchiving" considering "Book" class.

Ans:- "Archiving" is the process of converting an object or a graph of objects to a format that can be stored persistently, such as in a file or database. "Unarchiving" is the reverse process of converting an archived object or graph of objects back into live objects in memory.

In order to support archiving and unarchiving, classes whose instances need to be archived must conform to the NSCoding protocol, which defines two methods:

encode(with coder: NSCoder): This method is called to encode (or serialize) the object and its instance variables to an NSCoder object.

init?(coder: NSCoder): This method is called to decode (or deserialize) the object and its instance variables from an NSCoder object.

Here's an example Swift code snippet to demonstrate archiving and unarchiving of a "Book" class that conforms to the NSCoding protocol:

```
class Book: NSObject, NSCoding {
  let title: String
  let author: String
  let pageCount: Int

  init(title: String, author: String, pageCount: Int) {
    self.title = title
    self.author = author
    self.pageCount = pageCount
  }

  // MARK: - NSCoding

  required convenience init?(coder aDecoder: NSCoder) {
    guard let title = aDecoder.decodeObject(forKey: "title") as? String,
       let author = aDecoder.decodeObject(forKey: "author") as? String
    else {
      return nil
    }
    let pageCount = aDecoder.decodeInteger(forKey: "pageCount")
    self.init(title: title, author: author, pageCount: pageCount)
  }

  func encode(with aCoder: NSCoder) {
    aCoder.encode(title, forKey: "title")
    aCoder.encode(author, forKey: "author")
    aCoder.encode(pageCount, forKey: "pageCount")
  }
}
```

In the example above, we define a Book class that has three properties: title, author, and pageCount. The class conforms to the NSCoding protocol by implementing the init?(coder:) and encode(with:) methods.

In the init?(coder:) method, we decode the properties of the object from the provided NSCoder object using decodeObject(forKey:) and decodeInteger(forKey:) methods, and then initialize a new instance of the Book class with the decoded properties.

In the encode(with:) method, we encode the properties of the object to the provided NSCoder object using encode(_:forKey:) methods.

With the Book class implemented like this, we can use NSKeyedArchiver and NSKeyedUnarchiver to archive and unarchive instances of the Book class respectively. Here's an example code snippet:

```
let book = Book(title: "The Hitchhiker's Guide to the Galaxy", author: "Douglas Adams", pageCount: 193)

let data = try! NSKeyedArchiver.archivedData(withRootObject: book, requiringSecureCoding: false)

let unarchivedBook = try! NSKeyedUnarchiver.unarchiveTopLevelObjectWithData(data) as! Book

print(unarchivedBook.title) // prints "The Hitchhiker's Guide to the Galaxy"
```

In the example above, we create an instance of the Book class, and then use NSKeyedArchiver to convert it to an NSData object using the archivedData(withRootObject:requiringSecureCoding:) method. We then use NSKeyedUnarchiver to convert

Q2:- Write Swift code create student class having properties “enrollmentNumber”, “name”, and “age”. Also write and explain swift code to store and retrieve instance of “Student” class using “Archiving”

Ans:- Here is the Swift code to create a Student class with properties enrollmentNumber, name, and age:

```
class Student: NSObject, NSCoding {

  var enrollmentNumber: String

  var name: String

  var age: Int


  init(enrollmentNumber: String, name: String, age: Int) {

    self.enrollmentNumber = enrollmentNumber

    self.name = name
```

```
        self.age = age
    }

    func encode(with coder: NSCoder) {
        coder.encode(enrollmentNumber, forKey: "enrollmentNumber")
        coder.encode(name, forKey: "name")
        coder.encode(age, forKey: "age")
    }

    required convenience init?(coder: NSCoder) {
        guard let enrollmentNumber = coder.decodeObject(forKey: "enrollmentNumber") as? String,
              let name = coder.decodeObject(forKey: "name") as? String
        else {
            return nil
        }

        let age = coder.decodeInteger(forKey: "age")
        self.init(enrollmentNumber: enrollmentNumber, name: name, age: age)
    }
}
```

Here is the Swift code to store and retrieve an instance of the Student class using archiving:

```
// Create a Student instance
let student = Student(enrollmentNumber: "2023A001", name: "John Doe", age: 21)

// Get the document directory path
let documentsDirectory = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask).first!

// Append a file name to the document directory path
let archiveURL = documentsDirectory.appendingPathComponent("student_data")
```

```
// Archive the Student instance

let success = NSKeyedArchiver.archiveRootObject(student, toFile: archiveURL.path)


if success {

    print("Student instance archived successfully!")

} else {

    print("Failed to archive Student instance!")

}


// Unarchive the Student instance

if let unarchivedStudent = NSKeyedUnarchiver.unarchiveObject(withFile: archiveURL.path) as?
Student {

    print("Enrollment Number: \(unarchivedStudent.enrollmentNumber)")

    print("Name: \(unarchivedStudent.name)")

    print("Age: \(unarchivedStudent.age)")

} else {

    print("Failed to unarchive Student instance!")

}
```

In the above code, we first create a Student instance and then get the document directory path. We append a file name to the document directory path and then use NSKeyedArchiver to archive the Student instance to the specified path. We check if the archiving was successful and then use NSKeyedUnarchiver to unarchive the Student instance from the specified path. Finally, we print the properties of the unarchived Student instance.

Q4:- What is iOS application sandbox? List and explain content/use of each directory available in application sandbox.

Ans:- iOS application sandbox is a security mechanism used by iOS to protect and isolate apps from accessing or modifying files outside of their designated container. Each iOS app has its own sandbox, which is a separate directory where the app stores its data, preferences, caches, and other related files. The app is only allowed to access files within its own sandbox, and is restricted from accessing other apps' sandboxes or the iOS system files.

Here are the directories available in the iOS application sandbox and their respective content/use:

1. App Bundle: This is the directory that contains the app's executable and resources such as images, sounds, and localization strings.

2. Documents Directory: This is the directory where the app stores user-generated content such as documents, videos, and photos. Files stored in this directory are automatically backed up to iCloud if the user enables the backup feature.

3. Library Directory:

   Caches: This is the directory where the app stores temporary files that can be regenerated if necessary. Files stored in this directory are not backed up to iCloud and can be deleted by the system if the device runs low on storage.

   Preferences: This is the directory where the app stores its preference files, such as user settings, configurations, and application states.

   Application Support: This is the directory where the app stores files that are needed for the app to function properly, but are not user-generated. For example, files containing default data or downloaded content

4. tmp Directory: This is the directory where the app stores temporary files that are only needed for a short time. The files in this directory are not backed up to iCloud and are deleted by the system when the app is closed or when the device is restarted.

Q5:- Draw iOS application state-transition diagram and explain transition among any three states

Ans:- Here is a simple iOS application state-transition diagram:

```
+-------------+       +-----------------+       +----------------+

| Not Running |  ---->  | Inactive/Suspended |  --->  | Active/Running |

+-------------+       +-----------------+       +----------------+
```

Explanation of the states:

Not Running: The application is not launched or it was running but was terminated by the system.

Inactive/Suspended: The application is in the foreground but not receiving events. This state occurs when the app is transitioning from active to background state, or from inactive to active state.

Active/Running: The application is in the foreground and receiving events.

Transition among states:

Launch: When the user taps on the app icon, the application transitions from Not Running state to Inactive/Suspended state.

Background: When the user presses the home button, the application transitions from Active/Running state to Inactive/Suspended state.

Foreground: When the user taps on the app icon from the home screen, the application transitions from Inactive/Suspended state to Active/Running state.

Q17:- Compare "Archiving" and "Core Data" to store persistence data. Explain at least two differences and two similarities.

Ans:- Archiving and Core Data are two different ways of storing persistence data in iOS applications. Here are some differences and similarities between the two:

Differences:

Data Model: Archiving is a file-based persistence mechanism, whereas Core Data is an object graph management and persistence framework. Archiving stores a collection of objects into a single file, while Core Data stores data in a SQLite database that represents an object graph model.

Querying and Sorting: Archiving doesn't provide a query language to search and sort data. In contrast, Core Data provides a powerful query language and sorting functionality, which makes it easy to search and sort data based on specific criteria.

Similarities:

Object Graph: Both Archiving and Core Data are designed to store an object graph. This means that you can store not only the properties of an object but also the relationships between objects.

Data Types: Both Archiving and Core Data can store a wide range of data types, including primitive types, custom objects, and binary data.

Overall, the choice between Archiving and Core Data depends on the requirements of the application. If the application requires simple persistence and doesn't have complex data model requirements, Archiving may be sufficient. However, if the application requires a more sophisticated data model, query and sorting capabilities, and other advanced features such as data migration, then Core Data is a better choice.

UNIT-6:-

Q1:- What is the use of CocoaPods? Discuss advantages and limitations of using external libraries.

Ans:- CocoaPods is a popular dependency manager for iOS projects. It helps developers to easily manage external libraries and dependencies that are required in their projects. Here are some advantages and limitations of using external libraries:

Advantages:

Time-saving: Using external libraries can save development time by providing pre-written code that can be easily integrated into the project.

Improved functionality: External libraries can add new and advanced features to an application that may not have been possible or too time-consuming to implement manually.

Community support: The use of external libraries allows developers to tap into the knowledge and experience of a larger development community, which can help solve problems and provide support.

Code Quality: External libraries have already been tested and are frequently updated, which can improve the quality of the code.

Limitations:

Compatibility issues: External libraries may not always be compatible with the latest version of Xcode, Swift or iOS, which can cause issues.

Code Bloat: Using external libraries can increase the size of the application, which can affect app performance and may require additional testing.

Dependency Chain: External libraries can themselves have dependencies, which can create a chain of dependencies that can make it difficult to maintain.

Security: External libraries can introduce security risks, particularly if they have not been properly vetted or if they contain vulnerabilities.

Overall, the use of external libraries has many benefits, but it's important to carefully consider the advantages and limitations when deciding to use them. Proper testing and validation are critical to ensure that the libraries are properly integrated and don't introduce any issues or security risks to the application.

Q2:- What is CocoaPods? List alternatives of CocoaPods and explain use of any two.

Ans:- CocoaPods is an open-source dependency manager for Swift and Objective-C Cocoa projects. It provides a simple way to add external libraries or frameworks to your Xcode project. CocoaPods handles the entire process of downloading, integrating, and updating the external dependencies automatically. It is a very popular tool and widely used in the iOS development community.

Here are some alternatives to CocoaPods:

Carthage: Carthage is a decentralized dependency manager for Swift and Objective-C Cocoa projects. Unlike CocoaPods, it does not have a central repository of libraries. Instead, it relies on Git repositories for each dependency. Carthage does not modify your Xcode project, but instead creates a separate directory with the compiled frameworks that you can add to your project manually.

Swift Package Manager (SPM): SPM is an official dependency manager for Swift projects, and it's integrated into Xcode. It's primarily used for managing dependencies for command-line Swift applications and server-side Swift. SPM uses a Package.swift file to specify the dependencies and their versions. It can automatically download, build, and link dependencies, making it a straightforward alternative to CocoaPods.

Gradle: Gradle is a popular build automation tool that can be used for building iOS applications. It's an open-source tool that supports multiple languages, including Swift and Objective-C. It can be used to manage dependencies and build iOS applications with multiple modules.

Bazel: Bazel is a build tool similar to Gradle that supports multiple languages, including Swift and Objective-C. It's an open-source tool that uses a build configuration file to specify dependencies and build targets.

Each of these alternatives has its advantages and disadvantages, and the choice of dependency manager depends on your project requirements and preferences.

For example, Carthage is suitable for projects that require a lightweight and decentralized approach to dependency management, while SPM is suitable for projects that require a simple and integrated solution with Xcode.

Q4:- What is REST? Explain RESTful architecture and components of RESTful Web service.

Ans:- REST stands for Representational State Transfer. It is a software architectural style that defines a set of constraints to be used when creating web services. RESTful architecture is based on the HTTP protocol and is used to build lightweight, scalable, and maintainable web services.

A RESTful web service comprises the following components:

Resource: A resource is a piece of data that can be addressed by a URI (Uniform Resource Identifier). It can be a file, a document, a database record, or any other meaningful data.

URI: A URI is a unique identifier that represents a resource. It is used to access and manipulate the resource.

HTTP Methods: HTTP (Hypertext Transfer Protocol) methods are used to perform operations on the resources. The most commonly used HTTP methods are GET, POST, PUT, and DELETE.

Representation: A representation is the format in which a resource is presented to the client. It can be in the form of XML, JSON, HTML, or plain text.

Request: A request is an HTTP request that is sent by the client to the server to perform operations on the resources.

Response: A response is an HTTP response that is sent by the server to the client in response to a request.

Hypermedia: Hypermedia is a term used to describe the links that are embedded in the resources. These links allow the client to navigate through the resources and perform operations on them.

Q6:- Write sample JSON data and XML data. Write and explain at least two differences.

Ans:- Sample JSON data:

```
{
  "name": "John Smith",
  "age": 30,
  "email": "john.smith@example.com",
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "555-555-1212"
    },
    {
      "type": "work",
      "number": "555-555-2121"
    }
  ]
}
```

Sample XML data:

```
<user>
  <name>John Smith</name>
  <age>30</age>
  <email>john.smith@example.com</email>
  <address>
    <street>123 Main St</street>
    <city>Anytown</city>
    <state>CA</state>
    <zip>12345</zip>
  </address>
  <phoneNumbers>
    <phoneNumber>
      <type>home</type>
      <number>555-555-1212</number>
    </phoneNumber>
    <phoneNumber>
      <type>work</type>
      <number>555-555-2121</number>
      </phoneNumber>
   </phoneNumbers>
</user>
```

Differences between JSON and XML:

Syntax: JSON uses a syntax that is based on JavaScript object syntax, whereas XML uses its own markup language syntax. JSON is often considered easier to read and write than XML due to its simpler syntax.

Data types: JSON supports only a limited set of data types, such as string, number, boolean, array, and object, while XML supports a wide variety of data types through its schema definition language. XML allows for more complex data modeling and validation than JSON.

Overall, both JSON and XML are commonly used formats for data interchange, but their specific usage depends on the needs of the application and the data being transferred.